SQUARE KILOMETRE ARRAY

SCIENCE DATA PROCESSOR

SKA1 SDP Vertical Prototyping and Compute Efficiency Report

Document Number	SKA-TEL-SDP-0000154
Document Type	REP
Revision	02
Author	A. Brown, T. N. Chan, K. Adamek, F. Dulwich, C. Pearson, C. Broekema, W. Armour
Date	2019-03-15
Document Classification	UNRESTRICTED
Status	Released

Name	Designation	Affiliation	Signature	
Authored by:				
Anna Brown	SDP Team Member	University of Oxford	[illegible] Anna Brown (Mar 9, 2019)	
			Date:	
Owned by:				
Jeremy Coles	SDP Project Manager	University of Cambridge	Jeremy Coles Jeremy Coles (Mar 11, 2019)	
			Date:	
Approved by:				
Jeremy Coles	SDP Project Manager	University of Cambridge	Jeremy Coles Jeremy Coles (Mar 11, 2019)	
			Date:	
Released by:				
Paul Alexander	SDP Project Lead	University of Cambridge	Paul Alexander Paul Alexander (Mar 12, 2019)	
			Date:	

DOCUMENT HISTORY

Revision	Date Of Issue	Engineering Change Number	Comments
01	2018-10-31		Prepared for M21, SDP CDR review
02	2019-03-15	ECP-SDP-190001	Prepared for M22 SDP Closeout CDR OARs addressed in this document Major: SDPCDR-51 overall efficiency estimates SDPCDR-52 operational vs. computational efficiency SDPCDR-53 Rooflines and OI SDPCDR-54 Incorrect figures SDPCDR-55 Hard to follow/obvious SDPCDR-56 summary SDPCDR-57 Risks Minor: SDPCDR-49 Need to generalise. SDPCDR-50 number equations Typo: SDPCDR-48 Typo

DOCUMENT SOFTWARE

	Package	Version	Filename
Word processor	Google Docs		SKA-TEL-SKO-0000000-01_GenDocTemplate
Block diagrams			
Google docs Add-ons	Cross Reference Table of contents List of figures		Used for figure & table numbering and references. Used for heading numbering. Used to generate list of figures and tables

ORGANISATION DETAILS

Name	SDP Consortium
Lead Organisation	The Chancellor, Masters and Scholars of the University of Cambridge The Old Schools Trinity Lane Cambridge CB1 ITN United Kingdom
Website	www.ska-sdp.org

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

N/A

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
CPU	Central Processing Unit
FFT	Fast Fourier Transform
FLOPs	Floating Point operations per second
FMA	Fast multipy-add (a+b*c)
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit (or General Processing Unit)
HPC	High Performance Computing
KNL	Knights Landing
ML	Machine Learning
NDA	Non-disclosure Agreement
OI	Operational Intensity
SDK	Software Development Kit
SDP	Science Data Processor

1 Introduction

The starting point for this report was the goal of generating one value for the compute efficiency of a typical SDP pipeline, that could be used to determine which architecture would be most suitable to use and how many nodes of that architecture would be required. An analysis of the assumptions required to produce such a single value, the current lack of constraints on this problem and the wide range of possible architectures, implementations and operating conditions which could be used have shown the calculation of such a value to be erroneous at the current time. On top of this our work has demonstrated that focusing on compute efficiency alone (when considering the current computing architecture and hardware landscape) could lead to the wrong choice of hardware or node design.

Given the complexity and diversity of the current HPC landscape focusing on compute efficiency alone is no longer enough to inform good system design. An engineer needs to consider many aspects of an algorithm's performance when trying to understand and quantify efficiency. In this document we present several different examples of how an algorithm might be limited by the hardware on which it runs. For example compute throughput, size or bandwidth to cache or main memory, levels of cache, how the hardware is attached to the compute node in which it operates or how a node is attached to the wider system. Section 3 introduces a number of these issues.

A need for additional empirical data has motivated much of the vertical prototyping work documented in this report. This work is presented in two stages: Sections 4 and 5 summarise preliminary vertical prototyping work leading to an initial estimate of compute efficiency. Section 6 summarises additional vertical prototyping work done in response to the remaining high level of uncertainty associated with this efficiency estimate, with discussion of selected issues raised in this work in Section 7. Section 8 updates the preliminary compute efficiency estimate where new data is available, and highlights assumptions and uncertainties that still remain.

While the vertical prototyping work has helped to characterize the single node performance of several key algorithms on high performance architectures, a number of uncertainties remain which make it infeasible to estimate efficiency of the entire SDP system. In particular, these relate to the how implementations for multiple algorithms in a pipeline are combined, and to uncertainties in data to be processed per node and data movement between nodes. Relevant risks related to these uncertainties are listed in Section 9 and open questions and recommendations are detailed in Section 10.

2 References

2.1 Applicable Documents

The following documents are applicable to the extent stated herein. In the event of conflict between the contents of the applicable documents and this document, **the applicable documents** shall take precedence.

[AD01] SKA-TEL-SDP-0000052, SDP Risk Register, Rev 08

2.2 Reference documents

The following documents are referenced in this document. In the event of conflict between the contents of the referenced documents and this document, **this document** shall take precedence.

Reference Number	**Reference**
[RD01]	SKA-TEL-SDP-0000086 SDP Memo 10: Estimating the SDP Computational Efficiency
[RD02]	SKA-TEL-SDP-0000184 SDP Memo 87: Degridding Optimization for SKA on NVIDIA GPUs
[RD03]	SKA-TEL-SDP-0000169 SDP Memo 72: Vertical prototyping of the gridding algorithm on GPU
[RD04]	SKA-TEL-SDP-0000185 SDP Memo 88: Characterizing FFT performance on GPUs for SKA-SDP
[RD05]	Original moving-window code, J. Romein, 2012, https://github.com/awson/Romein-gridding/
[RD06]	Adapted moving-window code, 201, https://github.com/OxfordSKA/GPU-gridding-movingWindow
[RD07]	SKA-TEL-SDP-0000182 SDP Memo 85: PIP.IMG Gridding Algorithms
[RD08]	SKA-TEL-SDP-0000183 SDP Memo 86: SKA-SDP Gridding on Graphical Processing Units
[RD09]	Kernels workshop overview, 2016, https://github.com/SKA-ScienceDataProcessor/crocodile/blob/master/KERNEL_WORK.md
[RD10]	Kernels workshop workshop page, 2016, https://indico.skatelescope.org/event/427/overview
[RD11]	SKA-TEL-SDP-0000058 SDP Memo 3: Fast Fourier Transforms

[RD12]	SKA-TEL-SDP-0000188 SDP Memo 91: Antenna gain calibration on Graphical Processing Units
[RD13]	SKA-TEL-SDP-0000187 SDP Memo 90: Comparison of convolution methods for GPUs
[RD14]	SKA-TEL-SDP-0000186 SDP Memo 89: SKA-SDP Reprojection on Graphical Processing Units
[RD15]	SKA-TEL-SDP-0000122 SDP Memo 36: Convolution Gridding on CPU, GPU and KNL
[RD16]	SKA-TEL-SDP-0000170 SDP Memo 73: Chebyshev polynomial approximation of kernels in w-projection gridding algorithm on GPU
[RD17]	SKA-TEL-SDP-0000171 SDP Memo 74: Optimisation of the w-projection gridding algorithm for FPGA using Intel OpenCL
[RD18]	SKA-TEL-SDP-0000172 SDP Memo 76: Improving the efficiency of direct visibility prediction using NVIDIA Pascal and Volta GPUs
[RD19]	SKA-TEL-SDP-0000189 SDP Memo 92: The FFT calculation via NVIDIA cuFFT library
[RD20]	SKA-TEL-SDP-0000133 SDP Memo 54: Compute Node Pipeline Efficiency Assessment Framework
[RD21]	SKA-TEL-SDP-0000135 SDP Memo 56: Hardware Scaling Co-Design Recommendations
[RD22]	Gridding code, NVIDIA, 2016, https://github.com/NVIDIA/SKA-gpu-grid
[RD23]	SKA-TEL-SDP-0000192 SDP Memo 095: Convolutional Gridding Routine: GPU port, IBM
[RD24]	SKA-TEL-SDP-0000038 Rev 03: SKA1 SDP System Sizing

3 Introduction: Modern Computer Architectures

Computer architectures have never been more complicated. Over the last few decades we have witnessed profound changes in computing technologies, moving from single monolithic CPUs to complicated heterogeneous computing systems.

As researchers and industrials make significant advances in areas such as AI/ML, designers of computing technologies respond by tailoring their products for these emerging markets, increasing the complexity and diversity of the HPC landscape even more.

Traditionally the number of floating point operations per second (FLOPs) that a device can perform has, over time, increased at a greater rate (Figure 1) than the corresponding bandwidth of a device (Figure 2). Given this, it becomes ever harder to supply processing cores with enough data to keep them busy. Thus using the small, high bandwidth regions of on-chip memory (L1, L2.. caches) becomes increasingly important when working to ensure algorithms perform optimally.

Along with this, to support the emerging demands of AI/ML applications, hardware vendors are designing products that have accelerated processing for lower precision numbers (such as 8-bit integers or fp16). This makes selecting the most efficient hardware to execute an algorithm or processing pipeline on increasingly difficult.

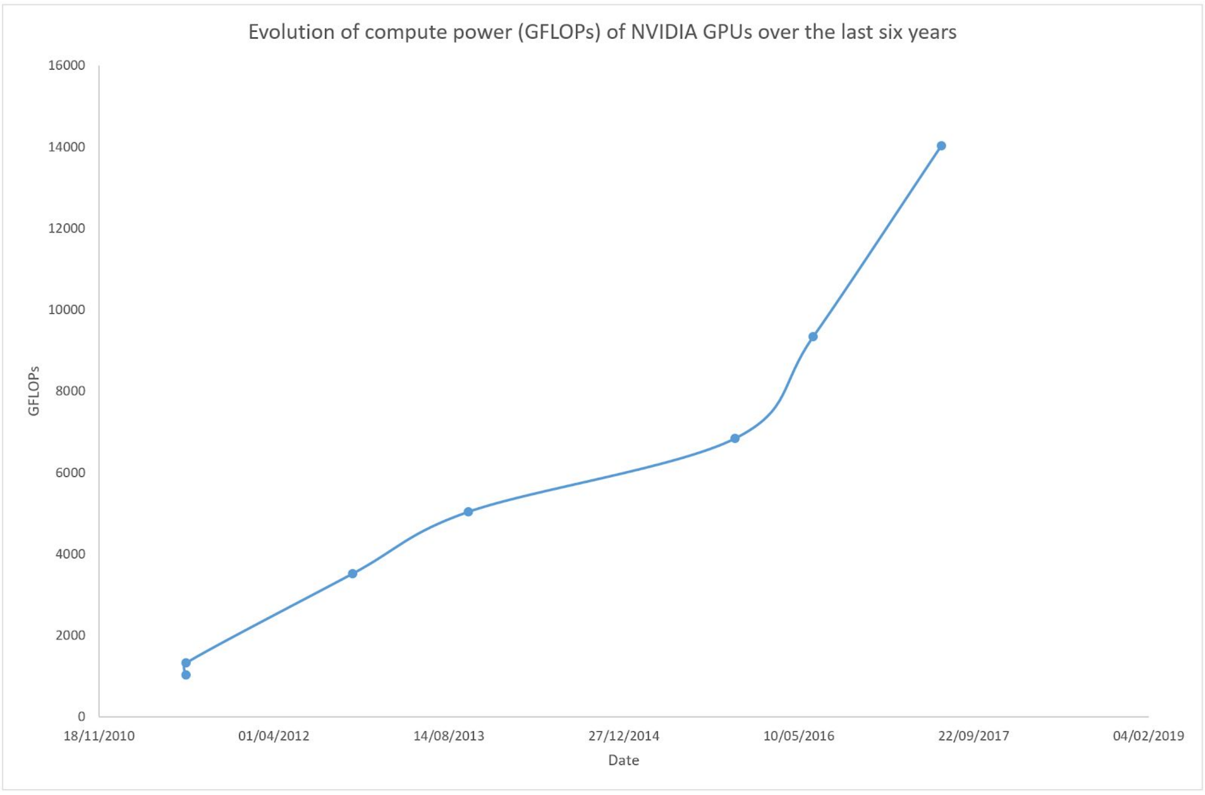


Figure 1: The evolution of peak compute performance of NVIDIA GPUs (measured in GFLOPs) over a six-year period.

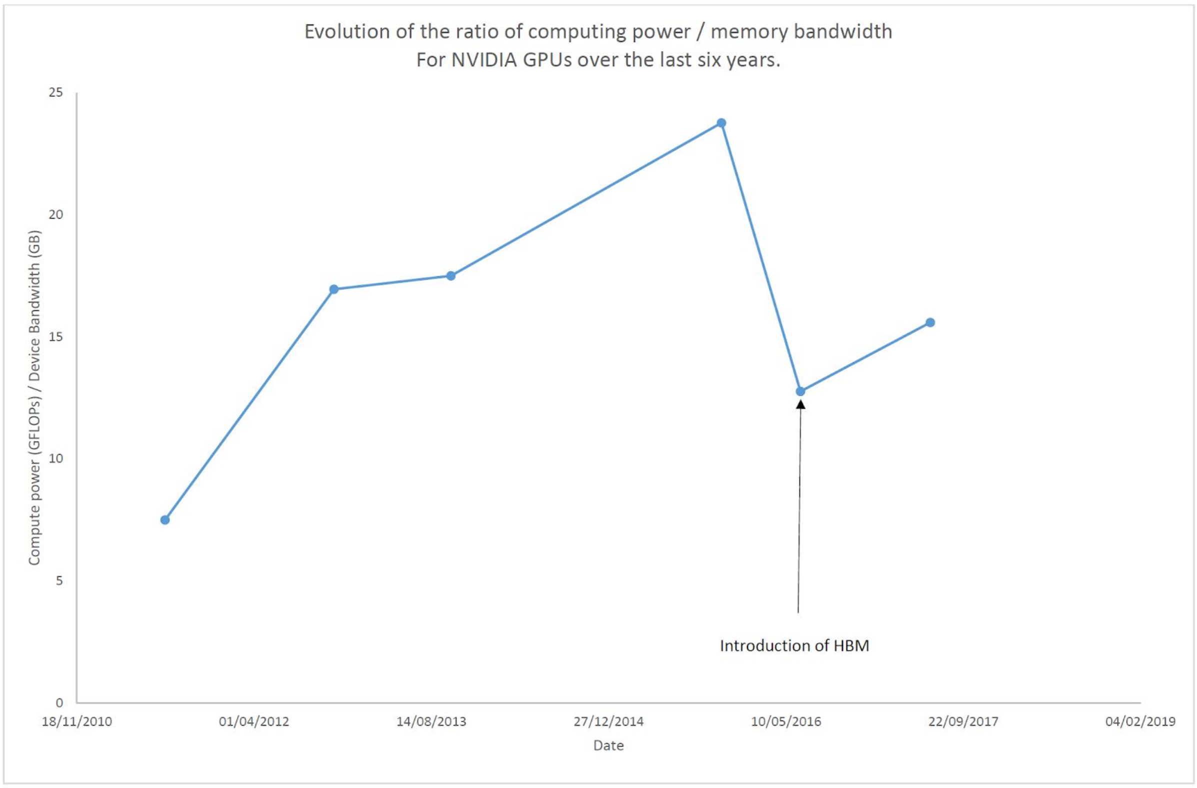


Figure 2: The evolution of the ratio of the peak computing power over the peak bandwidth (measured as FLOPs/Byte) for NVIDIA GPUs over a six-year period.

3.1 Efficiency, efficiency or efficiency

In the world of computing the word efficiency can take on different meanings. A cluster administrator might use the word efficiency to describe the utilisation of a cluster (how many nodes of the cluster are busy at any given time). A user of the cluster might use the word efficiency to describe how well their program utilises a cluster. Someone developing algorithms might use the term efficiency to describe how well a given computational algorithm utilises a piece of hardware. Others might use the word to refer to how energy efficient a data centre is.

Throughout this report we use the term efficiency to refer to the percentage of peak performance of the limiting factor (compute throughput, memory bandwidth) that an algorithm achieves on a given piece of computer hardware under consideration. Specifically, our focus is performance within a node.

How algorithms might scale beyond a single node requires additional treatment and this is given in a later section in this Report.

3.2 The efficiency of a single algorithm

The efficiency of a single computational algorithm can be determined by profiling the algorithm on target hardware and determining where the *limiting factor* for the algorithm exists. By this we pose the question, is the algorithm under consideration limited by, compute throughput, memory bandwidth, type conversion rate, the rate at which transcendentals can be calculated, etc. The algorithm could be limited by its ability to move data (bandwidth bound) or its ability to perform computations (compute bound). Some examples of this are:

- Bandwidth from main memory to processing cores.
- Bandwidth from cache (LLC, L3, L2, L1...) to processing cores.
- The rate at which computation can be performed.

For algorithms running on accelerators, the limiting factor may also be:

- PCIe bandwidth to device.
- Bandwidth from a user-managed cache (such as GPU shared memory) to processing cores.

If an algorithm requires more computational resource than is available on a single compute node then it must be spread across multiple nodes. In this case it might be that the limiting factor for the algorithm is the rate at which data can be transferred between nodes (fabric bandwidth), or the time taken to get the data onto the fabric in the first place (latency).

To define the efficiency of an algorithm (η) we must understand two different values. The first is the achieved value of the limiting factor (call this m), for example this might be the shared memory bandwidth on a GPU as measured using a profiling tool (such as nvvp in this case). The second thing that we need to understand is the maximum achievable performance of that limiting factor (call this M). Often M can be looked up, disclosed by manufactures via NDA, calculated, or measured using idealised codes. Given this we define:

$$\eta = \left(\frac{m}{M}\right) \qquad \text{(Eq. 1)}$$

Let's consider a few examples.

3.2.1 Example one – shared memory bandwidth bound

We have an algorithm that achieves 10 TB/s of shared memory bandwidth on a NVIDIA Volta GPU.

Although shared memory bandwidth is the limiting factor for this algorithm, we don't achieve the maximum theoretical bandwidth because we might have (for example) bank conflicts (Figure 3).

We know the maximum achievable shared memory bandwidth is:

80 streaming multiprocessors can make 32 (4 byte) transactions to shared memory per clock cycle. The GPU has a clock frequency of 1.455 GHz therefore

$$Shared\ memory\ bandwidth\ =\ 32\ x\ 4\ x\ 80\ x\ 1.455\ =\ 14.9\ TB/s$$

Hence our efficiency (η) is given by:

$$\eta = \left(\frac{m}{M}\right) = \left(\frac{10}{14}\right) = 0.71 \qquad \text{(Eq. 2)}$$

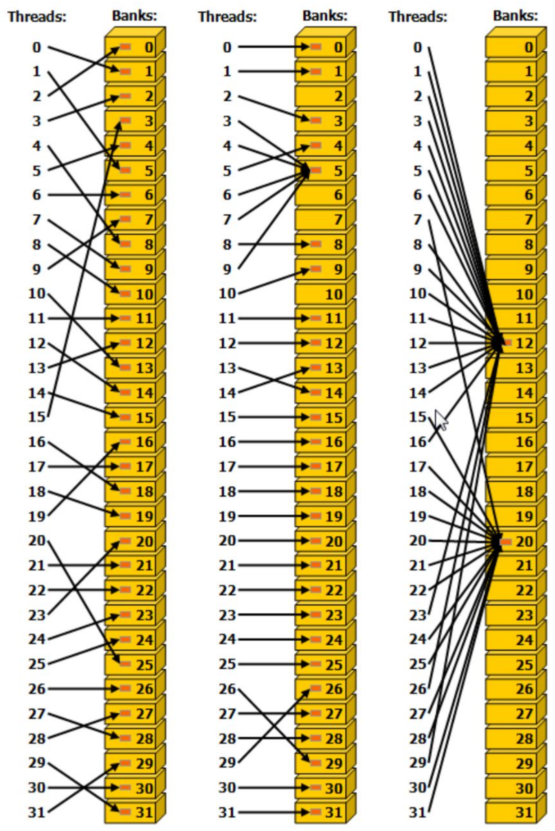


Figure 3: An example of (Left) coalesced memory access, (Middle) memory accesses with bank conflicts, and (Right) multicast memory accesses.

In this case the only thing that will change the efficiency of the algorithm is to either reduce the amount of data that the algorithm is required to move or to reuse cached data in a more efficient manner.

Any change in the number of FLOPs performed by the algorithm will have no effect on the shared memory efficiency at all.

3.2.2 Example two – compute bound

We have an algorithm that achieves 5 TFLOP/s of overall compute throughput on a NVIDIA Volta GPU.

Although our profiler tells us that the algorithm is compute bound we do not achieve the quoted maximum achievable compute throughput of ~15 TFLOP/s for single precision. The quoted maximum performance is for *fused multiply-adds* (FMAs), whereas other instructions, such as a sin() or square root, will run at different speeds. An example would be computing the sin() of a data value. On the NVIDIA Volta architecture this takes 4 clock cycles, whereas a FMA (a+b*c) can be completed in one clock.

In the above case of an achieved compute throughput of 5 TFLOP/s our efficiency (η) is given by:

$$\eta = \left(\frac{m}{M}\right) = \left(\frac{5}{15}\right) = 0.33 \quad \text{(Eq. 3)}$$

In this case the only thing that will change the efficiency of the algorithm is to either reduce the number of instructions issued by the algorithm or to change the instructions that are issued.

Any change in the amount of data that is moved by the algorithm will have no effect on the compute efficiency at all.

3.3 The Roofline model

The Roofline model gives (a) some indication of the limiting factor for a given algorithm whether an algorithm might be compute bound or bandwidth bound, and (b) facilitates the application of an improved approach for estimating the composite compute efficiency of a pipeline of science algorithms. This section is an explanation of how the Roofline Model helps. Section 8.4 provides an improved approach.

The Roofline model introduces the term Operational Intensity (OI) which is a critical factor for determining the compute efficiency of an algorithm. An algorithm to be assessed has a certain computational load in GFLOP or TFLOP and memory transfer requirements in GB or TB. The ratio of FLOP over Byte constitutes the OI. The value of this OI can be used to select the hardware device. This is possible because each hardware computing device has a pre-defined Roofline profile. A suitable device for an algorithm is one that allows the algorithm to access the theoretical compute capability peak which is anywhere on the right hand side of the device ridge point on the Roofline Diagram. Refer to [RD20] for details. This means the algorithm OI should correspond to the device OI within the range that is underneath the flat part of the roofline. In case the selected device provides the slanted part of the roofline over the algorithm OI, a Roofline Efficiency of less than 100% is incurred. In such a situation, the programmer will need to implement 2 dimensions of optimization. One is to shift the algorithm horizontally to the right hand side of the roofline diagram by increasing its OI and thus the Roofline Efficiency. The other is to raise the vertical location of the algorithm on the roofline diagram to as high as allowed by the roofline ceiling and the height is previously known as Programming Efficiency. In summary, Compute Efficiency (η_c) is the product of Roofline Efficiency (η_r) and Programming Efficiency (η_p) as expressed in Equation (1). [RD20] offers a few ideas for improving Roofline Efficiency.

$$\eta_c = \eta_r \times \eta_p \ldots\ldots\ldots\ldots\ldots\ldots \text{(Eq. 4)}$$

3.4 The case for vertical prototyping

Given the complexity of modern computer architectures, it becomes vital to study example implementations of algorithms which have been optimised specifically for the potential target architectures. Studying examples which are as close to realistic operation as possible makes it easier to understand performance characteristics such as limiting factor which may be difficult to model using theory alone.

4 Preliminary vertical prototyping work

The following list summarises known preliminary vertical prototyping work done between 2011 and 2017.

4.1 Gridding algorithm

4.1.1 An Efficient Work-Distribution Strategy for Gridding Radio-Telescope Data on GPUs, 2012

A w-gridding projection GPU implementation was developed and tested using CUDA and OpenCL on one and eight GPUs by John Romein. It was found to significantly outperform existing accelerator based gridders, using a strategy that minimizes required accessed to global memory. This implementation is referred to as the 'moving-window' code in subsequent vertical prototyping work.

Full analysis: An Efficient Work-Distribution Strategy for Gridding Radio-Telescope Data on GPUs, John W. Romein, ACM International Conference on Supercomputer (ICS'12), pp. 321-330, Venice, Italy, June 2012

Original code: [RD05]
Adapted code for comparison with new data sets: [RD06]

4.1.2 Review of gridding algorithms, 2014

A review of existing gridding algorithms in radio astronomy and medicine by Andrew Ensor, including their parallelism and memory access strategies and target architectures but not including performance results.

Full analysis: [RD07], Andrew Ensor, 2014,

4.1.3 SKA-SDP Gridding on Graphical Processing Units, 2015

Testing of gridding and degridding on GPUs by L. Barnes (NVIDIA). Different parallelisation strategies were tested on grids up to 16384x16384, number of visibilities up to 4 million and convolution kernel sizes up to 256x256 on Kepler GPUs.

Full analysis of gridding: [RD08], L. Barnes (NVIDIA), 2015

Full analysis of degridding: [RD02], L. Barnes (NVIDIA), 2015

Code: [RD22]

4.1.4 Kernels workshop, 2016

A workshop was held with the purpose of improving understanding SDP background, architecture, approach and status, including prototyping of the gridding algorithm, with input from industry partners.

Overview: [RD09]

Workshop page: [RD10]

4.1.5 Convolutional Gridding Routine: GPU port, 2017

A report by Samuel F. Antao of IBM. The w-projection gridding code as described in [RD15] was run on an IBM Minsky server with two sockets, 10 Power8 cores per socket, with NVLINK connecting each socket to two P100 GPUs. After setting memory to pinned, the code ran 15-20% faster depending on data set, which was assumed to be due to faster data transfer time as the algorithm was not modified. The usefulness of this data transfer speedup will depend on the size of the data set – if there is enough room for double buffering (ie room to fit the data that needs to be processed in one go and for data being copied to the GPU) it should be possible to hide transfer time by overlapping it with computation time. In this case, there would be nothing to be gained from the more expensive NVLINK connection. However, NVLINK should be considered if the data set is too large to fit on one GPU.

Full analysis: [RD23]

4.2 FFT algorithm

4.2.1 FFT Analysis, 2015

Analysis of the FFT algorithm by Stefano Salvini with the following findings: 1) A simple computational model of $N_{ops} = 5N log_2 N$ can be proposed but real performance will depend on the exact FFT length and implementation details. 2) If the inputs are in single precision, the effect of computing the FFT in single rather that double precision was the loss of half to one decimal digit in the cases studied, which is unlikely to have a major impact. 3) All platforms and libraries studied were consistent with each other in terms of numerical behaviour and can be considered interchangeable in that sense.

Full analysis: SDP [RD11], Stefano Salvini, 2015

4.2.2 Characterizing FFT performance on GPUs for SKA-SDP, 2016

Performance profiling of the CuFFT library on a K40m GPU by NVIDIA. Considers effects of zero padding arrays, exploiting symmetry of Hermitian matrices and batching of small FFTs. It was found that CuFFT performs best for data set dimensions that are powers of two, and that data sets smaller than 2K need to be batched to achieve good utilization of the GPU. On the K40m FFts were found to be limited by texture load instruction latency. Future GPU architectures were predicted to alleviate this bottleneck as compute gains would be greater than global memory bandwidth gains. It was then predicted that FFTs would be global memory bandwidth bound.

Full analysis: [RD04], NVIDIA, 2016

4.3 Miscellaneous algorithms

4.3.1 Antenna gain calibration on Graphical Processing Units

The algorithm for calculating antenna gain was optimised for the K40c GPU by NVIDIA, based off the StEFCal CPU algorithm. The algorithm was found to be relatively simple to implement and limited by data movement. Some small changes can be made to reduce data traffic and to make better use of caches, resulting in a 30% improvement over a simple implementation. Best performance requires a few thousand antennas. Performance falls slowly beyond about 5000 antennae. Better performance with fewer antennas can be achieved by batching several calibration computations together.

Full analysis: [RD12], NVIDIA

4.3.2 Comparison of convolution methods for GPUs
Methods for convolution of 2D images on the K40c GPU were profiled by NVIDIA. It was found that the implicit GEMM algorithm, part of the cuDNN, was outerformed by FFT-based convolution using cuFFT in almost all cases.

Full analysis: [RD13], NVIDIA

4.3.3 SKA-SDP Reprojection on Graphical Processing Units
The algorithm for reprojection of image data from one plane to another was implemented on a K40c GPU by NVIDIA. It was found to be limited by computational throughput, particularly FMA operations. Possible optimisations include using fast math operations and performing interpolation using textures, at the cost of reduced accuracy.

Full analysis: [RD14], NVIDIA

5 Preliminary estimate of compute efficiency
Estimating computational efficiency for the entirety of all SDP pipelines on as yet unavailable computational architecture is a highly complex problem. This is compounded by the fact that we have no knowledge of how the telescope will perform and what the RFI environment will ultimately be like -- as an example, the proportional time spent in different algorithms in a pipeline and therefore the overall performance characteristics, will depend on number of sources processed.

An initial estimate of the computational efficiency for SDP was made in [RD01], based in part on the preliminary vertical prototyping work outlined above. However, in the absence of sufficient data, unavoidable assumptions had to be made. This lack of data contributed to the motivation for the further vertical prototyping work outlined below.

An analysis of the assumptions made in [RD01] is given in Section 8, updated where possible with new findings from the prototyping work. Areas of uncertainty still remain -- these are outlined in Section 10 along with possible areas for further study.

6 Further prototyping work
This section contains a summary of vertical prototyping work coordinated from Oxford in 2017 and 2018.

6.1 Gridding algorithm
6.1.1 Convolution Gridding on CPU, GPU and KNL
An initial optimised version of the gridding algorithm, referred to as the `tile-based' gridding code, was produced by NAG from a very basic serial implementation. The GPU version was most performant, then a 24 core CPU implementation, then the KNL implementation.

Full Analysis: [RD15]

6.1.2 Chebyshev polynomial approximation of kernels in w-projection gridding algorithm on GPU

An attempt to optimise the w-projection gridding algorithm, which is naively bandwidth bound, by fitting Chebyshev polynomials to the convolution kernels and evaluating them on the fly; this should increase the computational work but reduce memory accesses. It was found that optimisations introduced in the tile-based version make the algorithm sufficiently compute bound on GPU to make further addition of computational work decrease rather than increase performance.

Full Analysis: [RD16]

6.1.3 NAG & NVIDIA implementation of gridding for W-projection

Describes improvements made to the tile-based gridding code by NVIDIA, largely in the area of load balancing work across GPU blocks. It also describes later improvements to the convolution kernel layout scheme, which targeted the high instruction latency of the algorithm. These combined improvements reduced the run time of the tile-based code by 12%.

Full Analysis: [RD03], Part 1

6.1.4 A comparison of moving-window and tile-based gridding codes on the same data set and GPU architecture

The tile-based code (including NVIDIA optimisations but not later improvements) was compared with an earlier gridding algorithm from 2012 by J.Romein, referred to as the `moving-window' code. Performance and accuracy are considered, in particular including the effect of assumptions made by the moving-window code. For simulation data, it was found that the two codes appeared to have comparable performance in the regime where the effect of assumptions made on the accuracy of the moving-window code is negligible. For real SKA data, the assumptions made are likely to be more valid and may have allowed the moving-window code to outperform the tile-based code.

Also included is a more thorough study of the performance limiters of both codes; the tile-based code was found to be compute bound and the moving-window code bandwidth bound. Finally, the process and complexities of preparing two different gridding algorithms for exact comparison on the same data set was documented.

Full Analysis:[RD03], Part 2

6.1.5 Optimisation of the w-projection gridding algorithm for FPGA using Intel OpenCL

Describes optimisation of the gridding algorithm for FPGA using OpenCL, with guidance from Intel. An FPGA version which is competitive with a many core CPU version could not be completed in the time available, with the largest challenge being optimising memory layouts and accesses to make best use of the chip's limited global bandwidth. An overview of FPGA optimisation areas relevant to the gridding algorithm and the effects of using OpenCL on developer time are included.

Full Analysis: [RD17]

6.2 Predict algorithm

6.2.1 Improving the efficiency of direct visibility prediction using NVIDIA Pascal and Volta GPUs

Four different configurations of the predict algorithm were studied in single and double precision on P100 and V100 GPUs and the best optimisation strategy in each case discussed.

Full Analysis: [RD18]

6.3 FFT algorithm

The time and performance in FLOP/s of the NVIDIA cuFFT library for 1D and 2D Complex to Real FFT calculation was measured. The performance of the cuFFT library for in 1D and 2D FFT is completely limited by device memory bandwidth. The compute utilisation of GPU depends on FFT length and algorithm used by the cuFFT library, but it is mostly below 15%. The FFTs lengths which are non-power of two should be avoided, by padding the data with zeros (or mean) to the nearest power of two FFT length. However this might not be applicable in all cases since padding itself has associated cost with it and it increases data size for further processing. Each such case should be investigated more closely. Smaller FFT lengths should be processed in batches otherwise GPU would be under-utilised and perform poorly.

Both previous investigations of the cuFFT library ([RD11], [RD04]) for SKA used NVIDIA K40 GPU from Kepler generation. When compared to NVIDIA TITAN V GPU from Volta generation used for this work, the K40 device memory bandwidth was 288GB/s and peak floating point performance was 5040 GFLOP/s. The TITAN V has device memory bandwidth 652GB/s (2.26x more) and peak floating point performance of 15700 GFLOP/s (3.1x more). Since cuFFT library is limited by device memory bandwidth we would expect that TITAN V would be 2.2x faster then K40 in computations of FFTs. This is indeed what we see when we have compared our results with results from [RD11]. The average speed-up for single precision is 2.6x and average speed-up for double precision is 2.3x. There are improvements in performance for specific FFT sizes. In general non-power of two FFT sizes have better performance when compared to K40. This is mostly due to improvements in algorithms rather then hardware. We can also see improvements for bigger power of two FFT sizes in single precision with speed-up 3.0x for FFT length 16384. This is again due to algorithm improvements in CUDA 9.0 package.

Full Analysis: [RD19]

7 Discussion of Vertical Prototyping Works

7.1 Programming models

As discussed, estimating computational efficiency is not straightforward, and it can be sensible to quote efficiency relative to different resources based on the limiting factor of a particular algorithm. As such a figure can be difficult to predict from theory alone, it is useful to have working, highly optimised prototype code which reflect the likely complexity of production level algorithms. These codes can be profiled to understand run time and limiting factors.

Vertical prototyping algorithms were written in C, with optimised code for accelerators written in CUDA for the NVIDIA GPU and the Intel OpenCL SDK for FPGA. These languages are sufficiently low level to allow significant control over resources required in optimised algorithms, such as, for instance, user managed caches in the GPU. The code referenced in[RD03], Part1 is a good example of the level of control over the accelerator which can be achieved using these programming models, but also illustrates the complexity of code that such optimisations can lead to. More complex code can be more expensive to maintain, take longer to develop, cost more in requiring experienced developers, and can be more vulnerable to bugs. These factors need to be balanced against the potential performance gains available.

One of the goals in the work with the Intel OpenCL SDK for FPGA, described in [RD17], was to determine how suited this language was to such development of optimised prototypes for algorithms in the pipeline. It was found that (from the perspective of a developer with no VHDL or FPGA experience) OpenCL provided a good balance between minimising development time and allowing enough control over how resources are used for optimisation purposes.

7.2 Different hardware

7.2.1 GPU vs FPGA

The majority of vertical prototyping work was done targeting the NVIDIA GPU architecture – P100 and V100. In addition, work was done to optimise the gridding algorithm on the Intel Arria 10 FPGA, as referenced in [RD17].

The FPGA is a highly customisable, highly parallel accelerator that allows for an essentially arbitrary (up to the maximum available on the device) set of resources to be assigned to a group of threads, in the form of DSPs (multiply/add units), ALUTs (logic tables) and RAMs (block memory). By contrast, A GPU contains a number of streaming multiprocessors each with a fixed number of compute units and fixed cache size, with these resources divided evenly between groups of threads.

In practice the performance bottleneck on the FPGA is often the bandwidth to main memory, which needs to be shared by the entire device. The maximum bandwidth on the development Arria 10 FPGA used in [sdp memo] was 19200 MB/s (compared to a peak theoretical bandwidth of 732GS/s for the P100 GPU) although this figure will improve with newer FPGA models.

Trying to compare GPU and FPGA implementations of SDP algorithms is made more difficult by the different potential limiting factors on the two devices, different capital costs, different energy use and operating costs and differences in developer and maintenance time. In addition, the ratio of compute capability to memory bandwidth for each device may change in the future with new models.

In addition, it is arguably much easier to calculate the value of efficiency as a fraction of the limiting factor for the GPU, where profilers can give values for the different resource usage measures described in “The efficiency of a single algorithm” section. For the FPGA, it is possible to measure bandwidth to global memory and static allocation of compute resources, but not, to the author’s best knowledge, to calculate runtime usage of compute resources. The FPGA may also have other limiting factors not considered in the “efficiency of a single algorithm” section; for example the fact that peak compute performance may not be achieved due to the clock frequency of the device being reduced as the layout of resources required on the chip becomes more complex.

7.2.2 X86 Acceleration Programming Model

The GPU is an accelerator and depends on the CPU to supply kernels (parcels of instruction) and data to perform. The current de facto open standard connecting the GPU to the CPU is PCIe. A full width of 16 PCIe lanes is normally needed to support the GPU. The memory transfer bandwidth of PCIe version 3 is about 16 GB/s in one direction. This bandwidth is far below the memory bandwidth of CPU main memory and further far below that of the GPU global memory. Unless the application is able to hide PCIe data transfer in execution time, PCIe data transfer time would add visible burden to the runtime of an application.

One measure of dealing with this deficiency is to eliminate the PCIe lane by designing the GPU as a peer to the CPU sharing the same main memory. In addition to the elimination of PCIe, GPU may be allocated a portion of the main memory that is larger than the tiny global memory normally supplied on a GPU card. AMD has been supplying such chips called APU (accelerated processing units) for a few years but those chips have up to 704 GPU cores max and are of limited use to SDP. AMD has hinted the supply of an Exa-scale APU by around 2020.

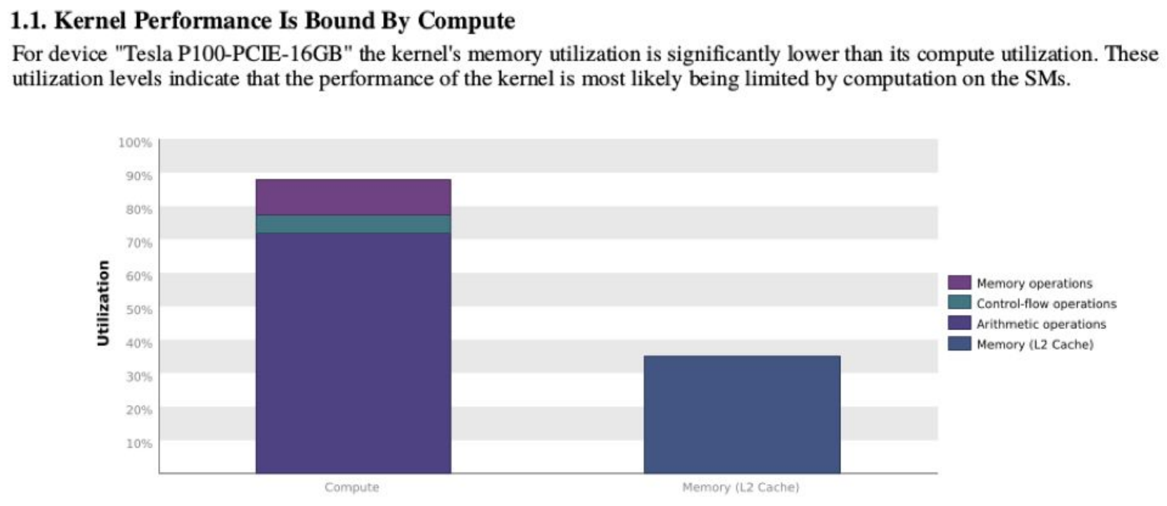
7.3 Algorithmic performance

7.3.1 Gridding

The gridding algorithm was examined in detail on GPU and in part on FPGA. Various optimised versions were considered, including tile-based versions on CPU, GPU and KNL by NVIDIA and NAG ([RD15] and [RD03], Part 1), a compute bound version which included chebyshev fitting of convolution kernels [RD16] and an FPGA version [RD17]. In addition, the tile-based algorithm was compared to an earlier, memory bound moving-window implementation ([RD03], Part 2) by J.Romein of Astron.

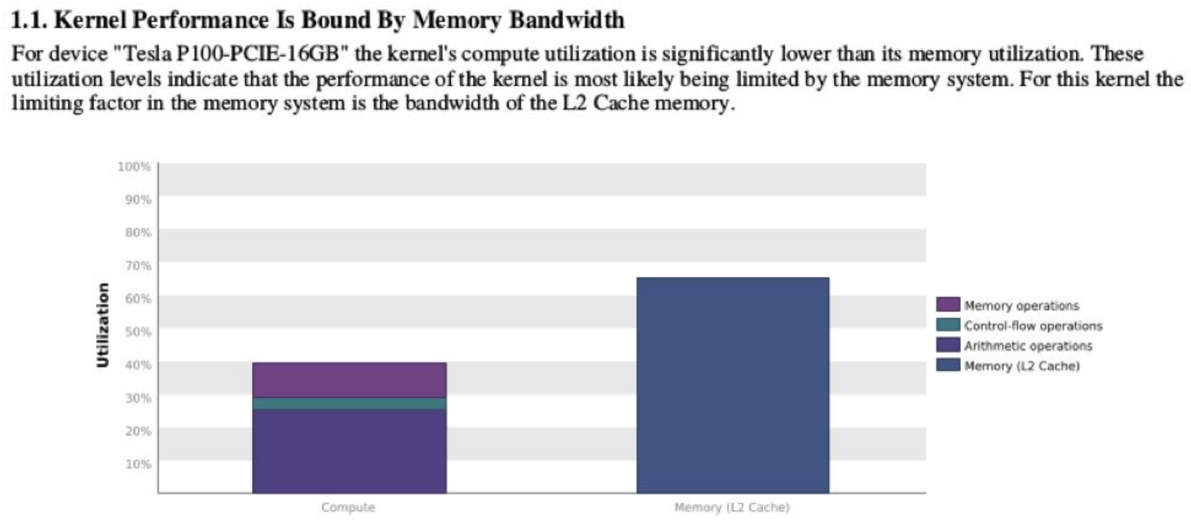
These comparisons together illustrate the important point that it is difficult to produce a single measure of efficiency, and that any such figure can be misleading. For instance, as shown in the figures below, the tile-based code was found to be compute bound on GPU, while the moving-window code was bandwidth bound. A simple measure of efficiency as percentage of compute resource used would prefer the tile-based code, where in this case the moving-window code actually had a faster run time.

In a similar vein, the addition of Chebyshev polynomial fitting to the convolution kernels used in the gridding algorithm [RD16] was an attempt to reduce the bandwidth requirements of the algorithm at the cost of adding more compute work. The “efficiency” of the code was increased – a larger percentage of the compute resources of the device were utilised – but too much compute work was added and this caused the already compute bound code to actually run slower. This shows that an algorithm that has a lower compute efficiency can be superior to an algorithm which naively uses more of the device but more poorly translates compute instructions into useful work.


1.1. Kernel Performance Is Bound By Compute

For device "Tesla P100-PCIE-16GB" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.

Figure 4: Performance limiter of tile-based code on GPU using nvvp [RD03] showing that the tile-based version is compute bound.


1.1. Kernel Performance Is Bound By Memory Bandwidth

For device "Tesla P100-PCIE-16GB" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L2 Cache memory.

Figure 5: Performance limiter of moving-window code on GPU using nvvp [RD03] showing that the moving-window version is memory bandwidth bound.

7.3.2 Predict

A "predict" algorithm is employed at various stages in a radio astronomy data processing pipeline to produce simulated visibilities, which are then used as an input model to either calibrate against or to subtract from the measured data. When used as part of the imaging process, the subtraction of predicted visibility data is done in the major cycle of the image deconvolution stage.

Typically, predicted visibilities are generated either by direct evaluation of a Fourier sum over discrete sky model components (using an explicit Measurement Equation), or by taking the fast Fourier transform of a model sky image, and then de-gridding the result onto the projected interferometer baselines. The direct-evaluation method is very flexible as it allows various effects to be modelled accurately for sources at arbitrary distance from the phase centre, although it is computationally intensive as the number of operations scales linearly with the number of sky model components and the number of baselines. The second method may be more appropriate to use for highly structured images of sky models around the phase centre. In practice, it is likely that both methods will be used by SDP, depending on the requirements of the data processing stage and complexity of the sky model in use.

We examined the performance of the first method as a case-study of how one version of a "predict" algorithm performs on modern GPU hardware, and the results of this work are discussed in detail in [RD18]. It was found that good performance (~4 TFLOPs/s) could be achieved in single precision using a single NVIDIA Titan-V GPU, although the limiting factor depended on the specific effects being modelled. In all cases the operational intensity, or the number of floating-point operations performed per byte of data loaded, is low. With time-average smearing and bandwidth smearing enabled, the performance was compute-bound and limited by the capability of the GPU's special-function units, as the smearing terms were each calculated by evaluating a sinc()-function per source and per baseline. Without smearing, particularly in the case of full polarisation, the performance was limited instead by bandwidth to cache: In this case, the optimisations discussed in [RD18], which focussed on data re-use by making better use of the GPU caches, were much more beneficial.

7.4 Scaling out

The compute efficiency of a target hardware device for a specific SDP pipeline, assuming it is determined, is not the complete story. This is simply because each SDP computer will need more than one compute node to execute the science pipelines. Distribution and gathering of data will be needed for multi-node execution modes. This involves a scheduler that may or may not be one of the executing nodes and a master-worker scheme. The runtime and memory transfer requirements of scheduling constitute an overhead. In addition, data transfers between compute nodes for maintaining data dependency and data coherence during science computation may create bottlenecks and wait states. The overall effect may bring a cluster of compute nodes to its knees by introducing a phenomenon called negative scaling- an additional node to help causes the cluster to slow down collectively. Factors affecting scaling include the execution style of the algorithm to be distributed, the dataset distribution pattern, the scheduler, and the hardware involved. If one single key performance indicator is needed, it would be the operational intensity of the algorithm- whether it is compute intensive or data intensive. A highly compute intensive algorithm would scale very well. Further detail can be found in SDP [RD21].

8 Updates to preliminary estimate of compute efficiency

As mentioned, the number of unavoidable assumptions that need to be made to estimate the compute efficiency of the entirety of a representative SDP pipeline makes it infeasible to calculate such a value at the current time. The situation is made more uncertain by the fact that compute efficiency alone may not be a good descriptor of an algorithm which has a different limiting factor, such as memory bandwidth.

In this section, the assumptions made in the preliminary estimate of compute efficiency [RD01] have been highlighted to clarify areas of uncertainty and possible areas for further study. Where possible, the assumptions made in that document have been updated with input from recent vertical prototyping activities.

The sections below refer to sections in the original document [RD01].

8.1 Experimental Evidence

Operational intensity for the gridding algorithm on one node is estimated using a measured max performance of 120 GFLOPs in [RD02] and the theoretical peak global memory bandwidth of 288 GB/s on a K40 GPU.

The following assumptions were made:

1. That the algorithm is limited by bandwidth to global memory.
2. That if an algorithm is limited by bandwidth to global memory the theoretical peak global memory bandwidth can be achieved.

Update and response:

1. The current highest performing gridding code that we are aware of is limited by L2 cache bandwidth on the GPU (The moving-window implementation of the gridding algorithm by J.Romein, running on P100, profiled in [RD03]). The tile-based code [RD03], which has similar performance as the moving-window code for simulation data, is actually compute bound.
2. As gridding is a scattered operation, it is unlikely that an implementation that worked directly out of GPU global memory would be coalesced.

The operational intensity for the FFT algorithm on one node is estimated using a measured max performance of 226 GFLOPs in [RD04] and the theoretical peak global memory bandwidth of 288 GB/s on a K40 GPU.

The following assumptions were made:

3. That the limiting factor for FFTs is global memory bandwidth and that the global bandwidth is close to being fully utilised, with utilisation level increasing with future memory architectures.

Update and response:

3. This is largely borne out by profiling work on the newer Titan V GPU card in [RD04], which finds that 1D and 2D FFTs are limited by global device memory bandwidth, with 1D FFTs at 90% utilisation of theoretical peak global bandwidth. However, importantly 2D FFTs achieve a much lower fraction of this due to the library implementation in cuFFT.

A single operational intensity figure for the entire pipeline is estimated from the operational intensities for gridding and FFT.

The following assumptions were made:

4. That gridding and FFTs take up the majority of the pipeline
5. That gridding and FFTs take up the same length of time and thus should contribute the same amount to the operational intensity figure
6. That operational intensity should be the same for different implementations of the same algorithm as optimised for CPU or GPU.
7. That the multi node implementation of each algorithm is still limited by global memory bandwidth.

Update and response:

4. We believe that it is important to understand the bottleneck for any given algorithm or processing step and then choose hardware according to this limitation. Abstracting the

limiting step for a given algorithm by only focusing on the compute efficiency for that algorithm could lead to a sub-optimal choice of hardware.

5. It cannot be guaranteed that gridding and FFTs will take the same length of time to process on all current and future computational architectures.
6. This is essentially a question of whether OI should be treated as an inherent property of an algorithm (and that therefore averaging implementations brings you closer to this true value) or a implementation dependent property under the programmer's control. It was found that two similarly performing implementations of the gridding algorithm for the same device had wildly different OIs, and in fact one was compute bound and one bandwidth bound (see [RD03], part 2). This suggests that OI cannot necessarily be treated as an invariant property of an algorithm.
7. For a large scale system algorithms that require multi-node processing are often limited by network communication.

8.2 Estimate of Future Computational Efficiency

A value for computational efficiency of the entire pipeline on future architecture is estimated from the single operational intensity value found in the previous section for the entire pipeline.

The following assumptions were made:

8. That a single figure of 0.6 for operational intensity is valid for the entire pipeline on all potential hardware.
9. That future generations of architecture will have the same limiting factor as for previous generations.
10. That the operational intensity on future generations of architecture will be the same as for previous generations.
11. That the roofline for a particular operational intensity can be reached (ie the maximum possible FLOPs for that operational intensity should be used).
12. That it is meaningful to measure what percentage of computational resources are used for an algorithm that is bound by memory resources.

Update and response:

8. See response to point 7.
9. While memory bandwidth in some form is likely to be the limiting resource, the exact form -- cache, global device memory etc -- is difficult to predict without explicit prototyping and benchmarking. It is also possible that revolutionary changes in architecture will change the limiting factor completely. Finally, the limiting factor could be changed due to explicit tailoring of architectures for the algorithm, or vice versa, through co-design.
10. It is possible that this might change if the limiting factor of the implementation changed with a new architecture.
11. In general, reaching theoretical performance limits is becoming more difficult for modern systems, due to increased parallelism, memory system heterogeneity and increased compiler complexity.
12. This is addressed in section 5 of [RD01], using memory bandwidth directly.

Experience from current generations of accelerators has shown a surprising breadth of bottlenecks that limit performance of even the gridding algorithm. From the various stages of memory hierarchy as stated above to missing hardware implementations of trigonometry functions such as sin and cos and slow performing global atomic adds. Arguing that the performance of all future hardware can be accurately determined by looking at memory bandwidth alone does not accurately reflect the complexity of modern and future hardware, or the difficulty of efficiently programming these.

This section extrapolates from two generations of Nvidia GPU that are separated by a revolutionary development in memory technology: the introduction of HBM. There is no corresponding technology lead after Pascal that we can rely on to continue this trend. It is dangerous to draw conclusions from these two data points. It has become clear recently that at least some technology leaders are seriously considering revolutionary changes in processing components (i.e. non-von Neumann architectures) for mainstream supercomputing applications. While this potentially means significantly higher efficiencies, until more details emerge of these architectures, this is impossible to say with any degree of certainty.

8.3 Using Memory Bandwidth Directly

The method for estimating the number of GPU/CPU units to be used in SDP makes the following assumptions:

- That 100 PFLOPs of required computational work is valid for the entire pipeline. This figure has been updated to 13.6 PFLOPs sustained for LOW and 11.5 PFLOPs for MID [RD24].
- As above:
    - That a single figure of 0.6 for operational intensity is valid for the entire pipeline on all potential hardware.
    - That all algorithms in the pipeline will be bound by bandwidth to global GPU memory. This is not guaranteed for single node implementations (see for example the moving-window gridding implementation). In addition, an implementation could change to being network limited when scaled up to multiple nodes.
    - That all algorithms will be running at the maximum theoretical device global bandwidth.

8.4 An Improved Model for Pipeline Composite Compute Efficiency Estimation

The Roofline Model is ideal for describing or predicting the compute efficiency of a single algorithm. An attempt is given here to apply the Roofline Model to predict the composite compute efficiency of a pipeline which consists of several applications or algorithms. The approach is based on time-weighted summation of the Operational Intensities (OI) of pipeline components and a further improvement is given in the reduction of uncertainty by reducing the spread of the OI's. Details are given in [RD20].

- The standard approach as shown in Equation (6) and (7) below would not be helpful in case the OI of components have a large spread because the hardware device always compute on an instruction basis and an algorithm is a smaller aggregate of instructions than a pipeline. A large spread of OI renders the average value meaningless.

  $Wi = Ti / \sum Ti$ where $i = 1, n$ ..... (6)

$(OI)p = \Sigma$ [$Wi * (OI)i$] .... (7)

where Wi is the runtime based weight, (OI)p is the Composite OI of the pipeline and n is the number of components of the pipeline.

An improved approach was given in [RD20] to substantially reduce the spread of OI components by treating all OI component values larger than the ridge point as the ridge point.

$(OI)p = \Sigma$ [$Wi * (OI)i$] if $(OI)i < (OI)r$ + Σ [$Wi * (OI)r$] if $(OI)i => (OI)r$ ..... (8)

where r stands for the ridge point of the Roofline Diagram.

9 SDP Risk register

The risks motivating the vertical prototyping work outlined in this document, as well as risks related to the remaining uncertainty in making an efficiency estimate for SDP pipelines, are listed here. These are defined in the SDP risk register [AD01].

9.1 Partially addressed by VP work

- [SDPRISK-392] Complexity of implementing very high performance processing components
- [SDPRISK-360] Uncertainty or inaccuracy in the Parametric Model computational estimate
- [SDPRISK-400] Hardware specifications do not reflect actual needs
- [SDPRISK-401] Actual system performance worse than modelled
- [SDPRISK-311] Cost overrun due to complexity of pipeline components

Of the above, SDPRISK-392 has been assessed to have the highest residual risk.

It is proposed that Risk Mitigation is continued during the Bridging Phase: As this report has mentioned many times, too many unknowns existed and they are targets for mitigation. Mitigation efforts will take place in 3 directions: improve science component coding efficiency, improve estimation of end-to-end runtime at scale with prototyping, and continue analysing the HPC technology landscape from a co-design perspective. End-to-end runtime at scale is the most challenging effort of all and a more holistic plan will be developed and executed during Bridging and Construction.

10 Open questions and recommendations for future work

The vertical prototyping work outlined in Section 4.0 and Section 6.0 advanced our understanding of the single node node performance of several key algorithms on currently available architectures, particularly in terms of the resource which will be the limiting factor of such implementations. However, there remain several uncertainties, particularly related to how these algorithms can be combined within a pipeline with minimum data transfer requirements, and how performance might change when multiple node implementations are considered. These uncertainties are referenced in Section 8.0 concerning preliminary compute efficiency estimates and Section 9.0 concerning the SDP risk register.

This section discusses in more detail some of the remaining open questions and possible approaches that could be used to collect more data relating to them.

10.1 Open questions

1. Which algorithm will the most time be spent in?
    a. Is the current assumption of the predict, gridding and FFT algorithms being the most expensive correct?
    b. Are the particulars of the algorithms chosen representative (eg the choice of implementing w-projection gridding). If they change significantly, could that affect the limiting factor?
2. Which pipeline will the most time be spent in?
3. How much data will each single node have to process? If different to the assumptions made here, could this change the single node performance/limiting factor?
4. In cases where different architectures are best for different algorithms in a pipeline, what is the cost of data transfer between devices?
5. Will there be a communication cost in the multi-node case such that implementations of some algorithms become network bound?
    a. Which algorithms are most likely to be implemented in ways that involve a data composition across nodes that isn't trivially parallel?

10.2 Approach to recent vertical prototyping work and recommendations for future work

10.2.1 Choosing which algorithms to study

Ideally the algorithms which will dominate the majority of the execution time within a pipeline are studied. This also needs to be scaled by the number of times a particular pipeline will run relative to the total. There is a chicken and egg problem here, but profiling existing code such as the ARL, and making predictions from theoretical models such as the parametric model can provide a starting point.

Initial profiling was done on a reduced size data set. It may be useful to also flag algorithms which are expected from theory to scale badly with increased data size on a single node or in their multiple node implementations.

10.2.2 Single node performance of a particular algorithm on a particular architecture

For each algorithm chosen, it is useful to understand the limiting factor of implementations of that algorithm on several different computing architectures. This information can be used to inform architecture choices -- each architecture will have different strengths and the limiting factor will inform which performance factor to care about when choosing.

Limiting factors were investigated for a limited set of algorithms by creating prototypes which are highly optimised for several currently available target architectures, and profiling run time as well as resource usage for each implementation. For instance, for recent GPU vertical prototyping work the limiting factor report in the Nvidia Visual Profiler was used.

A related outcome of this work should be to understand the limits of the parallelisation strategy that can be applied to implementations of each algorithm. For instance, whether there are global synchronisations which are inherent to the algorithm.

Any conclusions made from vertical prototyping work will be more accurate if there is a good data flow model available for each pipeline -- in particular, the amount of data each node will need to process can affect the limiting factor, for instance through data fitting into different levels of cache. If the amount of data isn't well defined it may be useful to profile implementations on multiple data set sizes to find the scaling with data size of each implementation.

In general when faced with unknowns relating to for instance data set size or future computer architecture performance, vertical prototyping work can at least provide options by generating multiple implementations with multiple strengths.

10.2.3 Single node performance across multiple algorithms in a pipeline, and potentially multiple architectures

It may be that future work finds that the best architecture choice is different for different algorithms in an SDP pipeline. In that case, it will be necessary to do a cost/benefit analysis of speeding up a particular algorithm vs the additional cost of data transfer between devices over PCIe/NVLINK etc. It is often the case that this data transfer cost can be significant.

Even in the case where a single accelerator architecture such as GPU is determined to be the most efficient for all algorithms in a pipeline, it may still be necessary to measure the cost of data transfer between the accelerator and host, if data cannot be kept on the same GPU for the entire length of the pipeline.

10.2.4 Multiple node performance

There is significant uncertainty in how a particular single node implementation of an algorithm will scale across multiple nodes if required to run on a data set that is too large to fit onto a single node, or would require a prohibitively long run time on a single node. Unless all algorithms in the pipeline are trivially parallel, there will not be linear scaling from one to multiple nodes. The exact scaling performance is difficult to predict from theory alone but there is a possibility that the limiting factor of the implementation will change to be network latency or bandwidth bound.

While this goes beyond the scope of vertical prototyping work, it would be useful to construct multiple-node prototype implementations for algorithms which are likely to require communication between nodes, and profile these prototypes at scale. A detailed data flow diagram which includes amount of data that needs to be transferred between nodes for each algorithm would aid in this work.

10.2.5 Predicting performance on future architectures

Implementations which have been highly optimised for modern computer hardware can often have a complex limiting factor (eg bandwidth to L2 cache). It can be difficult to predict exactly how that limiting resource will change with future versions of the hardware. In addition, it is unlikely to be just one resource which will change while all others are kept constant, which adds to the difficulty in predicting the performance and new limiting factor of an implementation on new hardware.

It is even possible that a particular implementation will be slower on new hardware, if something subtle is changed for the worse in a new architecture in order to improve performance in other areas, and the existing implementation is heavily optimised for a particular architecture. Such issues can be better predicted and/or mitigated through codesign and industry liaison.

As mentioned above, creating a number of different implementations of a particular algorithm with different limiting factors can at least provide options, making it more likely that an implementation that suits a future architecture has been studied.

10.3 Finally, while outside the scope of vertical prototyping, it is also worth noting that the performance of the interconnect between nodes and the connection between host and accelerators may also change over time and will need to be considered if these are the limiting factor of an algorithm implementation.

11 Summary

In this report we have tried to demonstrate the difficulties, subtleties and nuances in making a prediction for the computational efficiency of SDP.

In an effort to collect more empirical data to apply to this problem, a number of strongly optimised prototype implementations were created for key algorithms in the SDP pipeline, including gridding, predict and FFTs. The main focus was on the NVIDIA GPU architecture but the work also included Intel FPGA and Xeon Phi implementations.

It was found that the uncertainties remaining, including in the data decomposition across nodes, algorithm details and future architecture capabilities, made it infeasible to calculate a single value for the compute efficiency of the SDP at the current time, without having to make too many assumptions for the value to be useful. Additionally, looking at computational efficiency alone could lead to a suboptimal choice of hardware due to the dominant algorithms in a processing pipeline having bottlenecks in the memory hierarchy for example.

We recommend that vertical prototyping work continues to be done with a focus on understanding the limiting factor of algorithm implementations on high performance architectures. This will aid in understanding which hardware resources are most crucial to SDP performance when evaluating future architectures.

Open questions remain concerning the proportion of time that will be spent in each algorithm within SDP pipelines and the proportion of time that each pipeline will run for, as well as the amount of data that each node will have to process and the possibility of data communication across nodes changing the limiting factor of the implementations studied. Possible future work includes studying a wider range of SDP algorithms, investigating single node scaling with data set size, investigating the tradeoff in using one or more accelerators given the additional data copy costs required and creating horizontal prototypes which include communication between nodes at scale.