High performance gridding

ngVLA Computing Memo #5

Martin Pokorny

31 August 2021

We describe the design of a high performance gridding and degridding implementation for imaging codes capable of running on CPU and GPU architectures. An algorithm for the computational kernel of the convolutional resampling and summation of values on a fine regular grid onto a coarse regular grid, also known in aperture synthesis imaging as *gridding*, is described. An algorithm for the inverse operation, *degridding*, which is also required for imaging codes, is described as well. Both algorithms have been designed for a high degree of parallelism, as well as efficient access to arrays required by the algorithms, on both CPU and GPU architectures. The optimization of algorithm design parameters for different architectures is described. The performance of an implementation of the computational kernels in a new software library on various processors is presented, and is also compared with the performance of equivalent kernels derived from current CASA code. Scaling of algorithm performance with application or data-dependent parameters is measured.

Introduction

For image reconstruction (Rao et al. 2009), in every step of the iterative algorithm's outer loop (or*major cycle*), the raw visibility data in the UV plane are resampled onto a regular grid. Once the data have been resampled onto the grid, several iterations of the algorithm's inner loop (or *minor cycle*), which operates only on the resampled data, are executed. After a minor cycle completes, the gridded visibility data are then *degridded*, and another iteration of the major cycle may begin. While the shares of the total work of imaging that are done by the major cycle and the minor cycle may differ significantly for various imaging tasks, in many cases, the sheer difference in quantity of data managed by the major and minor cycles determines that the bulk of the work is done in the major cycle, and that work is itself dominated by the work of gridding. As an example, for typical VLA or ALMA imaging, the number of raw visibilities is roughly on the order 10^{10}, while the size of the grid

(*i.e*, image) summed over all channels and polarizations is on the order 10^8.[1] The work done by the major cycle in gridding will often dominate the work done by the minor cycle in these cases. For ngVLA, data sizes and image sizes both will be significantly larger on average, but it is expected that the computational work of gridding will continue to be the dominant share of computation required for imaging (Bhatnagar, Hiriart, and Pokorny 2021).

In this memo, we describe the design and implementation of a code library for gridding and degridding that is capable of running efficiently on both CPU and GPU processors, and which supports all functionality needed for use by a full imaging code (*e.g*, CASA `tclean` ("Synthesis Imaging Using CASA tclean" n.d.)). The library provides support for full Stokes imaging, sparse Mueller matrices, and access to residual and predicted visibility products. The gridding code has been implemented in a library accessible to client codes executing on CPU hosts, with a high-level interface that manages (potentially) asynchronous communication and computation with accelerator processors (GPUs).

Performance of the library using several different processor types over a wide range of imaging parameter space was measured, and also compared with the equivalent code that currently exists in CASA. Finally, we summarize the current state of the effort to integrate the GPU implementation of the gridding code with CASA imaging code, and provide a preliminary assessment of the potential impact of the current implementation on the estimated cost of the ngVLA imaging pipeline computing infrastructure.

Gridding kernel

In describing the algorithm for the gridding kernel below, we generalize the data types and context somewhat, to bring focus to the algorithm and not its application. The input to the gridding kernel comprises a *measured data set* of some quantity measured at points on a two-dimensional *fine grid*, and a *convolution function kernel* (on $\mathbb{R}^2$) with bounded support. The output comprises a *resampled data set* of the same quantity defined on points of a *coarse grid*, where the linear size of the coarse grid divides the linear size of the fine grid by an integer *oversampling* factor in each dimension[2].

To be precise, we define the following. Let the size of the coarse grid G in either dimension be D_G, and let Q be the oversampling factor. We then define the fine grid G^Q to be the grid with a size in each dimension of QD_G.

Next, let N_V be the number of points in the measured data set. Each element of the measured data set comprises a scalar v in some field $\mathcal{F}$, and a point $\mathbf{x}$ on the fine grid. The elements in the resampled data set S will comprise a single value in $\mathcal{F}$ for every point on the coarse grid G.

Finally, let D_C be the maximum size over all dimensions of the convolution function kernel support on the fine grid. Without loss of generality, we assume Q divides D_C[3]. Thus we define the convolution kernel on the fine grid by its values $C(x, y)$.

[1] Values for number of visibilities and pixels in images may vary widely; examples are representative of only some common cases.

[2] In general, both the coarse and fine grids will have multiple planes (*i.e*, channels, polarizations, *etc*). Although in this memo we only consider the case of a single plane for both measured and resampled data, the generalization of the algorithm to multiple planes is straightforward.

[3] Otherwise, extend the convolution kernel with zero values to a domain of size $\lceil D_C/Q \rceil Q$.

With these definitions, we can define the gridding calculation as

$$S(A,B) = \sum_{(v,(x,y))\in V} vC(QA - x, QB - y) \tag{1}$$

Using the coordinates of points in the measured data set, we partition the data set according to the value of $q \equiv \mathbf{x} \bmod Q \equiv (x \bmod Q, y \bmod Q)$. We define the *minor grid* G_Q as those points in $\mathbf{Z}^2$ with coordinates in $[0,Q)^2$, and then the subset V_q of V as $V_q \equiv \{(v,\mathbf{x}) \in V : \mathbf{x} \bmod Q = q\}$. With this partitioning, equation 1 becomes

$$S(A,B) = \sum_{q\in G_Q} \sum_{(v,(x,y))\in V_q} vC(QA - Q\lfloor x/Q \rfloor - q_0, QB - Q\lfloor y/Q \rfloor - q_1) \tag{2}$$

Finally, the above equation suggests the definition of a family of convolution kernels on the coarse grid: for $(q_0,q_1) \in G_Q$, let $\hat{C}^Q_{(q_0,q_1)}(X,Y) \equiv C(QX - q_0, QY - q_1)$. Which leads to the final formulation of the gridding algorithm:

$$S(A,B) = \sum_{q\in G_Q} \sum_{(v,(x,y))\in V_q} v\hat{C}^Q_q(A - \lfloor x/Q \rfloor, B - \lfloor y/Q \rfloor) \tag{3}$$

Algorithm design

To convert equation 3 into executable code, it is important to realize that all the values in the family of convolution kernels on all grid points in G can be indexed by a four dimensional array. The arguments to the functions $\hat{C}^Q_q$ in equation 3 all lie on points on the coarse grid, and all function values must be identically zero on all points outside of the support of the convolution kernel on the coarse grid, or $[0, D_C/Q)^2$. Using an ordered index corresponding to the values coarse-grid-X, minor-grid-x, coarse-grid-Y, minor-grid-y, the values of the entire family of convolution kernels can be stored in a four-dimensional array of size $K * Q * K * Q$, where $K \equiv D_C/Q$.

With this transformation, the algorithm to compute equation 3 may be written in pseudo-code as

```
int G; // grid size
int N; // measured data set size
int K; // convolution function kernel support size (coarse)
int Q; // oversampling factor
T v[N]; // measured data values
int x[N], y[N]; // measured data locations (fine)
T g[G, G]; // resampled data values
T c[K, Q, K, Q]; // convolution function kernel values (fine)
for (int i = 0; i < N; ++i)
  for (int X = 0; X < K; ++X)
    for (int Y = 0; Y < K; ++Y)
      g[X + x[i] / Q, Y + y[i] / Q] +=
        v[i] * c[X, x[i] % Q, Y, y[i] % Q];
```

Memory access analysis

Referring to the previous pseudo-code algorithm, it is seen that every iteration of the innermost loop (line 11) requires one multiplication (of a measured data value and convolution kernel value), and one addition (to a resampled value). The total number of floating point operations is thus two or eight, depending on whether $\mathcal{F}$ is the field of real numbers[4] or complex numbers[5], respectively.

The number of memory accesses required for every iteration of the innermost loop of the pseudo-code algorithm amounts to three reads of scalar data values, one write of a scalar data value, and two reads of position index values. To make a rough estimate of whether this algorithm will be memory- or compute-bandwidth bound, we will ignore the access to index values. For simplicity, we will also assume that each measured data value will be read only once, in the outermost loop (line 9), and not in every iteration of the innermost loop[6]. Given these assumptions, the ratio of floating point operations to accessed floating point values over all loop iterations is $2N_V K^2/[N_V(3K^2+1)]$ or $8N_V K^2/[N_V(6K^2+2)]$, depending on whether $\mathcal{F}$ is the field of real or complex numbers, respectively. Assuming that all floating point values are single precision (four bytes), we find that, for large values of N_K, the arithmetic intensity of the algorithm is approximately 0.17 or 0.33 (FLOPs per byte). These values of arithmetic intensity are far below the typical machine balance[7], suggesting that the algorithm will be memory-bandwidth limited. Note also that for many applications, the resampled data will have double precision, which will only drive the arithmetic intensity lower.

The preceding model is incomplete — it does not include degridding, calculations to convert visibility UV coordinates to grid coordinates, phase screen calculations, or integer index calculations, among other omissions[8] — but the arithmetic intensity is low enough that gridding is unlikely to be shown to be compute limited in a more complete model. Efforts to optimize the gridding kernel should, however, be based on measured, not model, values.

Parallel algorithm description

In a typical application of the gridding kernel, the size of the measured data set is significantly larger than the number of points in the coarse grid. On the basis of that consideration, parallelization of the outermost loop of the pseudo-code algorithm is highly desirable. On the other hand, the resampled data values on the coarse grid are accessed many more times than the measured data (by a factor K^2), and the algorithm is memory bandwidth limited, so iteration over the inner loops of the pseudo-code algorithm should also be parallelized, but in such a manner as to optimize efficient access to the resampled values on the coarse grid.

[4] One multiplication and one addition.

[5] One complex multiplication, requiring four floating point multiplications and two floating point additions, and one complex addition, requiring two floating point additions.

[6] The calculation can be repeated for the other cases, but which case occurs in practice will depend on the details of how the code is written, and what optimizations a compiler applies.

[7] Published specifications of the NVIDIA V100 PCIe GPU assert double-precision performance of 7 TFLOPS, and memory bandwidth of 900 GB/sec, which yields an estimate of the machine balance of $(7 \times 10^{12})/(900 \times 10^9/8)/8$, or about 7.8 double precision FLOPS per byte.

[8] Note that the value of the effective machine balance may change with the memory holding accessed values (register, L1 cache, L2 cache, main memory, *etc*), but the arithmetic intensity, being a characteristic of the gridding algorithm, remains independent of those factors.

In many cases, however, the measured data values are not sorted in any regular order on the fine grid, and this limits the efficiency that might be otherwise achieved by exploiting known access patterns on the coarse grid. In addition, were there a sort order that could be exploited, parallelization of the outermost loop would likely destroy that order, resulting in the loss of the potential gain. Any sorting of the measured data that could provide a boost to coarse grid access efficiency in the presence of outer loop parallelization would thus likely need to be aware of the degree of parallelization used by the implementation in the outer loop, potentially complicating the sorting algorithm, or imposing rigid constraints on the parallelization in the outermost loop.

Given that parallelization over the measured data set in the outer loop of the pseudo-code algorithm will create updates to resampled data values in an irregular, unpredictable order on the coarse grid, synchronization among units of execution for updates to the resampled data values is a necessity. Therefore, given the parallelization of line 9 in the pseudo-code algorithm, the update to the resampled data value in line 12 must be protected by some form of synchronization.

Optimizing efficient access to the resampled values on the coarse grid within the inner two loops requires exploiting the relation between array layout order in memory and memory access characteristics of the target computer architectures, whether CPU or GPU[9]. The flexibility of the pseudo-code algorithm with regard to array layout order is critical to achieving this goal. In particular, changing the domain of the convolution kernel from the "natural" two dimensions to four is an important characteristic of the algorithm in this regard. Note that given a logical definition of the array dimensions, the permutation of those dimensions in regard to the layout does not affect the logical order of those dimensions, which allows for the algorithm implementation logic to remain independent of layout.

Multi-dimensional array layouts are particularly important to determining the efficiency of access to array elements in memory within loop constructs. Depending on the computer architecture, the strategies needed to optimize memory access are significantly different. Controlling array layouts is an effective method of optimizing memory access for both CPU and GPU architectures. There is a single general pattern in loop constructs to be followed to achieve efficient memory accesses for each of typical CPU and GPU architectures. In the case of CPUs, a thread's access is efficient if logically sequential accesses to values are also sequential in the memory address space, allowing for optimal cache memory usage. In the case of GPUs, a thread block's access is efficient if consecutive threads in the block access consecutive values in memory at once, allowing for optimally coalesced memory access. In general, these access patterns can be achieved either through implementation logic or through array layouts.

Thus the following design elements are key to the high performance version of the pseudo-code algorithm: parallelization of the loop over measured data (line 9), synchronization among parallel units of execution in the access to resampled data (line 12), parallelization over the support of one or both dimensions of the convolution kernels (lines 10 and 11), and array layout optimization of the arrays for resampled data and convolution kernels (lines 7 and 8, including four dimensional index for the convolution kernels). High performance gridding can be expected to be limited by the following

[9]By "layout" we refer to the mapping from a logical array index to a linear memory address offset. In normal usage, for the C and C++ programming languages, multi-dimensional arrays are stored in "row-major" layout, while for the Fortran programming language, arrays are stored in "column-major" format. The notion of row-major and column-major orders for multi-dimensional arrays is generalized by layouts. For dense, multi-dimensional arrays, the options for regular, dense layouts are equivalent to permutations of the dimensions of the array.

factors: latency for the first access to the resampled values on the coarse grid for each measured data value, cost of synchronization in the access to resampled data, and memory access efficiency of the loops over the convolution kernel support. Because of the unordered access to points on the coarse grid that is a result of the unordered measured data (plus parallelization), every iteration of the outermost loop will likely have a high latency cost because the resampled data for the points on the coarse grid that are to be updated will not be in "near" memory at the beginning of that iteration. The latency cost of each measured data point will be only weakly dependent on other factors such as grid size, convolution support size, oversampling factor, or the set of all points in the measured data, and we may consider it to be a fixed value (on average) per measured data point. The cost of synchronization to update the resampled data will depend upon the synchronization method, but, on average, may be expected to be a function of the level of contention. The level of contention may depend upon many factors, such as the method of synchronization, the degree of parallelization, the size of the measured data set, the coarse grid size, and the convolution kernel support size. The memory access efficiency to the grid by the inner loops will depend on the array layouts of both the resampled data and the convolution kernel. With the right layouts, the algorithm's performance should approach the limit set by the maximum memory bandwidth of the computer architecture as the convolution kernel support size increases (disregarding, for the moment, the other limiting factors discussed previously). Note that for GPU architectures, typical convolution kernel support sizes are far too small to utilize the maximum memory bandwidth in the inner loops, and therefore, all else being equal, the convolution kernel support size will be the determining factor for realized memory throughput by the algorithm.

Additional features

Full Stokes imaging with Mueller matrices

For high quality, full polarization imaging, it is necessary to model the complete Mueller matrix. Whereas all imaging may be described with Mueller matrix components, in many applications several components of the matrix are identically zero, potentially reducing both memory (along with device I/O) and compute loads. To realize the potential gains of identically zero Mueller matrix elements, an efficient, sparse representation of the matrix may be beneficial. The Mueller matrix affects the selection of convolution kernels used in both gridding and degridding.

Full Stokes imaging of course assumes that visibilities being gridded exist in the form of various polarization products. During gridding, for every point on the fine grid, the Mueller matrix linearly combines the various polarization products of the visibilities measured at that point to compute the update to a point on the coarse grid. Consideration of computational efficiency thus suggests that the incremental update (*i.e*, the linear combination of visibilities) be computed first, to be then followed by a single update to a point on the coarse grid. These considerations lead to the conclusion that it may be most efficient to organize visibilities into groups containing all polarization products at every point on the fine grid.

Sum of weights

In order to compute the correct normalization of the values resampled onto the coarse grid, it is necessary to compute the sum of the convolution kernel values used for gridding each visibility to every Stokes plane in the coarse (image) grid, multiplied by a weight associated with the visibility, summed over all visibilities. The sums of convolution kernel values may be implemented using reductions to short vectors (of length equal to the number polarization products in the visibility data) in the innermost loops over the domain of the convolution function kernel in pseudo-code algorithm in order to restrict memory accesses to local memory, followed by a single update to a global array of weights, one per visibility. As the updates to the sum-of-weights global array occur in parallel over the set of visibilities, these updates must also be synchronized. Finally, the computation of the sum of weights should be optional, as the sum does not change from one major cycle to the next, and thus the cost of its computation may be avoided by the majority of major cycles.

Degridding

A single major cycle of the iterative imaging algorithm begins with the gridding phase, in which residual, measured visibilities are resampled onto a regular grid, and the cycle completes with the degridding phase, in which model visibilities are constructed at the coordinates of the measured visibilities from the residual image. The degridding algorithm is closely related to the gridding algorithm, and requires very similar inputs and operations as the gridding algorithm.

An important difference of degridding from gridding is that in degridding the visibilities are updated, and the model values are only read, not written. When the outer loop over visibilities is parallelized (as in gridding), the access pattern of visibility values in degridding does not require synchronized access at this level. Additionally, because model values are read-only during degridding, no synchronization is required to access those values.

The grouping of visibilities across polarization products can also be exploited by the degridding algorithm. In degridding, similar to gridding, there exist two, nested loops over the domain of the convolution function kernel (on the coarse grid). These inner loops may be implemented using reductions to short vectors (of length equal to the number of polarization products in the visibility data) to accumulate the contributions of the product of model and convolution function kernel over all points in the domain of the convolution function kernel to every visibility polarization product.

Residual and predicted visibilities

At the conclusion of imaging, it is often desirable to produce the final, residual visibilities. Similarly, it can be useful to produce predicted visibilities, that is the visibilities at the coordinates of the measured visibilities but based entirely on the model visibilities.

Convolution kernel phase screen

For the correction of some pointing offsets (for example, a difference in antenna pointing center and correlator phase center) a complex phase gradient can be applied to the convolution kernel during gridding and degridding. On-the-fly calculation of the correction factors given user-supplied phase gradient values, and their application during gridding and degridding is an efficient strategy given the memory-bandwidth limited nature of the gridding and degridding algorithms.

Efficient I/O to accelerators

Effective use of GPU accelerators often requires the ability to overlap data movement between CPU and GPU memory with computation on the GPU. Further contributions to efficiency are possible by overlapping work on the CPU host with the asynchronous progress of GPU tasks. The implementation of a software *library* that implements high performance gridding should support these optimizations to promote efficiency in the use of high performance gridding by client CPU codes.

Implementation

HPG

The implementation of high performance gridding is available in the HPG project. The main repository of the HPG source code is located at https://gitlab.nrao.edu/mpokorny/hpg. The primary build artifacts are a library, `libhpg`, and C++ header files for use by client codes. A secondary build artifact is `minigridder`, a mini-app designed for HPG code debugging and profiling[10].

Full Stokes imaging with Mueller matrices

For full Stokes imaging, in both gridding and degridding, the visibilities for multiple polarization products are combined according to the values of a Mueller matrix. For this reason, it is computationally efficient to represent groups of visibilities that share common metadata and differ only in their polarization product identities. An additional benefit of such a representation is the reduction of (repeated) metadata that would otherwise be associated with each visibility separately. However, as the visibilities must often be copied into device memory prior to gridding or degridding, to reduce

[10] The `minigridder` mini-app, which was used for many of the performance tests reported in this memo, has been subject to significant changes since those original tests were conducted. For example, a previous version of `minigridder` included the CASA gridding algorithm as an option, in which the relevant CASA gridding source code (for A/W-projection) was copied to `minigridder` and modified with the bare minimum of changes needed to function in `minigridder` (in the attempt to maintain fidelity with the original CASA code.) However, later development of additional HPG features required changes to `minigridder` that were difficult to integrate with the CASA gridding algorithm option, and thus that option was subsequently removed. Similarly, previous `minigridder` versions provided options to control array layouts in order to measure their effect on performance, but these options have also been removed. Metrics reported by `minigridder`, such as utilized memory bandwidth, have also been removed as the gridding/degridding algorithms developed further.

device I/O as well as device memory usage, it is important to minimize the size of the groups of visibilities. For the support of efficient representations of visibility groups, the size of the visibility groups is determined at compile time by HPG client code.[11]

The elements of a Mueller matrix used for imaging may be considered as convolution operators. It is not only sufficient to implement the elements of a Mueller matrix as indexes in an ordered set of convolution kernels, but it is also efficient to do so when a single convolution kernel may be indexed by multiple elements of the Mueller matrices used for gridding or degridding. An additional measure of efficiency is gained by representing identically zero convolution operators by a special, sentinel index value for those elements in the Mueller matrix, thus potentially avoiding some floating point operations, as well as reducing memory requirements. In HPG we implement this as follows: given a set of convolution kernels $C = \{C_0, C_1, ..., C_{N-1}\}$, where none of the C_i is identically zero, then the Mueller matrix elements are integers $M_{i,j} \in \{-1, 0, 1, ...N-1\}$ where an element with value -1 indicates an identically zero convolution kernel.

In current CASA and HPG degridding and gridding code, it is assumed that the Mueller matrix is unitary. In HPG, while the representation of Mueller matrices described above is fully generic, the assumption of a unitary Mueller matrix is manifest in the implementation of the degridding and gridding kernels. Although it would be simple to support the general case in HPG without a change to its API, this would generally impose a runtime cost in memory usage[12], and therefore we have opted for simplicity over generality, and consistency with CASA at this time.

Degridding

During degridding, the coordinates of each visibility are used to evaluate the model at those coordinates. Degridding a visibility requires write access to local variables, and read-only access to the visibility, model values and convolution kernel values.

The degridding kernel is distinct from the gridding kernel. While this design requires an additional traversal of visibility data, experiments indicate that overall performance improves on many-core processors and GPUs with this design.

Gridding

Gridding a visibility requires write access to global variables (the grid values), and read-only access to the visibility and convolution kernel values. Write access to global variables in a parallel program requires synchronization to values that are updated to prevent interference by other threads of execution; in HPG, we implement this with atomic updates, which is an efficient technique for GPU processors.

[11] The HPG type is `VisData<N>`, where the template parameter `N` is an unsigned integer in the set $\{1, 2, 3, 4\}$. Additionally, the type `VisDataVector` represents the type `std::vector<VisData<N>>` with the parameter `N` being erased, both for use by client code and to provide isolation of the HPG API from its implementation.

[12] The additional memory requirement could be avoided with the addition of new methods in the HPG API, however.

Residual and predicted visibilities

For some use cases the HPG client may need to access the residual visibilities, or the difference of measured and model visibilities at the coordinates of the measured visibilities. Typically, the residual visibilities are only accessed at the end of imaging. Another use case requires access to the predicted visibilities, or the model visibilities at the coordinates of the measured visibilities. In order to support asynchronous progress of computation on capable devices (*e.g*, GPUs), both of these values are returned as "futures." The HPG future types are somewhat different from the usual `std::future` types in that completion can only occur during a call to an HPG `Gridder` or `GridderState` method or function, despite computations continuing asynchronously on capable devices. Because of this design, a blocking call to something like `std::future::wait()` would be prone to deadlock (assuming single-threaded client code). The typical usage in HPG, rather, is to call `hpg::future::get()` to poll the state of the `hpg::future` value. Futures are always assured to have completed after a call to the `fence()` method of `Gridder` or `GridderState`.

The choice of predicted or residual visibilities depends on whether the operation is only degridding, or both degridding and gridding. In both cases, the evaluation of predicted or residual visibilities takes place after degridding (if selected), and before gridding (if selected).

Convolution kernel phase screen

Convolution phase gradients are provided by client code as input for gridding and degridding as part of the metadata of each group of visibilities. Complex phase values derived from phase gradients are computed and applied to the convolution kernels on-the-fly, as needed, for degridding and gridding. Following testing conducted during development, it was determined that the application of the phase screen has no negative performance impact, and therefore a phase screen is always applied by HPG. If no effective phase screen application is to be used, users can simply provide zero-valued phase gradients where they are required.

Array layouts

Performance of gridding and degridding is sensitive primarily to the layout in memory of the arrays used to represent the convolution kernels, the grid values, and the model values. HPG ensures that there is a distinction in the client-side representation of these arrays from the HPG implementation side in order that array layouts used by HPG can be optimized to every supported processor type independent of client code. Nevertheless, for improved efficiency in data transfer to device memory during gridding, HPG has added support for clients to reorganize convolution kernel values into optimal layouts for devices in host memory. This capability allows storing those arrays in persistent memory well ahead of time, outside the imaging process itself, and then loading those arrays to device memory without the cost of reorganization during imaging.

Asynchronous data transfers

HPG supports asynchronous data transfers between host and device memory on all devices that are capable. Quantities that allow asynchronous transfers are convolution kernels, model values, data visibilities and residual or predicted visibilities. HPG implements a task queue of limited size, determined by the client code, for asynchronous device tasks. Although calls to most HPG functions may be non-blocking in many cases, those functions may block when the task queue is full. The primary purpose of limiting the size of the queue is to control the use of device memory for degridding and gridding.

Testing

Phase 1

Phase 1 testing was conducted using an early version of HPG[13] and `minigridder`, with the primary goal of demonstrating the performance effects of array layouts, and for a comparison with CASA gridding performance.

All phase 1 performance tests were conducted on the machine "yaghan.aoc.nrao.edu." This machine has 256 GiB of RAM, a dual socket 16-core Intel Xeon Gold 5222 CPU @ 3.80 GHz, and an NVIDIA Tesla V100 GPU with 32 GiB RAM. Compiler options were used to create the `minigridder` executable for the "cascadelake" x86_64 micro-architecture, and the "7.0" NVIDIA compute capability. In order to match the types used for gridding in CASA, all reported tests have used single-precision complex values for the measured data and the convolution kernel, and double-precision complex values for the resampled data.

Layout tuning

The selection of optimal layouts for the multi-dimensional arrays used in high performance gridding ought to be relatively simple. But before describing the method used to optimize array layouts, it is useful to introduce the encoded descriptions of layouts used by `minigridder`, both as a shorthand in this discussion of layouts, and in following descriptions of trial configurations and their results.

Layouts used by `minigridder` may be encoded by a short string of characters: `X` and `Y` for the coarse grid dimensions, and `x` and `y` for the minor grid dimensions. A layout is specified by concatenating the characters from the set that describes the logical array dimensions, but in order from smallest stride (in memory) to largest stride. For example, the coarse grid G, with logical array indexes X and Y, is encoded by the string "YX" for row-major layout order, and "XY" for column-major layout order.

The layouts for the convolution kernel array are somewhat more diverse; with four dimensions, there are 4! possible layouts. Also of note is that each of the two "usual" two dimensional array layouts is equivalent to one of the four dimensional layouts. For example, the four dimensional layout "xXyY"

[13] The original demonstration version of the gridding kernel implementation went by the name "HTG," which one may still find in some of the following plots.

yields the same layout of values in memory as the original two dimensional array in column-major order. This feature allows for a single convolution kernel array instance needed for CASA as a two-dimensional array to be created and used by `minigridder` as a four-dimensional array.

The principles of cached access to CPU memory, and coalesced access to GPU memory, suggest the following heuristic (table 1) for optimal array layouts for the high performance gridding algorithm

Table 1: Optimal array layout heuristic

device	grid	ck
CPU	YX	YX..
GPU	XY	XY..

where the `.` characters are used to indicate an indeterminate choice (of `x` and `y`, in the case of the convolution kernel array). Because the dimensions `x` and `y` do not correspond to any loop indexes, the effect on performance of their position in the layout is less significant than the positions of `X` and `Y`, as well as being more difficult to predict.

The optimal array layouts may not necessarily agree with those determined by the above heuristic. The actual effect of the array layouts on performance may depend upon specific details of a processor, such as cache size and instruction set, the compiler used to build the executable, or the problem shape. While one could identify the optimal array layouts for each of these cases, for the purposes of this memo we wish simply to evaluate the quality of the heuristic, and to provide an illustration of the effect of array layouts.

The following table 2 shows the effect on performance of various array layouts for one example problem shape. While only some of the forty-eight possible layout combinations are shown, those include the best performing, the worst performing, the expected best performing by the heuristic, and the two-dimensional equivalent layouts of the convolution kernel.

Table 2: Estimated memory throughput (GB/s) for selected table layouts. Other parameters: grid size 10000, ck size 31, oversampling 40.

grid layout	ck layout	bw	rel perf
XY	XYxy	592	1.000
XY	XYyx	592	1.000
XY	YXxy	386	0.652
XY	xXyY	335	0.566
XY	yYxX	304	0.514
YX	XxYy	51	0.086
YX	xXyY	48	0.081
YX	yYxX	47	0.079
YX	yxXY	47	0.079

Note that the optimal layouts on the GPU agree with the layout heuristics, and also note that the

performance degrades significantly with many other layouts. In particular, note that for each grid layout, the convolution kernel layouts that are equivalent to two-dimensional layouts are among the worst performing.

Effect of synchronization

Here we measure the effect of synchronization on algorithm performance on a GPU. An atomic update operation will likely have some fixed performance cost plus a cost that increases with increasing contention[14]. Increasing the size of the measured data set (*i.e*, number of visibilities) while the grid size remains fixed will result in increasing contention to updates on the grid. Results are shown in table 3.

Table 3: Estimated memory throughput (GB/s), with and without synchronized grid value updates. Other parameters: grid size 1000, ck size 11, oversampling 20, grid layout XY, ck layout XYxy.

num vis	sync bw	no sync bw	sync:no sync ratio
1e+04	576	638	0.90
1e+05	644	765	0.84
1e+06	651	779	0.84
1e+07	656	781	0.84
1e+08	667	796	0.84

Performance scaling

In this section we present performance scaling results with various problem shape parameters.

Grid size To measure the performance scaling effect of grid size, we control for the level of update contention by also scaling the number of visibilities with the number of points in the grid, as shown in table 4. While not presented here, similar results are obtained when the number of visibilities remains fixed.

Table 4: Grid sizes and number of visibilities for grid size scaling trials (figure 1) .

gsize	nvis
500	1.0e+04
1000	4.0e+04
2000	1.6e+05
4000	6.4e+05
8000	2.56e+06

[14]There are conditions under which atomic updates on GPUs are known to be faster than non-atomic updates, so the fixed cost is by no means assured.

gsize	nvis
16000	1.024e+07
32000	4.096e+07

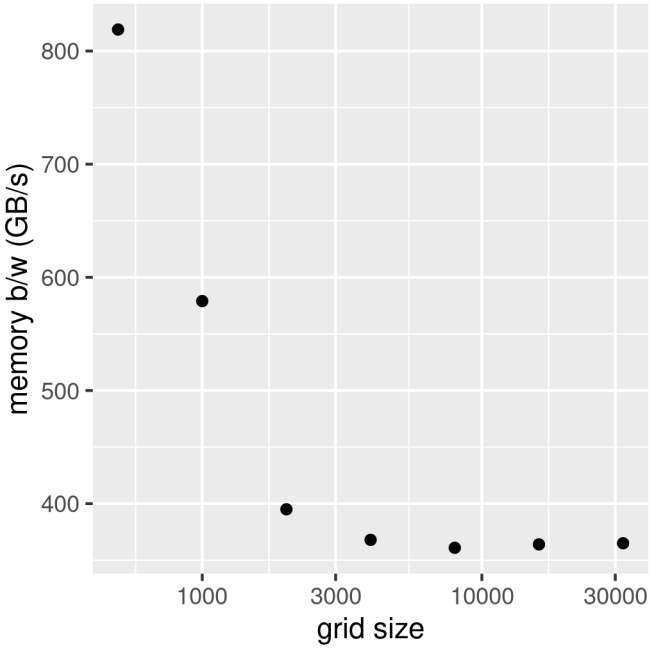

Figure 1: Performance scaling with grid size. Other parameters: ck size 9, oversampling 20, grid layout XY, ck layout XYxy.

The increase in performance for very small grid sizes is likely due to increased cache efficiency. A grid of size 500 is small enough to fit entirely into L2 cache on the Tesla V100 GPU.

Number of visibilities The performance scaling effect of number of visibilities is effectively constant (table 5). Because of overheads due to CUDA kernel launches, performance slightly improves for a larger number of visibilities.

Table 5: Number of visibilities versus estimated memory throughput (GB/s). Other parameters: grid size 10000, ck size 9, oversampling 20, grid layout XY, ck layout XYxy.

num vis	bw
1e+04	357
1e+06	360
1e+07	367
1e+08	367

Convolution kernel support size Here we measure the performance scaling effect of the convolution kernel support size. As has been described above, it is expected that the memory bandwidth will improve as the support size increases, and to approach a limit determined by access latency to the grid and the convolution kernel values. Results are shown in figure 2.

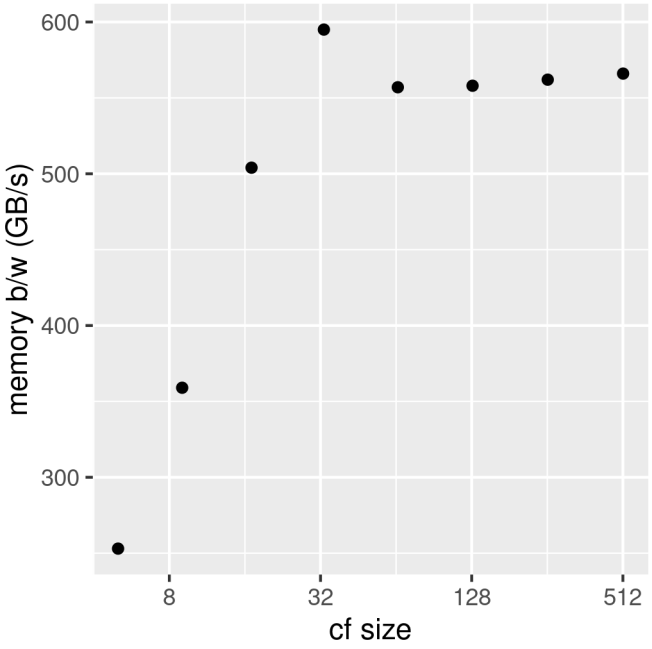

Figure 2: Performance scaling with convolution kernel size. Other parameters: grid size 10000, oversampling 20, number of visibilities 1e6, grid layout XY, ck layout XYxy.

Performance comparisons

In this section we compare the performance of the high performance gridding kernel GPU implementation with the performance of the CASA gridding kernel CPU implementation in `minigridder` for various problem shapes. All results compare the gridding rate of visibilities for high performance gridding on a single GPU *vs* CASA gridding on a single CPU. Extrapolating these results to real applications will require some caution, as there are significant factors that will affect outcomes. For example, although the CASA gridder is single-threaded, in typical usage a single node will run concurrent processes for an imaging job, and the scaling performance of those concurrent processes will have some bearing on the comparison. Also, the GPU tests were conducted with a single kernel running at a time; the performance effect of multiple, concurrent kernels on the GPU has not been determined.

For the first test, we vary the size of the convolution kernel and the oversampling factor. The results (figure 3) show that the high performance gridding kernel processes visibilities at a rate of at least one hundred times that of the CASA gridding kernel, across the entire range of convolution sizes and oversampling factors. This result also shows that the high performance gridder is largely unaffected by the oversampling factor.

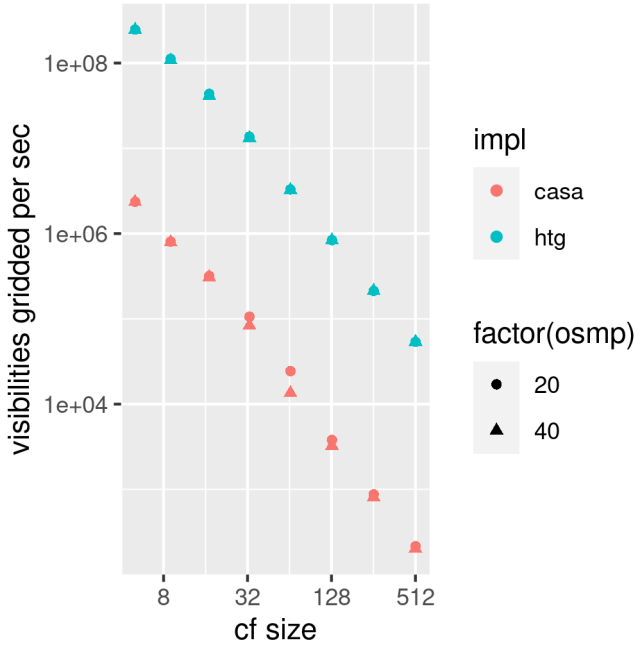

Figure 3: Performance of high performance gridding on GPU vs CASA gridding on CPU as a function of convolution size and oversampling. Other parameters: grid size 10000

Phase 2

Performance tests in phase 2 were completed using a very recent version of HPG on two NVIDIA GPU systems. As implemented, the degridding and gridding algorithms are expected to be most efficient running on GPU systems; measurements for multi-core CPU performance have not yet been completed.

The tests have been designed to measure the performance of the degridding/gridding kernel, specifically not including the I/O to move data between CPU and GPU memories. In a complete imaging application the I/O might, in fact, become the performance limiting factor, but for the development and evaluation of the degridding/gridding kernel, the I/O becomes a source of confusion and is best omitted from the test. On the other hand, the HPG library is designed to support overlapping computation and communication with capable devices. Ongoing work to integrate HPG and CASA will provide a basis to evaluate the usability and effectiveness of the current HPG design for asynchronous I/O, but the results of that work will only appear in a future memo.

Tests were conducted to measure the scaling of performance with various imaging parameters. The most basic performance metric is the rate at which visibilities are processed, whether degridding, gridding or both.

Test parameters:

- Convolution kernel (linear) size
  - $2^i + 1, \forall i \in \{3, 4, ..., 9\}$

- oversampling
  - $10 \times 2^i, \forall i \in \{0, 1, ..., 6\}$
- devices
  - V100
  - A100
- Mueller matrix shape
  - I1 (1x1 diagonal)
  - I2 (2x2 diagonal)
  - I4 (4x4 diagonal)
  - D2 (2x2 dense)
  - D4 (4x4 dense)
- sum-of-weights in gridding
  - on
  - off
- operations
  - gridding
  - degridding
  - degridding and gridding

All scripts used to conduct the performance tests for this memo, and analyze the test data are available at http://gitlab.nrao.edu/mpokorny/ngvla_hpg_memo, under the `analysis` sub-directory. The test data itself can also be found at that web site under the `data` sub-directory. We provide several plots here to summarize the performance data collected.

Test methods

As described below, HPG has been implemented using the Kokkos[15] (Edwards, Trott, and Sunderland 2014) core libraries. For this performance test suite, we used the Kokkos profiling tools interface to acquire timing data for the execution of gridding and degridding kernels. The Kokkos profiling tools interface supports the acquisition of timing data to be enabled at runtime, without recompiling HPG or `minigridder`[16]. The tool we use for gathering the timing metrics is timemory[17] (Madsen et al. 2020).

The regions of code for which timing metrics are collected by the test suite are as close as possible for the Kokkos profiling interface to the gridding and degridding computational kernels alone. It is important to understand that the overhead for most of these metrics has been reduced to only the

[15] Part of the Kokkos C++ Performance Portability Programming EcoSystem. For more information, see https://github.com/kokkos/kokkos.

[16] There is no significant impact to performance when Kokkos profiling is not enabled at runtime. The Kokkos profiling tools interface is a permanent feature of Kokkos in recent versions.

[17] https://github.com/NERSC/timemory

kernel launch time[18]; in particular, no data movement to or from the memory used by the kernels is included in the overhead. What this means for applications is that the timing metrics collected for these tests represent the best possible performance of HPG, and applications may find that other factors (not included in these measurements) are performance limiting.

Results

Test results are shown in the following sections with graphical plots. In many cases, plots contain a set of graphs for various choices of HPG parameters. The names of parameters used in the plots indicate the following:

- operation: HPG operation on visibility and/or model data
  - grid: gridding only
  - degrid: degridding only
  - both: degridding and gridding
- mueller_indexes: shape of Mueller matrix
  - I1: 1x1 diagonal
  - I2: 2x2 diagonal
  - I4: 4x4 diagonal
  - D2: 2x2 dense
  - D4: 4x4 dense
- ck_size: linear size of convolution kernel
- oversampling: convolution kernel oversampling factor

Performance with NVIDIA V100 GPU Tests were performed using the machine "yaghan.aoc.nrao.edu," which has one NVIDIA V100 GPU with 32 GiB Memory. Results are summarized in figure 4.

Performance with NVIDIA A100 GPU Tests were performed using the machine "hal-dgx.ncsa.illinois.edu"[19], which has eight NVIDIA A100 GPUs, each with 40GiB memory. Note that the test suite uses only one GPU. Results are summarized in figure 5.

Relative performance of NVIDIA A100 to NVIDIA V100 Differences in performance of the V100 and A100 GPUs are not readily apparent in the log-log graphs of figures 4 and 5. For easier comparison of differences, figure 6 depicts the ratios of performance metrics.

Note that the HPG source code used for the two models of GPU is strictly identical.

[18] Timing metrics used for this memo reflect what are close to the best possible externally visible measurements. For example, timing for degridding followed by gridding (what is named the "both" operation, subsequently) is recorded as a single measurement, although, internally, HPG will run two kernels in sequence.

[19] This work utilizes resources supported by the National Science Foundation's Major Research Instrumentation program, grant #1725729, as well as the University of Illinois at Urbana-Champaign.

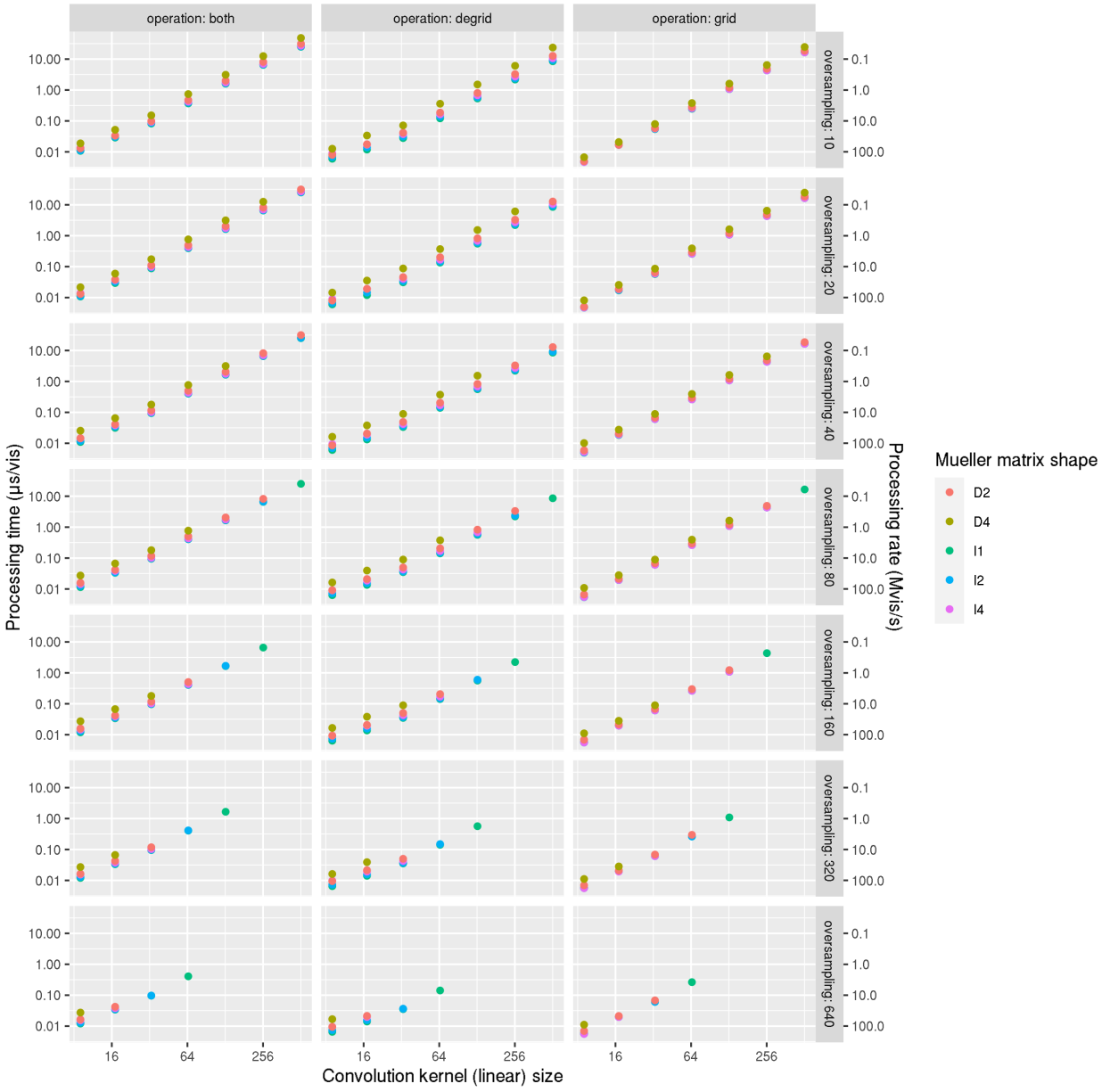

Figure 4: Visibility processing performance for V100.

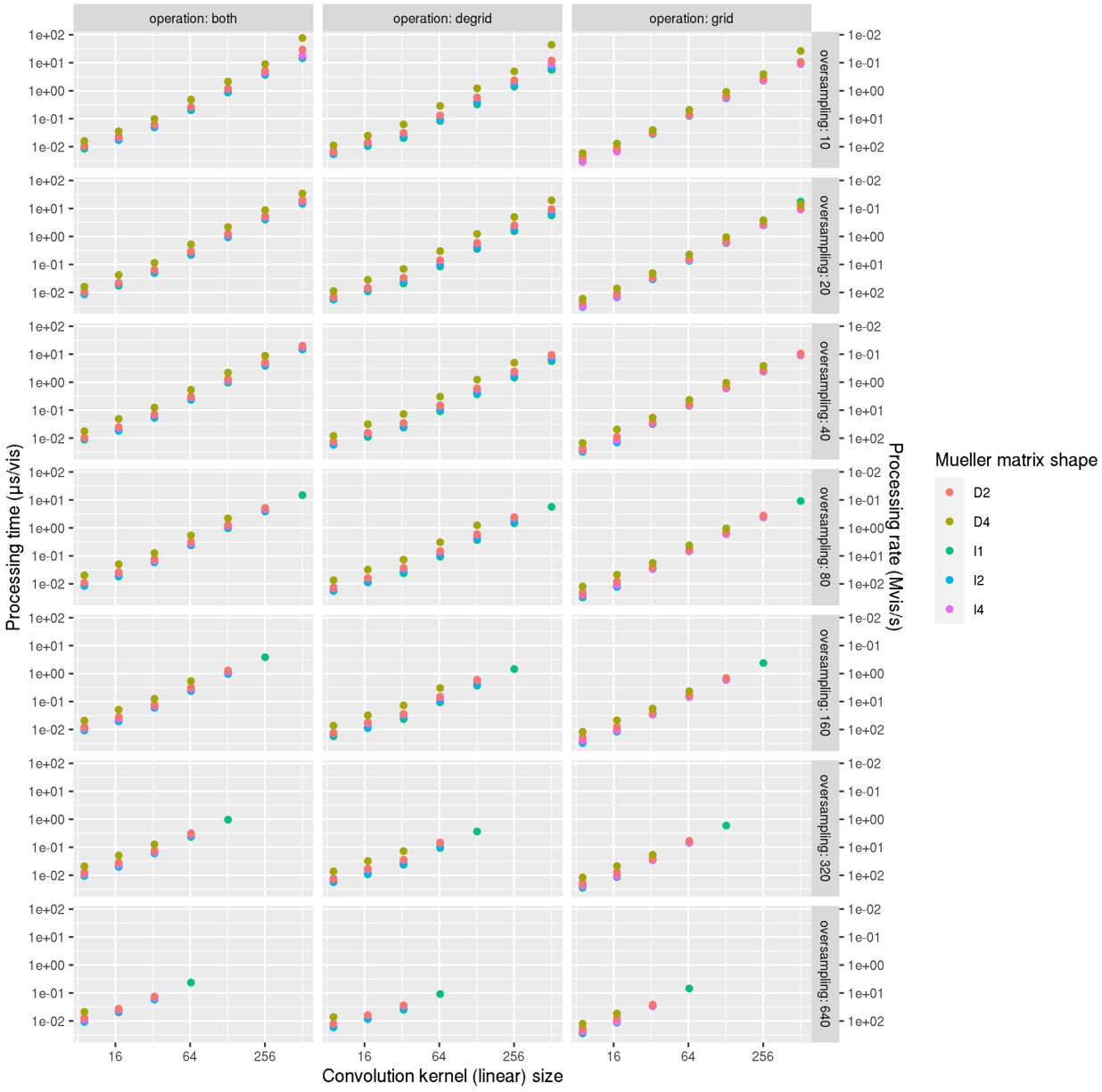

Figure 5: Visibility processing performance for A100

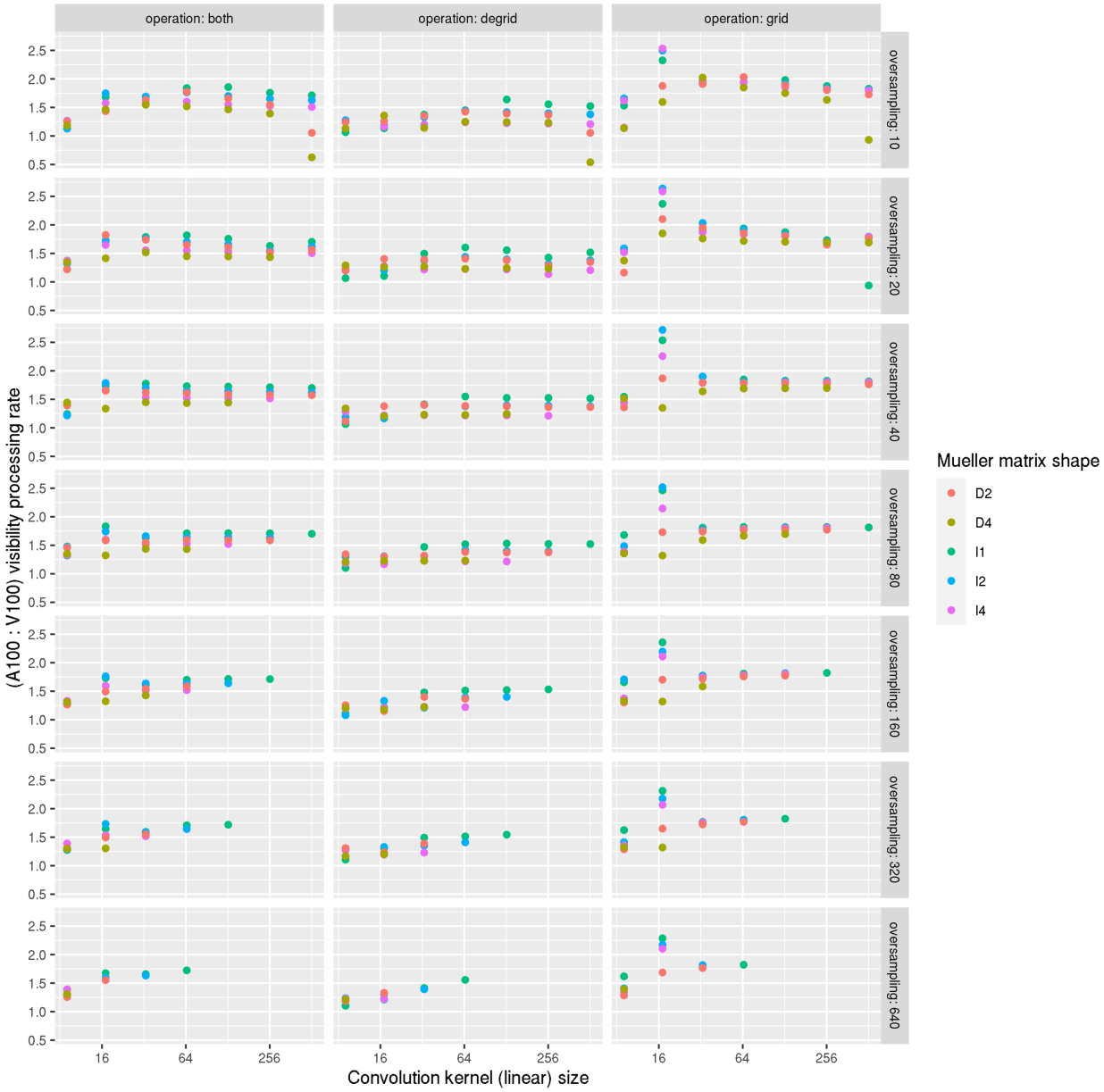

Figure 6: Relative performance A100 vs V100

Performance for choice of operation To highlight the performance differences for the various operations supported by HPG — namely, only gridding, only degridding, or both degridding and gridding — figure 7 depicts line graphs of the data shown in figures 4 and 5, but for a single value of the oversampling factor (20).

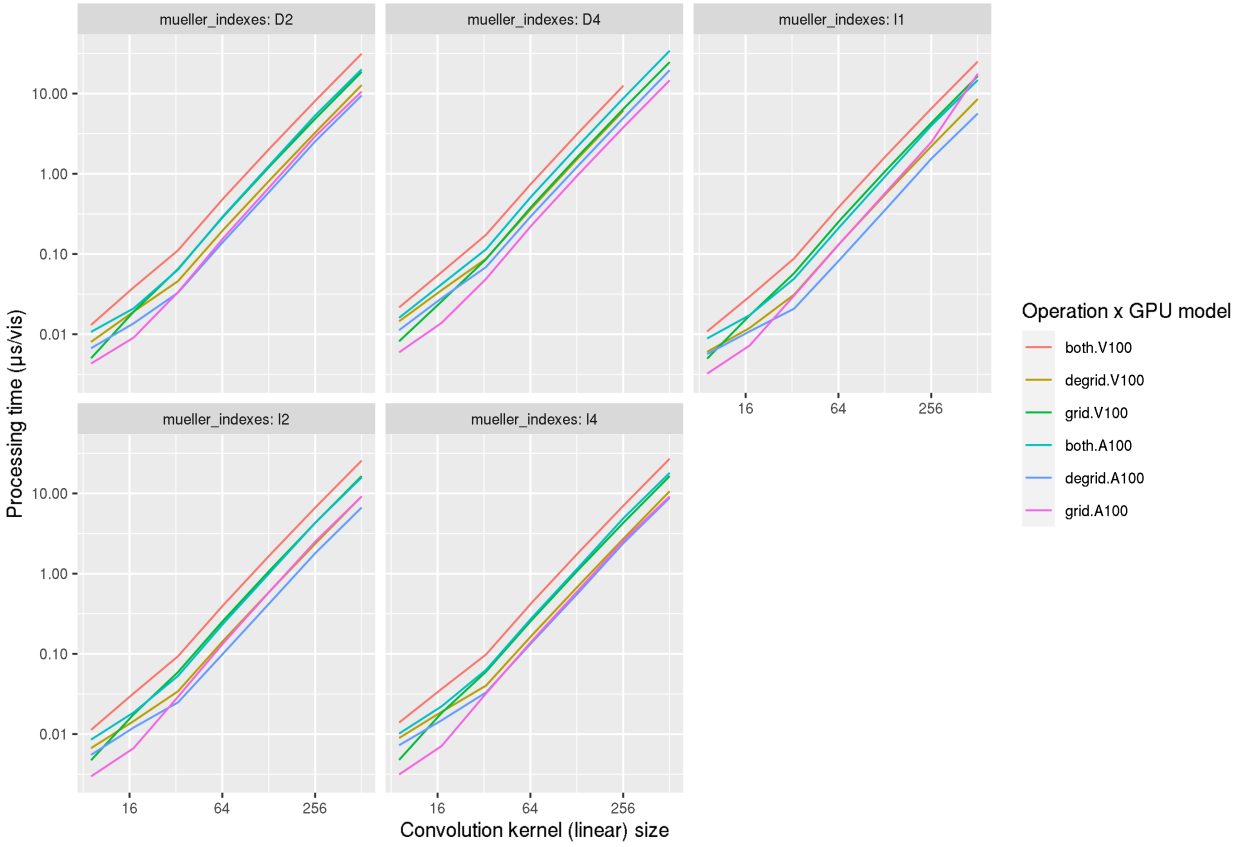

Figure 7: Performance for oversampling factor 20

Performance for sum-of-weights computation

To show the performance effect of the choice of whether or not HPG computes the sum of weights, we plot the ratio of performance metrics in figure 8.

Performance variation with oversampling factor

To show the performance effect of the oversampling factor, we plot the performance at a given value of the oversampling factor relative to the performance at the minimum tested oversampling factor (10) in figure 9.

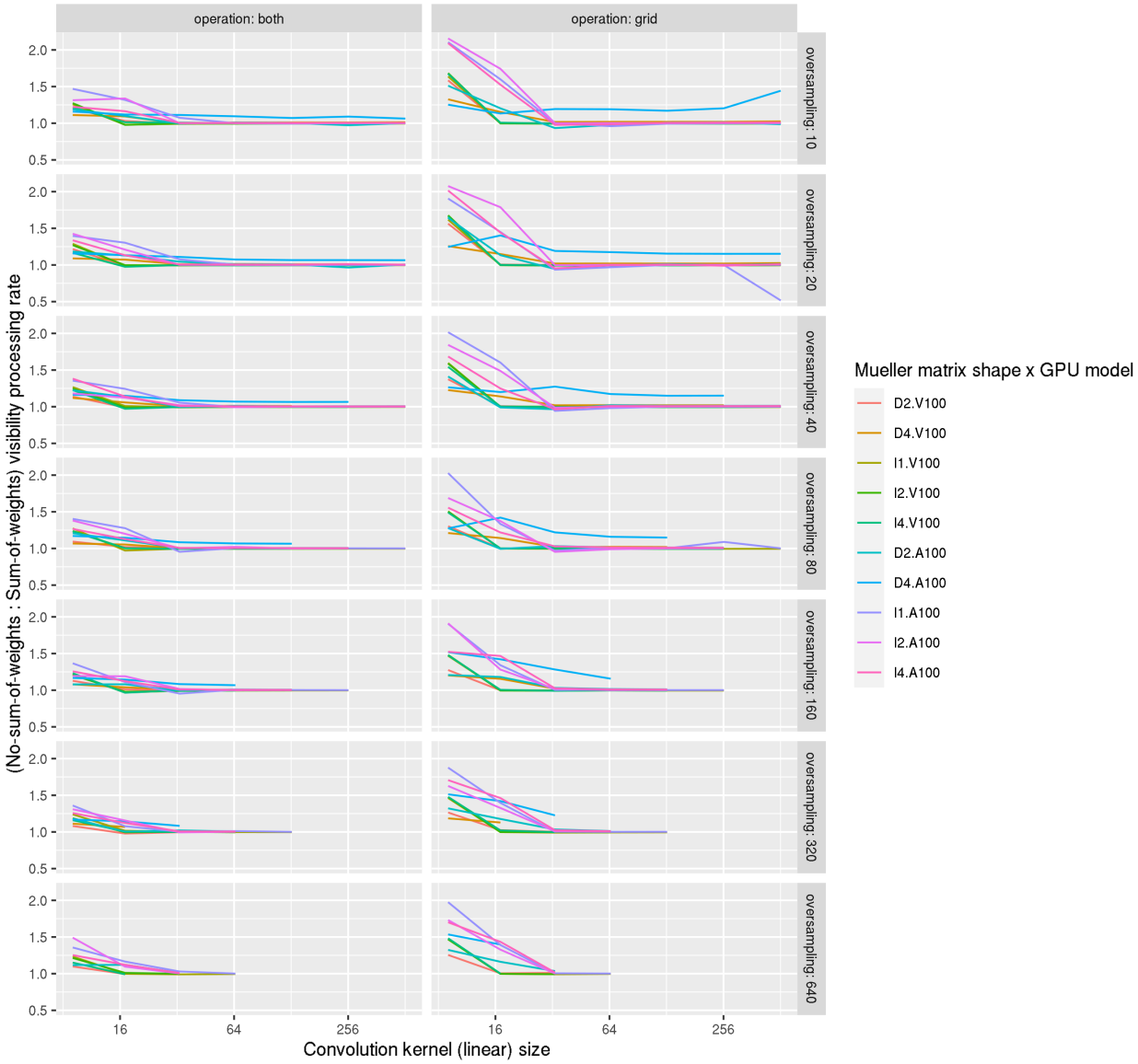

Figure 8: Effect of sum-of-weights computation

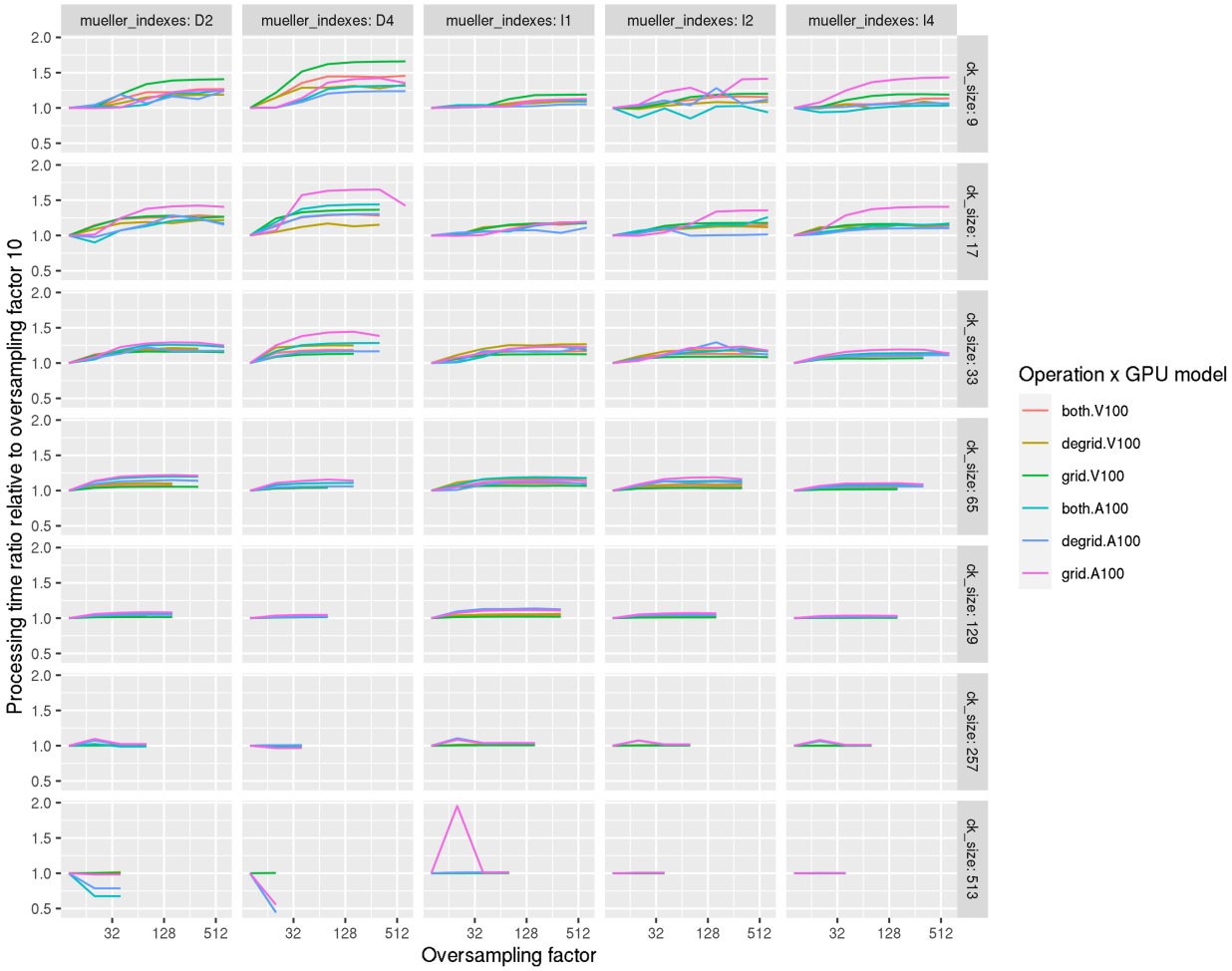

Figure 9: Effect of oversampling factor

Evaluation

Dependence on GPU model

As the HPG source code is implemented using the Kokkos core libraries[20], the identical source code can be built to run on a specific version of GPU at compile time. Assuming that the gridding kernel is memory bandwidth limited as described above, and memory accesses are to GPU system memory, performance metric ratios between models of GPUs should be proportional to the ratio of GPU memory bandwidths. This description of expected relative performance is, of course, based on a simple model, and actual relative performance will depend on detailed differences of GPU architectures, as well as the ability of the Kokkos core libraries to optimize code for each architecture. As the memory bandwidths of the tested V100 and A100 systems are 900 GB/sec and 1555 GB/sec, respectively, we should expect roughly a 1555/900, or 1.73 performance ratio for HPG gridding with these two GPU architectures. As shown in figure 6, pmeasured performance ratios exhibit a complex dependence on gridding parameters, but expectations based on the simple model of memory bandwidths do appear to be reasonable. We conclude that the Kokkos core library is highly effective for the efficient implementation of performance portable code.

Dependence on choice of operation

One expects processing time to be proportional to the square of the linear size of the convolution kernel; although not shown here, linear regressions of log-log plots in figure 7 in every case verify the exponent is very close to the expected value.

The details of which of the gridding or degridding kernels performs better is complex. Naively, one would expect degridding to perform better than gridding, due to degridding using only updates to local data. Measurements indicate that actual performance is somewhat more complicated: at small convolution kernel sizes gridding performs better than degridding in many cases, although as kernel sizes increase, the performance ranking of kernels meets expectations.

Dependence on sum-of-weights computation

Computing the sum of weights generally needs only to be done once during the imaging process (during gridding). To avoid doing excessive computation, HPG allows users to select whether or not to do this computation, but the runtime cost of the sum-of-weights computation is unclear *a priori*, as the computational cost is a small fraction of the cost of gridding. Test results (figure 8) show some benefit of avoiding the sum-of-weights computation, in particular for smaller convolution kernel sizes.

[20] Part of the Kokkos C++ Performance Portability Programming EcoSystem. For more information, see https://github.com/kokkos/kokkos.

Dependence on oversampling factor

The layouts of the convolution kernel, model and grid values in HPG were designed for maximum performance. The layout of the convolution kernel values, especially, was designed to minimize effects on performance of the oversampling factor. The test results, as shown in figure 9, do exhibit some dependence on the oversampling factor, especially for dense Mueller matrices and small convolution kernel sizes. In most cases, the performance penalty with increasing oversampling factor grows up to some plateau level, with the plateau level decreasing with increasing convolution kernel size.

HPG with CASA

Performance tests

Work on integrating HPG with CASA for further testing is currently ongoing. A test version of CASA that uses HPG for degridding and gridding is complete, and full scale imaging testing for accuracy and performance using VLASS data is underway. One of the outcomes of these tests will be an evaluation of the impact of HPG gridding on overall imaging performance.

Scaling tests

Whatever the outcome of current testing with HPG and CASA for VLASS, ngVLA imaging performance will ultimately depend strongly on the scaling characteristics of imaging with the number of processes, and system characteristics (processor type, clock rate, memory, I/O). Given that the GPU implementation in HPG is the highest performing implementation, scaling of imaging performance with number of GPUs is of the greatest interest. Two different scaling dimensions are possible: multiple GPUs in a single cluster node, and multiple nodes in a cluster. Eventually both scaling dimensions should be explored to determine the optimal performance and resource usage. Preliminary planning is underway to develop these tests, but the test program has not been started; results will appear in a following memo. Investigations have begun into the feasibility of using high-performance GPU platforms at the National Center for Supercomputing Applications (to which the ngVLA computing IPT was recently granted access) for these scaling tests.

Other applications of HPG

Full-size ngVLA simulations

It is currently impractical to create full-size ngVLA simulated data sets using CASA due to performance limitations. With the performance improvements of HPG, this limitation will be raised significantly, allowing the creation of more accurate simulated data sets to enable improved evaluations of ngVLA performance under a wide range of use cases.

Appendices

Current HPG architecture

The most recent version of HPG is geared to users of the top-level API, as manifested in `hpg.hpp`. The design of lower levels of the implementation, however, is structured to permit future developments that will support access to the library functionality at these levels. Much of the implementation currently is present in the `hpg.cc` and `hpg_impl.hpp` source files, and it is not recommended that users write code against the interfaces therein. For all intents and purposes, the current architecture is accurately depicted in figure 10.

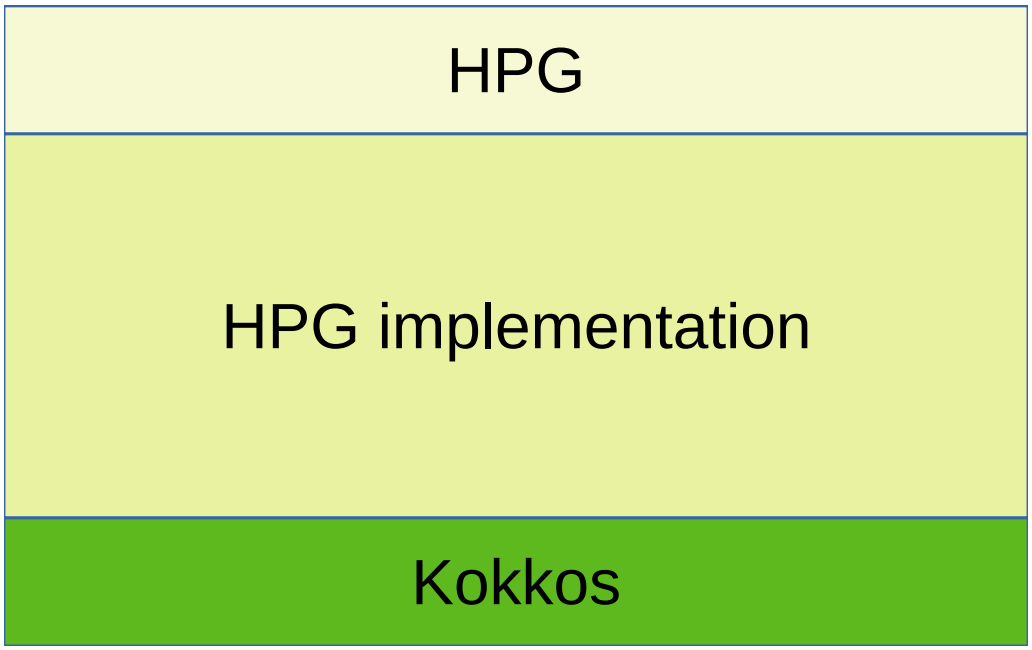

Figure 10: Current architecture

Nevertheless, interested developers may get some sense of a latent, future architectural design within those files.

Future HPG architecture

Current plans are to design and implement an architecture within the current HPG implementation to allow client codes to access HPG functionality at lower levels. Conceptually, the architectural layers, with progressively lower layers containing progressively fewer built-in features will be designed as follows.

HPG high level API (`Gridder`, `GridderState`)
HPG-impl implementation of high level API (`hpg.cc`, or direct access to runtime and Kokkos)
HPG-runtime stream management, data movement, kernel launches

HPG-core compute kernels

The intent of this architecture is primarily to provide a clean interface to the HPG compute kernels, opening the possibility of calling those kernels from alternative runtime code.

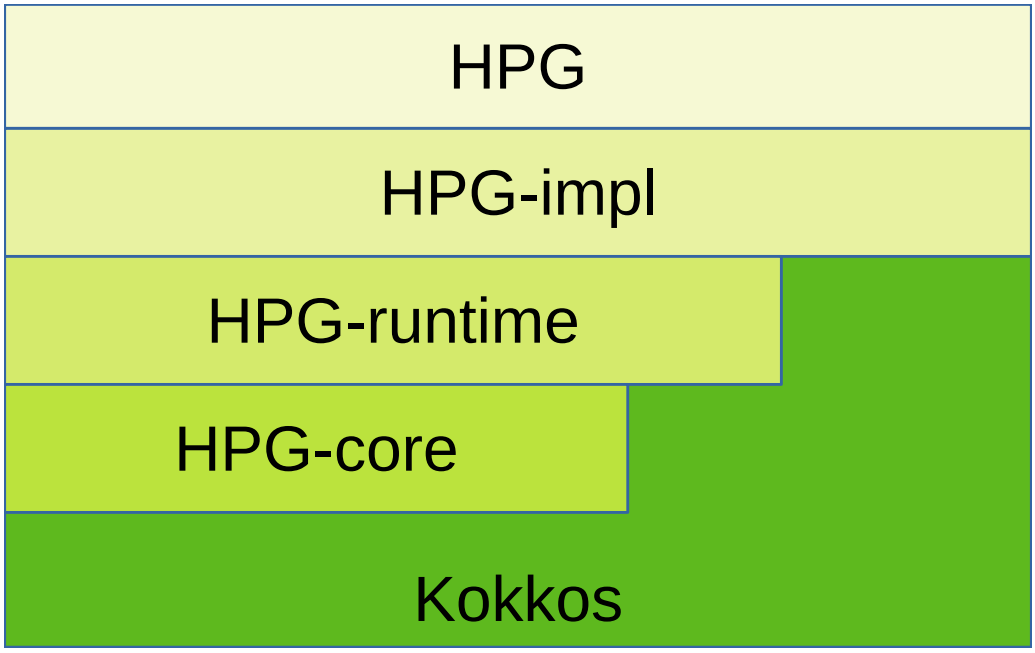

Figure 11: Future architecture

References

Bhatnagar, Sanjay, Rafael Hiriart, and Martin Pokorny. 2021. "Size-of-Computing Estimates for ngVLA Synthesis Imaging." *Next Generation Very Large Array Computing Memos*. http://library.nrao.edu/public/memos/ngvla/NGVLAC_04.pdf.

Edwards, H. Carter, Christian R. Trott, and Daniel Sunderland. 2014. "Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns." *Journal of Parallel and Distributed Computing* 74 (12): 3202–16. https://doi.org/10.1016/j.jpdc.2014.07.003.

Madsen, Jonathan R., Muaaz G. Awan, Hugo Brunie, Jack Deslippe, Rahul Gayatri, Leonid Oliker, Yunsong Wang, Charlene Yang, and Samuel Williams. 2020. "Timemory: Modular Performance Analysis for HPC." In *High Performance Computing*, 12151:434–52. Springer, Cham. https://doi.org/10.1007/978-3-030-50743-5_22.

Rao, Urvashi, Sanjay Bhatnagar, Maxim A. Voronkov, and Tim J. Cornwell. 2009. "Advances in Calibration and Imaging Techniques in Radio Interferometry." *Proceedings of the IEEE* 97 (8): 1472–81. https://doi.org/10.1109/JPROC.2009.2014853.

"Synthesis Imaging Using CASA tclean." n.d. Casadocs. Accessed August 19, 2021. https://casadocs.readthedocs.io/en/stable/notebooks/synthesis_imaging.html.