2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00081

Jigsaw: A Slice-and-Dice Approach to Non-uniform FFT Acceleration for MRI Image Reconstruction

Brendan L. West
University of Michigan
westbl@umich.edu

Jeffrey A. Fessler
University of Michigan
fessler@umich.edu

Thomas F. Wenisch
University of Michigan
twenisch@umich.edu

***Abstract*—The Fast Fourier Transform (FFT) is a fundamental algorithm in signal processing; significant efforts have been made to improve its performance using software optimizations and specialized hardware accelerators. Computational imaging modalities, such as MRI, often rely on the Non-uniform Fast Fourier Transform (NuFFT), a variant of the FFT for processing data acquired from non-uniform sampling patterns. The most time-consuming step of the NuFFT algorithm is "gridding," wherein non-uniform samples are interpolated to allow a uniform FFT to be computed over the data. Each non-uniform sample affects a window of non-contiguous memory locations, resulting in poor cache and memory bandwidth utilization. As a result, gridding can account for more than 99.6% of the NuFFT computation time, while the FFT requires less than 0.4%.**

We present *Slice-and-Dice*, a novel approach to the NuFFT's gridding step that eliminates the presorting operations required by prior methods and maps more efficiently to hardware. Our GPU implementation achieves gridding speedups of over 250× and 16× vs prior state-of-the-art CPU and GPU implementations, respectively. We achieve further speedup and energy efficiency gains by implementing *Slice-and-Dice* in hardware with JIGSAW, a streaming hardware accelerator for non-uniform data gridding. JIGSAW uses stall-free fixed-point pipelines to process M non-uniform samples in approximately M cycles, irrespective of sampling pattern—yielding speedups of over 1500× the CPU baseline and 36× the state-of-the-art GPU implementation, consuming ~200 mW power and ~12 mm^2 area in 16 nm technology. *Slice-and-Dice* GPU and JIGSAW ASIC implementations achieve unprecedented end-to-end NuFFT speedups of 8× and 36× compared to the state-of-the-art GPU implementation, respectively.

I. INTRODUCTION

One of the ten most influential signal processing algorithms of the 20th century [22], the Fast Fourier Transform (FFT) quickly approximates the Discrete Fourier Transform using a divide-and-conquer approach. As an integral component of countless applications, much work has been done to increase FFT performance, including algorithmic optimizations—such as those used in the well-known FFTW library [9], [13]—and specialized instructions and hardware units embedded in modern processors. However, the conventional FFT is applicable only to data that is uniformly sampled—sample coordinates must have equal spacing, such as integers on a Cartesian grid. Imaging applications such as magnetic resonance imaging (MRI) [1], [4], [10], [16], [23], [28], [30], computed tomography [24], [31], synthetic aperture radar [11], [15], and radio astronomy [14], [29] use non-uniform sampling to enable reduced imaging scan time or irregular sensor placement.

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

To enable quick processing of an irregular data set, the Non-uniform FFT (NuFFT) extends the FFT to non-uniform sampling patterns [6]. The NuFFT uses a three step process: (1) non-uniform interpolation, or gridding, (2) apodization, or weighting, and (3) a normal (uniform) FFT. Fast, efficient, and accurate NuFFT operations are of paramount importance for applications where sparse sampling patterns enable real-time imaging tasks and/or large problem sizes. Unfortunately, while FFT performance has significantly improved over the years, NuFFT performance has lagged severely behind.

A decade ago, 85–95% of the computation time required for the NuFFT was due to the gridding step [16], [17], [19], [27], wherein non-uniform samples are interpolated onto a uniform grid so that an FFT can be computed. However, with the vastly improved FFT performance available today using state-of-the-art processors and software libraries, we find that gridding now requires upwards of 99.6% of the NuFFT computation time using a representative 2D data set [25]. The reason that gridding dominates computation time is fairly straightforward: each non-uniform sample in the data set, which is often randomly ordered, affects a window of non-contiguous points on the uniform grid. With non-uniformly spaced samples, prefetching and caching mechanisms in modern processors are unable to alleviate the widening gap between processor and memory speeds. The lack of spatial locality, minimal temporal locality, and resultant poor cache utilization create massive memory bandwidth utilization problems for gridding implementations. With the rise in real-time [8] and iterative image reconstruction techniques [5]—particularly in 3D, wherein millions of NuFFTs are taken iteratively to reconstruct a single volume—NuFFT performance is key to computing answers quickly and enabling emerging applications.

In an attempt to overcome these challenges, many optimized variants of gridding have been proposed to improve performance of the NuFFT [1]–[3], [10], [12], [18], [19], [23], [27]. The most popular method, a form of geometric tiling known as *binning*, pre-sorts the non-uniform samples into "bins" corresponding to distinct regions, or *tiles*, of the uniform grid [12]. These tiles are often configured such that they are small enough to fit in an on-chip cache or memory [2], [3], [19]. Binning sequentially processes tile–bin pairs, improving memory bandwidth use by reducing the

978-1-6654-4066-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00081

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.

number of cache evictions caused by spatially diverse reads and writes. Binning is common in both software and hardware accelerator works [2], [3], [10], [18], [19], [23], [27], with GPU and FPGA implementations among the best in terms of performance and power efficiency. However, despite these optimizations achieving gridding performance improvements ranging up to 40×, they only partially mitigate the bottleneck—NuFFT computation time remains dominated by gridding.

The prior works suffer from three distinct disadvantages: (1) they require a pre-processing step to sort the non-uniform samples, (2) they process elements multiple times when the interpolation kernel overlaps multiple regions, and (3) they do not fully alleviate the memory system bottlenecks, causing computational stalls even in FPGA implementations.

This work presents a novel approach to NuFFT gridding, *Slice-and-Dice*, that obviates the binning step in a manner that maps efficiently to parallel hardware, such as a GPU or ASIC design. Rather than processing a single tile at a time, as in binning, *Slice-and-Dice* stacks the tiles to create "dice." *Slice-and-Dice* processes contributions from an input sample to all corresponding points in the dice in a manner that mitigates the bandwidth utilization problems encountered by prior works. On a GPU, *Slice-and-Dice* achieves an average gridding speedup of over 250× and 16× when compared to the CPU baseline [7] and state-of-the-art GPU implementation [10], respectively. When run as part of the complete NuFFT algorithm, *Slice-and-Dice* GPU results in an end-to-end speedup of over 118× the CPU baseline and 8× the prior work, with equal gridding and FFT computation time.

To fully demonstrate the acceleration potential of our *Slice-and-Dice* model, we design and implement JIGSAW. Using a small set of computational pipelines, JIGSAW performs gridding with time complexity equal to the number of non-uniform samples—irrespective of the non-uniform sampling pattern, interpolation kernel width, or uniform grid size—in a single pass over the data. JIGSAW performs all operations in 32-bit fixed-point, simultaneously decreasing hardware complexity, lowering power requirements, and bringing the reconstruction error closer to the ideal (double precision) as compared to the prior works. With a pipelined architecture and sufficient caching to handle all outstanding memory requests, JIGSAW experiences no stalls—leading to an average gridding speedup of over 1500× relative to the CPU baseline [7], over 95× when compared to the state-of-the-art GPU implementation [10], and 6× relative to *Slice-and-Dice* on GPU. Implemented in SystemVerilog and synthesized using an industrial 16nm node, JIGSAW requires an area of ~12 mm^2 and a power budget of only ~200 mW—nearly 1300× more power efficient than our GPU *Slice-and-Dice* implementation. JIGSAW results in an end-to-end NuFFT speedup of over 258× vs the CPU baseline and 36× the state-of-the-art GPU implementation, with gridding consuming only 25% of the computation time.

II. Background and Motivation

The Fast Fourier Transform is widely used in applications such as signal and image processing to quickly compute the Fourier Transform of evenly-spaced data; i.e., the coordinates (or array indices) lie on a uniform grid, such as pixels in an image. However, computational imaging applications often rely on non-uniform sampling patterns—such as spiral and radial scans in MRI—to reduce latency and enable emerging algorithms and sensor configurations. These non-uniform patterns result in data that does not lie on a uniform grid. For irregularly sampled data, applications must rely on the Non-uniform Fast Fourier Transform (NuFFT). To understand the algorithmic differences imposed by non-uniform data, we first review the Non-uniform Discrete Fourier Transform (NuDFT), which directly computes the Fourier Transform of non-uniform data. We then take an in-depth look at the NuFFT, which provides an efficient approximation of the NuDFT by combining an interpolation step—used to map the non-uniform samples onto a uniform grid—with a traditional FFT.

A. Non-uniform Discrete Fourier Transform

A generalization of the Discrete Fourier Transform, the NuDFT allows for processing of non-uniform data. Following the notation in [17], [27], given a set of M non-uniform samples $\{x_j\}$ and a uniform Cartesian grid with N points in each of d dimensions, let f_j denote the complex Fourier coefficient corresponding to the non-uniform sample x_j. For the complex Fourier coefficient $\hat{f}_k$ corresponding to the uniform points k in $\{0, \ldots, N-1\}^d$, the forward NuDFT is used to compute

$$f_j = \sum_{\boldsymbol{k} \in \{0,\ldots,N-1\}^d} \hat{f}_{\boldsymbol{k}} \, \mathrm{e}^{-2\pi i \boldsymbol{k} \cdot \boldsymbol{x}_j}, \quad j = 0, \ldots, M-1 \quad (1)$$

The adjoint NuDFT is similarly defined as

$$\hat{h}_{\boldsymbol{k}} = \sum_{j=0}^{M-1} \hat{f}_j \, \mathrm{e}^{2\pi i \boldsymbol{k} \cdot \boldsymbol{x}_j}, \quad \boldsymbol{k} \in \{0, \ldots, N-1\}^d \quad (2)$$

Equations (1) and (2) can also be written as matrix-vector products, as shown in the following equations:

$$\boldsymbol{f} = \boldsymbol{A}\hat{\boldsymbol{f}} \quad (3) \qquad \hat{\boldsymbol{h}} = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{f} \quad (4)$$

where $\boldsymbol{A}$ denotes the $M \times N^d$ matrix whose elements are the complex exponential terms above.

Direct calculation of these operations requires MN^d floating-point operations, which is too expensive for many applications, even for small problem sizes. Worse, direct "inversion" of the $\boldsymbol{A}$ matrix would require immense amounts of memory, quickly becoming prohibitive as the matrix grows.

B. Non-uniform Fast Fourier Transform

The NuFFT extends the traditional FFT to support non-uniform data, providing approximate solutions to the NuDFT with significant reductions in computational complexity and memory requirements. Using three steps, (1) interpolation, (2) apodization (i.e., amplitude weighting), and (3) an FFT, the NuFFT computes nearly the same result as the NuDFT but with a computational complexity of only $M + N^d \log(N^d)$—orders of magnitude lower than the NuDFT for useful data sizes. The NuFFT has several variants to handle different combinations of uniform and non-uniform inputs and outputs,

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.

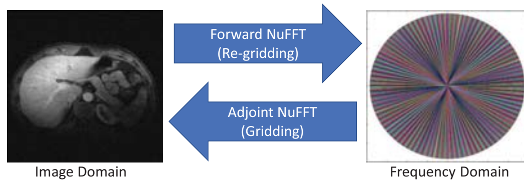


Fig. 1: Each NuFFT variant comprises three steps. Forward: (1) pre-apodization, (2) FFT, (3) regridding. Adjoint: (1) gridding, (2) FFT, (3) de-apodization. Image data from [25].

with the forward and adjoint NuFFTs a staple in image reconstruction. As shown in Figure 1, the forward NuFFT transforms image data to the frequency domain, while the adjoint NuFFT transforms frequency data to the image domain. The interpolation step dominates the NuFFT's computation time, accounting for upwards of 99.6% [7].

Often called gridding in the adjoint NuFFT and re-gridding in the forward NuFFT, this interpolation step—visualized in Figure 2—transforms the data between a uniform grid and a set of non-uniform samples using an interpolation kernel. Each sample has a corresponding interpolation window of W^d uniform points, where W is the width of the window. The distance from the sample to each of the uniform points within its interpolation window is used to determine the kernel weight, where points closer to the sample use a larger weight than those further away. The supported non-uniform coordinate granularity is defined by the table oversampling factor, L, which determines the number of weights between each point W in the interpolation kernel. There are WL discrete interpolation weights for each dimension of the interpolation kernel window, and locations within the interpolation window are rounded to the nearest weight. By constraining the kernel granularity, offline precomputation and storage of the discrete kernel weights in a look-up table (LUT) is possible for hardware with limited on-chip memory, reducing the amount of online computation required for each interpolation operation.

Due to the periodicity of complex exponential functions, the uniform grid is a torus in the frequency domain. As a consequence, any sample lying within $W/2$ of an "edge" of the grid will involve interpolation using "neighbors" that are determined using periodic boundary conditions (i.e., the interpolation window affects points on opposite sides of the grid). This wrapping is visualized in Figure 2, where the interpolation kernels of samples a, c, and f wrap to other "sides" of the grid, requiring complicated circular boundary checks to determine the points lying within the window. The interpolation kernel itself can be one of a variety of windowing functions, such as Kaiser-Bessel, Gaussian, B-spline, Sinc, etc. The choice of windowing function is application-specific.

To improve the NuFFT interpolation accuracy, an oversampling factor σ, set to two in Figure 2, is multiplied by the uniform and non-uniform coordinates prior to an FFT being applied. Oversampling increases the resolution of the resulting grid, reducing overall signal noise. While crucial for accuracy, oversampling comes with two undesirable traits: (1) increased FFT computation time, as an N sample FFT now becomes a

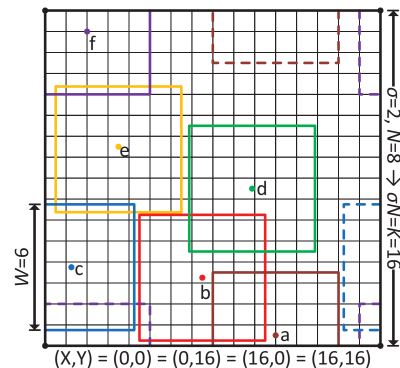


Fig. 2: Uniform grid (flattened torus) with $d = 2$ dimensions, $M = 6$ input samples, base grid dimension $N = 8$, oversampling factor $\sigma = 2$, and interpolation kernel width $W = 6$.

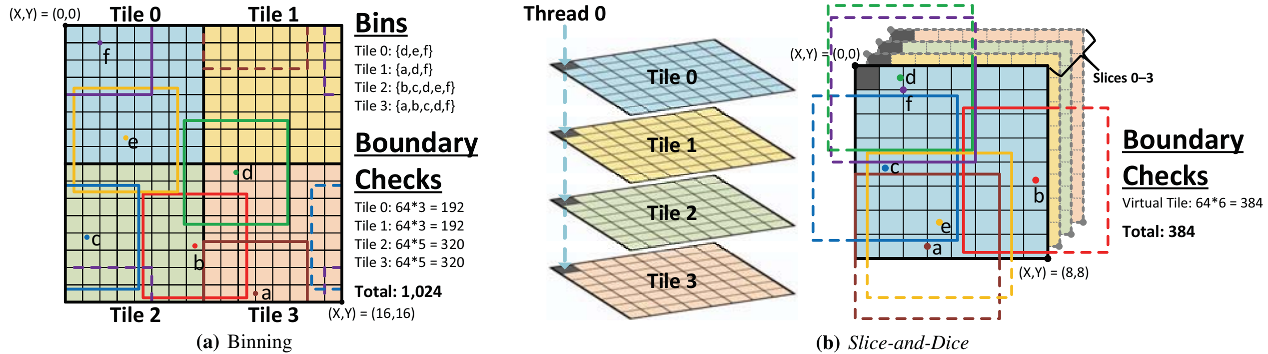
σN FFT along each dimension, and (2) increased gridding memory requirements as N or d grow. To alleviate these effects, Beatty et al. proposed a method of gridding using smaller oversampling factors, i.e., $\sigma \leq 2$. However, when the oversampling factor σ is reduced, the interpolation kernel must be widened (i.e., larger W) to maintain accuracy [1]. While a smaller σ leads to faster FFT operations—by processing a smaller grid—and lower memory requirements, a wider interpolation kernel increases latency and causes the NuFFT to be even further dominated by the interpolation operation.

C. Traditional Gridding

As the dominant computational component of the NuFFT, optimizing the interpolation step of the NuFFT has been a focus of many previous works [1]–[4], [7], [10], [16]–[19], [23], [27]. To understand why gridding dominates NuFFT computation time, we first consider a typical gridding implementation and its basic parameters. As an example, we consider the adjoint NuFFT, wherein gridding is performed to map non-uniform frequency data onto a uniform grid prior to computing an FFT. Whereas gridding's computational *complexity* depends primarily on the number of non-uniform samples M and the number of uniform grid points N^d, the computation *time* can depend substantially on other implementation parameters. In particular, gridding time depends on the oversampling factor σ and the interpolation kernel width W. Commonly set to four or six, the interpolation kernel width determines how many points in the target grid are affected by each non-uniform sample. This width is fixed and is not affected by the grid dimensions or resolution; rather, its size is often determined by the choice of σ as part of the trade-off between accuracy, memory requirements, and computation time [1]. We next enumerate common gridding inefficiencies and discuss how existing solutions fail to address them.

Gridding is not easily parallelizable. Whereas gridding has tractable computational complexity, modern hardware realities—notably the significant gap between memory and processor speed—can lead to considerable slowdowns based on the implementation. Non-uniform samples—often arriving in effectively random order—each affect a window of W^d

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.



(a) Binning (b) *Slice-and-Dice*

Fig. 3: Binning vs *Slice-and-Dice*. In this example, binning performs a boundary check between each of the 64 uniform points in a tile and each non-uniform sample in its associated bin. Due to some samples affecting multiple tiles—and therefore being placed in multiple bins—binning processes 16 samples. In contrast, *Slice-and-Dice* obviates the need to presort the data by performing a two step boundary check. *Slice-and-Dice* performs a single comparison between the "top" view and each sample to determine which relative coordinates (i.e., "columns") in the stack are affected; a combination of the tile coordinates gives the tile index of the uniform point affected within the stack.

uniform points that are discontiguous in memory, which results in gridding commonly suffering from poor memory locality.

The simplest gridding implementation processes the randomly-ordered non-uniform samples serially. Any uniform point lying within $W/2$ distance of the sample's coordinates is accumulated with a distance-based contribution of the sample's magnitude, with points closer to the sample's coordinates receiving a greater contribution than those further away. Once all affected uniform points have received their updates, the next sample is processed. Such a serial approach benefits from being able to quickly determine which points are affected by a given sample and avoiding write conflicts among samples with overlapping interpolation kernels. However, since caches are too small to store the entire output grid, nearly all grid point accesses incur an off-chip read-modify-write miss. Moreover, this input-oriented approach has limited parallelism, failing to take advantage of massively multithreaded systems.

Instead, GPU and FPGA implementations commonly turn to output-oriented parallelism, wherein one thread or pipeline accumulates all sample values that affect a single grid point. This approach does not need any synchronization among threads, since each modifies disjoint memory locations. However, output-parallel implementations suffer from a significant drawback: there is no way to determine if a thread is affected by a sample without performing a distance boundary check between the sample and the thread coordinates. Although a single non-uniform sample only affects W^d uniform points (or threads), a naïve output-parallel implementation must perform a boundary check between each non-uniform sample and every grid point, requiring M boundary checks for each of N^d uniform grid points. These boundary checks are almost as expensive as the interpolation operation itself—a table lookup and multiply-accumulate. Furthermore, since the target grid dimension N is usually far larger than the interpolation kernel width W, the vast majority of the checks will fail, undermining the effectiveness of output-parallel gridding.

Binning suffers from additional overheads. To reduce the number of boundary checks and global memory accesses encountered using output-driven parallelism, modern NuFFT implementations instead rely on a form of geometric tiling known as *binning*. Binning, visualized in Figure 3a, breaks the uniform grid into small subsections, or *tiles*, the dimensions of which are chosen such that a single tile fits in the on-chip cache of the target system. The non-uniform samples are then pre-sorted into subsets, or *bins*, corresponding to the tiles that they affect. Rather than performing a boundary check between every sample and every uniform point in the entire grid, samples in each bin must only be checked against the uniform points in the associated tile. Tile–bin pairs are processed sequentially, significantly reducing global memory accesses after a tile is fully loaded into the on-chip cache. In Figure 3a, Tile 0 has an associated bin consisting of samples $\{d, e, f\}$, Tile 1 has an associated bin $\{a, d, f\}$, and so on. Processing the samples in this manner allows for significant execution time improvement, as the tile remains cached and only the samples in the bin must be read from global memory.

While binning greatly improves memory locality and reduces stalls due to global memory accesses, it still suffers from several factors that result in suboptimal computation time. First, the non-uniform data must be pre-sorted to map well to the hardware; good binning parameters are hardware and data-set dependent and are not always readily evident. If the wrong parameters are chosen, such as the tiles not maximally utilizing the available cache, binning performance can be severely limited. Second, non-uniform samples lying within $W/2$ of tile edges affect multiple tiles, requiring those samples to be processed as part of multiple bins. As an example, in Figure 3a, samples d and f must be placed in all four bins, resulting in a significant increase in redundant boundary checks. Third, and perhaps most important, binning limits parallelism. While binning improves cache hit rates, its restriction of memory accesses to a single tile severely limits the available Memory-Level Parallelism (MLP). With limited MLP, instruction reordering is insufficient to entirely hide the memory latency, causing computational stalls due to pending memory requests. GPU-based implementations try to parallelize across titles to leverage the massive Thread-

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.

Level Parallelism (TLP) of GPUs. However, their approach undermines the original goal of binning—improving locality—as different warps evict one another's data from the cache. As a result, neither CPUs nor GPUs are able to entirely hide the memory latency via Instruction-Level Parallelism (ILP) or TLP, negatively impacting the performance due to memory stalls. GPU-based implementations also suffer from severe branch divergence as all threads within a warp (operating on different points in a tile) are not affected by all samples within a bin, resulting in massive under-utilization of SIMD execution lanes. More specifically, with warp and interpolation kernel sizes T and W, T/W threads will be unaffected—and thus idle—when processing every sample.

D. Summary and Goals

NuFFT computation is dominated by the gridding operation, which can account for upwards of 99.6% of computation time. Even with the optimizations proposed by the prior work, gridding suffers from limited parallelism, poor memory bandwidth utilization, redundant computation, and computational stalls due to outstanding memory requests. Our objective is to find an approach that eliminates (1) pre-processing of the data set, (2) duplicate sample processing, (3) and the limited MLP encountered using binning, to enable an efficient streaming model for high-throughput gridding computation. Using hardware/software co-design, we aim to break down the gridding algorithm into core components, redesigning gridding from the ground up to better map to commodity parallel architectures and enable novel hardware accelerators.

III. Slice-and-Dice Design

We next introduce the *Slice-and-Dice* processing model for NuFFT gridding and describe its design choices, which facilitate a streaming model that is able to unlock substantial performance gains in conventional parallel architectures, such as GPUs, as well as enable highly optimized accelerators.

In contrast to optimizations such as binning, *Slice-and-Dice* obviates the pre-processing step of sorting the data into bins, instead incorporating the sorting step into a two-part boundary check. *Slice-and-Dice* uses a stacked tile memory layout, which increases MLP, reduces the number of boundary checks, and enables streaming implementations by guaranteeing interaction-free processing of parallel threads or pipelines.

Rethinking Binning. Most recent NuFFT acceleration implementations use binning to improve spatial locality. However, sorting data samples into the appropriate bins requires an additional step in the gridding process. Worse, inevitably some samples are assigned into up to four bins (or more, with higher dimensionality) because their interpolation window intersects adjacent tiles, resulting in redundant boundary checks. As a result, whereas binning improves locality, its performance potential significantly suffers from (1) precomputation to sort the samples, (2) unnecessary boundary checks, and (3) a lack of (memory-level) parallelism.

As an alternative to binning, we introduce *Slice-and-Dice*, a novel gridding optimization technique which obviates the need for an additional pre-sorting step and drastically reduces the number of boundary checks required. We design *Slice-and-Dice* by leveraging a tiling-based, output-driven parallel model and a unique decomposition of the sample coordinates. As shown in Figure 3b, *Slice-and-Dice* breaks the target grid into multiple, smaller tiles, each of dimensions T^d. The tiles, which we will refer to as *virtual tiles* hereafter, are stacked to form *dice*. Whereas binning sequentially processes tiles that each fit into the on-chip cache, *Slice-and-Dice* leaves the dice in global memory, instead relying on overlapping memory requests and using the cache to exploit memory locality opportunities available in the dice layout. In this model, a block of T^d threads processes the entire grid, where each thread processes a single position in each of the virtual tiles, corresponding to a "column" in the dice. For example, Thread 0 in Figure 3b will process the uniform grid points at relative position $(x, y) = (0, 0)$ in each of the four virtual tiles. As long as the interpolation window is smaller than the tile size—i.e., $W \leq T$—this layout guarantees that each sample will only affect a single point in any column.

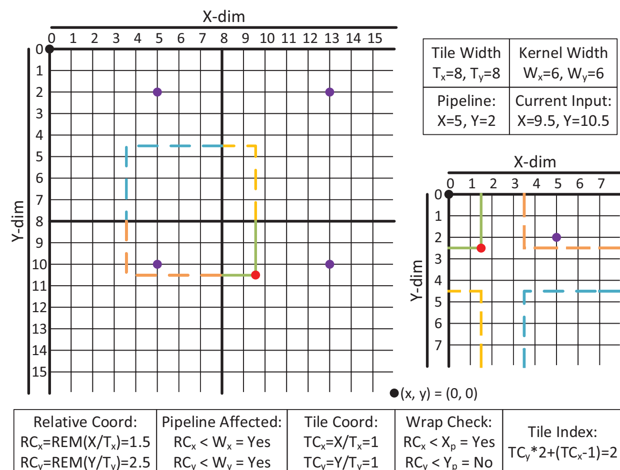


Fig. 4: *Slice-and-Dice* in action. The thread assigned to uniform points with relative positions $(x, y) = (5, 2)$ is affected by an input sample if both relative coordinates are less than the interpolation kernel width. This thread is affected in tile $(0, 1)$ but the sample's coordinate is in tile $(1, 1)$, causing a wrap in the X dimension.

Efficiently handling boundary checks in the stacked tile model is enabled by a novel decomposition of the input samples' coordinates, as shown in Figure 4. For coordinates ranging from $[0{:}N)$ in each dimension, we perform a two-part decomposition by dividing each coordinate by the virtual tile size. The division's quotient is the *tile coordinate*, while the remainder is the *relative coordinate*. The relative coordinate indicates where the sample lies within the tile (i.e., in which column), and is used to determine whether the current input sample affects the uniform points assigned to a given thread. For each sample, *Slice-and-Dice* performs a boundary check only against every column in the stack of virtual tiles, rather than every point in the target grid—resulting in MT^d total checks. Since points are processed sequentially in

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.

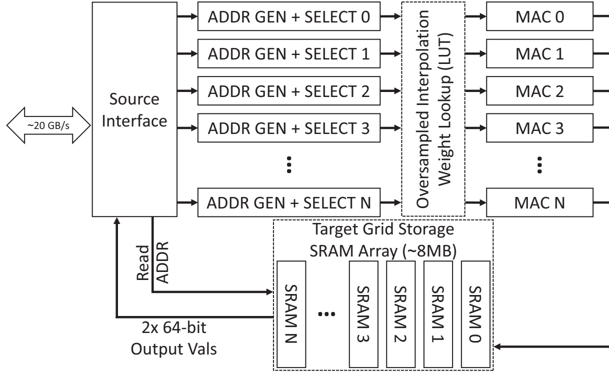


Fig. 5: JIGSAW microarchitecture. Operating at 1.0 GHz, non-uniform samples arrive on a 128-bit bus and are broadcast to all pipelines in parallel. After gridding is complete, two 64-bit uniform target points are read through the bus each cycle.

output-driven parallelism—while their effects are applied in parallel—the block of threads quickly determines which samples affect their assigned target points. Using an arrangement in which the target grid points assigned to each thread are placed in a contiguous array, for each affected thread we next determine the index into that array (i.e., the "depth" in the column) by finding the *global tile address*, a combination of the tile coordinates in each dimension—much like calculating a total linear index in GPU programming. Even with an all-sample to all-column comparison, use of the *Slice-and-Dice* binning-free model results in a computational complexity reduction of N^d/T^d versus a naïve parallel implementation. In Figure 4—where different colors in the interpolation window indicate different depths (tiles) in the dice—we demonstrate the coordinate decomposition, tile index calculation, and wrap compensation utilized by *Slice-and-Dice* for a single non-uniform sample.

IV. JIGSAW MICROARCHITECTURE

A primary bottleneck in many gridding acceleration works lies within the memory subsystem, due to poor bandwidth utilization, non-contiguous memory accesses, or insufficient memory-level parallelism. Even in FPGA implementations, the memory layout requires that each pipeline have access to any point within the target tile, resulting in high inter-pipeline communication requirements and memory system contention. In *Slice-and-Dice*, we use the stacked slice memory layout to eliminate unnecessary hardware interaction in custom accelerators. With a single hardware thread, or pipeline, assigned to process the contributions to each column in the dice, memory accesses can be controlled on a per-pipeline basis—allowing for deterministic, constant performance. To demonstrate the full potential of *Slice-and-Dice* when coupled with a custom processing pipeline and memory hierarchy, we present the JIGSAW streaming architecture for 2D and 3D NuFFT gridding acceleration. As a hardware implementation of *Slice-and-Dice*, JIGSAW uses a streaming approach to fully process an input stream in a single stall-free pass, accumulating contributions to the uniform target grid columns stored in private memory arrays. This leads to runtime linear in the number of non-uniform samples, independent of ordering of the input samples, uniform target grid size, or interpolation window width.

TABLE I: JIGSAW System Parameters.

Property	Value
Target Grid Dimensions (N)	8–1024
Virtual Tile Dimensions (T)	8
Interpolation Window Dimensions (W)	1–8
Table Oversampling Factor (L)	1–64
Pipeline Bit Width	32-bit
Interpolation Weight Bit Width	16-bit

We implement JIGSAW in SystemVerilog to enable functional verification and obtain power/area synthesis estimates. As illustrated in Figure 5, JIGSAW comprises a set of identical 32-bit fixed-point pipelines logically arranged as a 2D grid. To support both 2D and 3D gridding operations, there are two variants of JIGSAW: JIGSAW 2D and JIGSAW 3D Slice. In this section, we describe the basic hardware implementation of JIGSAW, followed by the differences between processing 2D and 3D data. For clarity, we assume a target grid with $N = 1024$ points in each dimension, virtual tiles with $T = 8$ points in each dimension, an interpolation window width of $W = 6$, and an interpolation table oversampling factor of $L = 32$. JIGSAW comprises pipelines logically arranged as a 2D grid of dimensions T^2, to match the virtual tile size. JIGSAW's range of supported runtime parameters are listed in Table I. Each pipeline $T_{x,y}$ is split into four stages: select, weight lookup, interpolation, and accumulate.

Select. The select unit, shown in Figure 5, is responsible for determining whether a sample affects grid points assigned to a given pipeline, calculating the relative and tile coordinates, the global tile address, and the table addresses for each dimension. To determine whether a pipeline's points are affected by a sample, the select unit calculates the distance from the sample to the relative coordinates (i.e., the column) assigned to the pipeline. For each dimension, the select unit performs the distance calculation in hardware using two steps: (1) truncating the upper bits of sample's coordinates to obtain the relative coordinate, and (2) adding the tile dimension to the relative coordinate and subtracting the pipeline's index. The resulting value is the forward distance (i.e., left to right in 1D) from the pipeline's column and the input sample. The select unit then compares the distance against the interpolation window size to determine whether the sample affects a point in the column. If the distance in each dimension is less than the interpolation window size, the sample affects a point in the column.

For any sample affecting a point in its column, the select unit uses the previously truncated coordinate bits (the tile coordinate) to determine the appropriate entry in the accumulation array. Interpolation window overlaps onto other tiles in a given dimension, or "wraps," are handled by decrementing the tile coordinate in that dimension. To check if a tile wrap occurred, the select unit compares the relative coordinate against the pipeline index—if the relative coordinate is less than the pipeline index, a wrap has occurred in that dimension; this is visualized in Figure 4. The updated tile coordinates are then

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.

combined to form the global tile address.

The select unit next calculates the table addresses. The table addresses are used to index an oversampled interpolation table, extracting the weights in each dimension necessary to generate the distance-based weight for the final interpolation. The select unit determines the table addresses by multiplying the previously calculated distances by the table oversampling factor—32 in this example—and rounding to the nearest integer. Because the table oversampling factor is a power of two, the multiplication can be implemented efficiently by truncating the lower bits of the distances.

Weight Lookup. Each interpolation weight lookup unit contains a dual-ported SRAM (in the 2D variant) capable of storing up to 256 32-bit complex weights (16 bits for each real and imaginary component), which are used to generate the final weight for the interpolation. As the interpolation window is symmetric around its center, only half of the weights for a given window must be stored: capacity for 256 weights enables an oversampling factor of up to $L = 64$ given a maximum interpolation kernel width of $W = 8$. The weight lookup unit multiplies the complex weights for each dimension—retrieved according to the addresses generated in the select unit—to calculate the final weight. Because weights are complex numbers, the unit performs the multiplication using three real multiplication operations and five real addition/subtraction operations, as described by Knuth [21].

Interpolation. The interpolation unit, part of the Multiply-and-Accumulate (MAC) blocks in Figure 5, multiplies the complex interpolation weight and the complex sample magnitude, again using Knuth's method [21]. The interpolation unit then forwards the resulting value, which represents the weighted contribution of the input sample to the target grid point, to the accumulation unit.

Accumulation. The accumulation unit maintains the sum of the interpolated values for all points in the pipeline's associated *Slice-and-Dice* column. In JIGSAW, we collocate the adders used for accumulation with local SRAM arrays that store the partial sums for the set of points assigned to the corresponding pipeline. Inputs to the accumulation unit are the newly calculated interpolation value and the global tile address—previously calculated in the select unit—which acts as an index into the output array. Once the data stream is complete, data stored in SRAM arrays is read out tile by tile.

Gridding in 2D and 3D. Due to the large memory requirements found in 3D NuFFT processing—approximately 8GB for a 1024^3 target grid—modern algorithms and accelerators often process 3D volumes in a series of 2D slices. We follow this trend with JIGSAW, implementing support for a third coordinate dimension in the select and interpolation weight lookup pipeline stages. Since JIGSAW only has ~8MB of on-chip SRAM, a $N^3 = 1024^3$ target grid is processed by iterating over 2D slices; i.e., a 1024^3 grid is broken into 1024 slices of 1024^2 points each. While this decreases the overall gridding performance of the accelerator, as unsorted input data must be processed up to N_z times, this approach requires minimal additional logic and reaps the benefits of on-chip memory, thereby allowing stall-free execution of each layer.

System Integration. JIGSAW can be interfaced with a host system much like other standalone accelerators, such as GPUs and FPGAs. Input data is transmitted to JIGSAW from the host via a direct memory access (DMA) stream, with one non-uniform sample and its associated coordinates arriving each cycle. Using a DMA stream instead of individual copy commands frees the host to continue operations while JIGSAW performs gridding asynchronously. With a synthesized clock speed of 1.0 GHz, JIGSAW is able to transmit and receive data at DDR4 bandwidth (~20GB/s). Once the data stream is complete, the DMA controller notifies the host via an interrupt signal. The host then initiates a second stream, which transfers the gridded data from JIGSAW to the host memory. The lightweight communication is enabled by JIGSAW's fully-provisioned hardware architecture—no delay is required between the host-to-device stream completing and the device-to-host stream being initiated, minimizing end-to-end latency.

V. EVALUATION METHODOLOGY

To evaluate the *Slice-and-Dice* model and our JIGSAW microarchitecture, we implement *Slice-and-Dice* in GPU and SystemVerilog ASIC variants with virtual tiles of dimension 8×8. We employ the Michigan Image Reconstruction Toolbox (MIRT) [7]—a matrix-based Matlab implementation that uses double-precision floating point for all calculations—as a baseline for both quality and performance. The performance of the CPU-based MIRT is on par with that of the well-known NFFT [17] library, but separates the gridding and FFT steps, allowing for a more direct comparison.

To highlight how *Slice-and-Dice* performs versus a state-of-the-art GPU implementation, we compare to Impatient [10]. Impatient is a GPU-accelerated framework for non-Cartesian sampling trajectories in MRI. Using an output-driven parallelism gridding approach combined with binning, Impatient achieves significant speedups versus direct matrix inversion. With the fastest publicly available code base—updated in 2018 to support new-generation GPUs—Impatient provides a comparison to *Slice-and-Dice* on traditional parallel systems.

Our test system uses an Intel i9-9900KS with 128 GB of DDR4 3600 MHz memory for the CPU-based benchmarks, and an Nvidia Titan Xp for the GPU-based benchmarks. The GPU implementation of *Slice-and-Dice* uses single-precision floating-point values to closely match the prior work, while the ASIC implementation uses 32-bit fixed-point pipelines. We estimate the performance of our JIGSAW ASIC implementation using its synthesized 1.0 GHz clock frequency, the pipeline depth (12 cycles), and the number of non-uniform input samples. Functional verification and quality evaluation is performed against MIRT's output using doubles.

For power and area analysis, we synthesize our SystemVerilog implementations of JIGSAW 2D and JIGSAW 3D Slice using an industrial 16 nm node and a 1.0 GHz clock speed.

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.

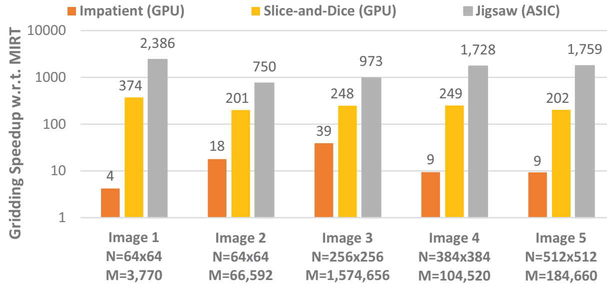


Fig. 6: Gridding speedups, normalized to MIRT.

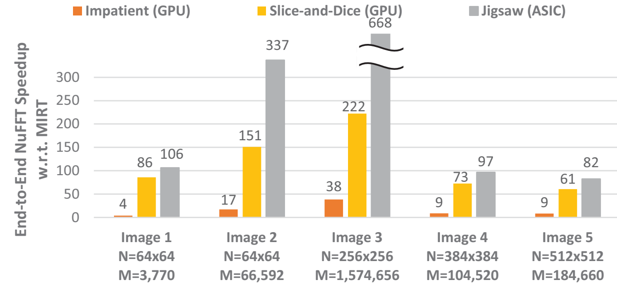


Fig. 7: End-to-end NuFFT speedups, normalized to MIRT.

VI. Results

A. Performance Comparison

We evaluate gridding performance using five images of differing dimension and number of non-uniform samples. Figure 6 and Figure 7 show the gridding and end-to-end NuFFT performance of each implementation normalized to the performance of MIRT. Since 3D gridding is a derivative of 2D gridding—serially operating on 2D slices—for the compared implementations, we only report results for the 2D case.

GPU. The *Slice-and-Dice* GPU implementation relies on the CUDA threading model to hide the long external memory latencies through quick context switching among thread warps. *Slice-and-Dice* GPU uses blocks of 8×8 threads, where each thread is assigned to a single uniform grid point in each virtual tile. A single block does not contain nearly enough threads to fully utilize the GPU's processing units; we therefore populate a grid of 128×128 blocks to improve occupancy, with each block operating on its own subset of the non-uniform input data and writing to the shared output grid. This implementation breaks the *Slice-and-Dice* output-parallelization model, as multiple threads may now write to the same output location. We use atomic addition instructions to ensure proper synchronization and that no updates are lost. In this manner, the *Slice-and-Dice* GPU implementation achieves an average gridding speedup of over 250× relative to the baseline and approximately 16× over Impatient [10] for all problem sizes, as shown in Figure 6. The end-to-end NuFFT performance improvements in Figure 7 follow a similar trend, with an average speedup of over 118× relative to the baseline and 8× over Impatient. This dramatic increase in performance relative to the prior work arises for several reasons: (1) *Slice-and-Dice* GPU uses a lookup table for interpolation weights, while Impatient [10] calculates them during processing, (2) *Slice-and-Dice* GPU achieves an L2 hit rate of ~98% compared to Impatient's ~80%, (3) *Slice-and-Dice* achieves an occupancy of ~80% compared to the ~47% for Impatient, and (4) *Slice-and-Dice* GPU utilizes parallelism across both the non-uniform input array and the output grid, allowing for far more computational overlap between warps. In short, *Slice-and-Dice* maps more efficiently to GPU hardware.

ASIC. Jigsaw 2D, *Slice-and-Dice*'s ASIC implementation for 2D gridding, comprises a set of 8×8 pipelines with a 12-cycle pipeline latency from input to accumulation. Using the fixed-point pipeline and on-chip buffers to hide memory latency, Jigsaw is fully pipelined and does not experience any hardware stalls, accepting a new non-uniform sample each cycle. This uninterrupted data flow provides constant throughput: total runtime is proportional to the input data set size, irrespective of sampling pattern, uniform target grid size, or interpolation window width. Given the pipeline depth of 12 cycles and the synthesized 1.0 GHz clock speed, the runtime of an M-sample input is $M+12$ cycles, or $M+12$ nanoseconds.

TABLE II: Jigsaw Synthesis Results in 16 nm Technology.

Jigsaw with 1.0 GHz Clock	Power	Area
2D (8MB SRAM)	216.86 mW	12.20 mm^2
2D (no accum SRAM)	94.22 mW	0.42 mm^2
3D Slice (8MB SRAM)	104.36 mW	12.42 mm^2
3D Slice (no accum SRAM)	63.62 mW	0.64 mm^2

The Jigsaw 3D Slice variant extends Jigsaw's 2D implementation to support a third dimension in the input data and output grid. Like the 2D variant, the 3D implementation is synthesized for a 1.0 GHz clock speed; 3D support extends the pipeline depth to 15 cycles. The 3D Slice variant performs 3D gridding in N_z sequential steps, where N_z is the size of the grid in the Z dimension. For an arbitrary, unsorted 3D dataset of M samples, the runtime is $(M+15)*N_z$ cycles. However, if the dataset is pre-sorted into subsets of samples affecting each Z-dimension slice—essentially binning in the Z-dimension and letting *Slice-and-Dice* obviate binning in 2D—runtime can be reduced to $(M+15)*W_z$ cycles, where W_z is the width of the interpolation kernel in the Z dimension.

With a streaming architecture and no memory or computational stalls, Jigsaw offers orders of magnitude better performance than all prior works. Figure 6 demonstrates an average gridding speedup of approximately 1500× relative to the baseline. End-to-end NuFFT performance is also drastically increased in Figure 7, with speedups of over 258× and 16× compared to the baseline and prior work, respectively.

B. Power & Area

Table II reports our synthesis results, with the Jigsaw 2D and 3D Slice variants shown both with and without target grid SRAM to illustrate the amount of power and area required for the accelerator pipeline and lookup tables. The 2D variant has an estimated power consumption of 216.86 mW and an area of 12.20 mm^2. Approximately 95% of this area is used for the on-chip storage of the 1024×1024 uniform target grid, which

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.

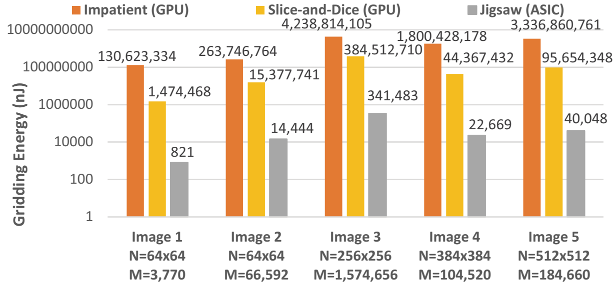
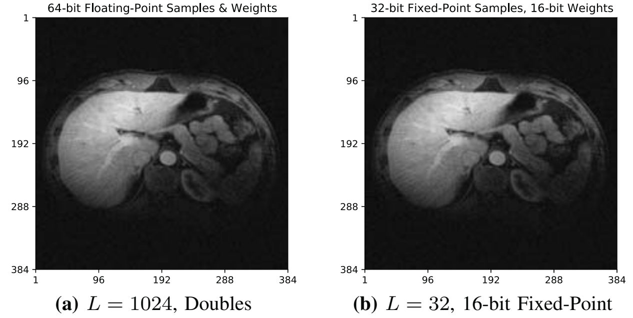


Fig. 8: Energy requirements of gridding implementations.



(a) $L = 1024$, Doubles **(b)** $L = 32$, 16-bit Fixed-Point

Fig. 9: Direct NuFFT reconstructions with differing table oversampling factors and numeric precision. Image data from [25].

is also responsible for over 56% of the power consumption. The 3D Slice variant shows lower power consumption due to reduced switching activity, as each slice fully processes only a subset of the non-uniform points (approximately M/N_z); only the select stage processes all M points for any individual slice.

To showcase the energy efficiency achieved by JIGSAW, we compare it to the single-precision floating-point GPU gridding implementations of *Slice-and-Dice* and Impatient in Figure 8. Across all of the images tested, Impatient energy consumption averages 1.95 J, while *Slice-and-Dice* GPU averages 108.27 mJ. In contrast, JIGSAW consumes only 83.89 μJ—an energy reduction of over 23000× compared to Impatient and nearly 1300× compared to *Slice-and-Dice* GPU.

C. Image Quality

We verify JIGSAW image quality using visual comparison of 2D liver slices from [25] and their associated normalized root mean square difference (NRMSD). Using our SystemVerilog implementation to simulate the hardware, we feed the non-uniform sample data into the JIGSAW pipeline, comparing the final output grid to that of our Matlab reference implementation using doubles. As shown in Figure 9, our fixed-point hardware produces images indistinguishable from those produced with double-precision floating-point, even when the oversampling factor is reduced by a factor of 32×. Similarly, the NRMSD percentages for 32-bit floating-point and our 32-bit fixed-point implementations are 0.047% and 0.012%, respectively—$1/4$ the error while halving the ALU width and table storage requirements. These results demonstrate that JIGSAW's fixed-point hardware contributes negligible image quality degradation compared to the reference [20].

VII. Related Work

Gridding acceleration methods have been previously proposed using algorithmic optimizations [7], [23], [26] and hardware accelerators, such as GPUs [10], [27] and FPGAs [2], [3], [18], [19]. We describe several works from each category.

A. Algorithmic Optimizations

[23], [26] describe variations of an algorithmic approach that breaks the uniform grid into smaller tiles and pre-sorts the non-uniform samples into bins based on which tile(s) they affect. This approach, today commonly known as binning, reduces computational complexity in parallel implementations by lowering the number of total boundary checks for a sample.

The Michigan Image Reconstruction Toolbox (MIRT) [7], a collection of open-source algorithms for MRI reconstruction and related imaging problems widely used in the imaging community, relies on optimized matrix processing and compiled executables to efficiently perform gridding using both interpolation table and sparse matrix implementations.

B. GPU Implementations

In one of the earliest GPU gridding works, Sørensen et al. [27] implement a model in which each thread is responsible for the interpolation operations affecting a set of neighboring points, with all points held in registers during computation. Operating on sets of eight points in parallel, Sørensen et al. achieve a 20-85× speedup versus a contemporary CPU implementation for various problem sizes and sampling trajectories.

More recently, a Toeplitz-based MRI reconstruction strategy using binning was implemented on GPUs [10]. To reduce data races caused by multiple non-uniform samples affecting the same uniform points, every uniform grid point was assigned to a different thread. Paired with a Kaiser-Bessel interpolation kernel, this output-driven parallel model achieved speedups of over 200× compared to the same group's previous direct matrix inversion (NuDFT) approach to NuFFT computation [30].

In comparison to these works, we eliminate the need to pre-sort the data for binning by instead using our *Slice-and-Dice* decomposition model, which "sorts" samples within the boundary check operation. Our GPU implementation combines input- and output-driven parallelism, parallelizing across distinct subsets of the input samples and between columns in the dice. With thousands of blocks operating on the same grid, overlapping computation effectively hides memory latency.

C. FPGA Implementations

Kestur et al. [19] implemented a variant of gridding using binning in an FPGA, dynamically adding non-uniform samples to linked lists associated with each tile. Focusing on maximizing write throughput of the samples, the implementation uses contiguous local memory storage to improve spatial locality; a trajectory-optimized variant was later implemented [18].

In [2], [3], Cheema et al. design an FPGA gridding accelerator which implements binning using a set of fixed-size FIFOs. The FPGA reads non-uniform samples from external memory and sorts them into the FIFOs, with an arbiter determining

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.

which FIFO should be processed. The associated tile is then loaded into on-chip memory, operating on 16 points in parallel.

In contrast, JIGSAW is integrates the sorting process into the boundary check, reducing the end-to-end latency of the gridding operation. JIGSAW further enhances performance by storing the entire target grid in a set of SRAM arrays—where each array holds a single column in the dice—and operating on all tiles in parallel. JIGSAW's specialized parallel architecture eliminates all pipeline stalls, using the *Slice-and-Dice* model to achieve trajectory-agnostic, deterministic performance.

VIII. Conclusion

In this work we presented *Slice-and-Dice*, a model for NuFFT gridding that maps efficiently to traditional parallel hardware architectures. *Slice-and-Dice* eliminates preprocessing of the non-uniform data by performing binning on-the-fly, breaking the target grid into multiple tiles and operating on all tiles simultaneously. When implemented on a GPU, *Slice-and-Dice* achieves average speedups of over 250× and 16× when compared to the CPU baseline and GPU state-of-the-art implementations for representative problems.

We then described JIGSAW, a streaming accelerator implementation of *Slice-and-Dice* that performs gridding with runtime proportional to the number of non-uniform samples—irrespective of uniform target grid size, interpolation kernel width, or sampling pattern. Our 32-bit fixed-point implementation achieves orders of magnitude better performance and lower power while retaining better error characteristics over single-precision floating point. Our evaluation of JIGSAW demonstrates gridding speedups of over 1500× and 95× when compared to the CPU baseline and state-of-the-art GPU implementations, respectively. Synthesized using an industrial 16 nm technology node, the 2D variant of JIGSAW achieves a 1.0 GHz clock frequency with an area of ~12mm^2 and a power consumption of ~200 mW. JIGSAW's fixed-point streaming implementation is nearly 1300× more power efficient than *Slice-and-Dice* GPU.

Slice-and-Dice GPU and JIGSAW achieve end-to-end NuFFT speedups of over 118× and 258× the CPU baseline, and 8× and 36× the state-of-the-art GPU implementation, resulting in the FFT being the bottleneck for the first time.

References

[1] P. J. Beatty, D. G. Nishimura, and J. M. Pauly, "Rapid gridding reconstruction with a minimal oversampling ratio," *IEEE Transactions on Medical Imaging*, Jun. 2005.

[2] U. I. Cheema, G. Nash, R. Ansari, and A. Khokhar, "Memory-optimized re-gridding architecture for non-uniform fast Fourier transform," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Jul. 2017.

[3] U. I. Cheema, G. Nash, R. Ansari, and A. A. Khokhar, "Power-efficient re-gridding architecture for accelerating non-uniform fast Fourier transform," in *Int. Conf. on Field Prog. Logic and Appl. (FPL)*, Sep. 2014.

[4] J. A. Fessler, "On NUFFT-based gridding for non-Cartesian MRI," *Journal of magnetic resonance (San Diego, Calif. : 1997)*, Nov. 2007.

[5] J. A. Fessler, "Model-based image reconstruction for MRI," *IEEE Signal Processing Magazine*, Jul. 2010.

[6] J. A. Fessler and B. P. Sutton, "Nonuniform fast Fourier transforms using min-max interpolation," *IEEE Trans. Signal Proc.*, Feb. 2003.

[7] J. A. Fessler, "Michigan image reconstruction toolbox (MIRT)." [Online]. Available: https://web.eecs.umich.edu/~fessler/code/

[8] J. Frahm, D. Voit, and M. Uecker, "Real-time magnetic resonance imaging: Radial gradient-echo sequences with nonlinear inverse reconstruction," *Investigative Radiology*, Dec. 2019.

[9] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, Feb. 2005.

[10] J. Gai, N. Obeid, J. Holtrop, X.-L. Wu, F. Lam, M. Fu, J. Haldar, W.-m. Hwu, Z.-P. Liang, and B. Sutton, "More IMPATIENT: a gridding-accelerated Toeplitz-based strategy for non-Cartesian high-resolution 3D MRI on GPUs," *Journal of parallel and distrib. computing*, May 2013.

[11] X. Y. He, X. Y. Zhou, and T. J. Cui, "Fast 3D-ISAR image simulation of targets at arbitrary aspect angles through nonuniform fast Fourier transform (NUFFT)," *IEEE Trans. Antennas Propagation*, May 2012.

[12] K. O. Johnson and J. G. Pipe, "Convolution kernel design and efficient algorithm for sampling density correction," *Mag. Res. Med.*, Feb. 2009.

[13] S. G. Johnson and M. Frigo, "Implementing FFTs in practice," in *Fast Fourier Transforms*, Sep. 2008.

[14] Junklewitz, H., Bell, M. R., Selig, M., and Enßlin, T. A., "RESOLVE: A new algorithm for aperture synthesis imaging of extended emission in radio astronomy," *Astronomy & Astrophysics*, Feb. 2016.

[15] H. Kajbaf, J. T. Case, Y. R. Zheng, S. Kharkovsky, and R. Zoughi, "Quantitative and qualitative comparison of SAR images from incomplete measurements using compressed sensing and nonuniform FFT," in *IEEE RadarCon (RADAR)*, May 2011.

[16] D. D. Kalamkar, J. D. Trzaskoz, S. Sridharan, M. Smelyanskiy, D. Kim, A. Manduca, Y. Shu, M. A. Bernstein, B. Kaul, and P. Dubey, "High performance non-uniform FFT on modern x86-based multi-core systems," in *IEEE Int. Parallel and Distrib. Processing Symposium*, May 2012.

[17] J. Keiner, S. Kunis, and D. Potts, "Using NFFT 3—a software library for various nonequispaced fast Fourier transforms," *ACM Transactions on Mathematical Software*, Aug. 2009.

[18] S. Kestur, K. Irick, S. Park, A. Al Maashri, V. Narayanan, and C. Chakrabarti, "An algorithm-architecture co-design framework for gridding reconstruction using FPGAs," in *Des. Automat. Conf*, Jun. 2011.

[19] S. Kestur, S. Park, K. M. Irick, and V. Narayanan, "Accelerating the nonuniform fast Fourier transform using FPGAs," in *IEEE Int. Symp. on Field-Programmable Custom Computing Machines*, May 2010.

[20] F. Knoll, T. Murrell, A. Sriram, N. Yakubova, J. Zbontar, M. Rabbat, A. Defazio, M. J. Muckley, D. K. Sodickson, C. L. Zitnick, and M. P. Recht, "Advancing machine learning for MR image reconstruction with an open competition: Overview of the 2019 fastMRI challenge," *Magnetic Resonance in Medicine*, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.28338

[21] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing, 1997.

[22] G. Madey, X. Xiang, S. E. Cabaniss, and Y. Huang, "Agent-based scientific simulation," *Computing in Science & Engineering*, Jan. 2005.

[23] N. M. Obeid, I. C. Atkinson, K. R. Thulborn, and W.-M. W. Hwu, "GPU-accelerated gridding for rapid reconstruction of non-Cartesian MRI," in *Proc. of the Int. Society for Magnetic Resonance in Medicine*, 2010.

[24] Y. Z. O'Connor and J. A. Fessler, "Fourier-based forward and back-projectors in iterative fan-beam tomographic image reconstruction," *IEEE Transactions on Medical Imaging*, May 2006.

[25] R. Otazo, E. Candès, and D. K. Sodickson, "Low-rank plus sparse matrix decomposition for accelerated dynamic MRI with separation of background and dynamic components," *Magnetic Reson. in Med.*, 2015.

[26] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W.-M. W. Hwu, "GPU acceleration of cutoff pair potentials for molecular modeling applications," in *Proc. 5th Conf. on Computing Frontiers*, May 2008.

[27] T. S. Sørensen, T. Schaeffter, K. ø. Noe, and M. S. Hansen, "Accelerating the nonequispaced fast Fourier transform on commodity graphics hardware," *IEEE Transactions on Medical Imaging*, Apr. 2008.

[28] B. Sutton, D. Noll, and J. Fessler, "Fast, iterative image reconstruction for MRI in the presence of field inhomogeneities," *IEEE Transactions on Medical Imaging*, Mar. 2003.

[29] S. Wenger, M. Magnor, Y. Pihlström, S. Bhatnagar, and U. Rau, "SparseRI: A compressed sensing framework for aperture synthesis imaging in radio astronomy," *Astronomical Soc. of the Pac.*, Nov. 2010.

[30] X. Wu, J. Gai, F. Lam, M. Fu, J. P. Haldar, Y. Zhuo, Z. Liang, W. Hwu, and B. P. Sutton, "Impatient MRI: Illinois Massively Parallel Acceleration Toolkit for image reconstruction with enhanced throughput in MRI," in *Int. Symp. Biomed. Imag.: From Nano to Macro*, Mar. 2011.

[31] D. Xu, Y. Huang, and J. U. Kang, "GPU-accelerated non-uniform fast Fourier transform-based compressive sensing spectral domain optical coherence tomography," *Optics Express*, Jun. 2014.

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 18,2021 at 18:18:15 UTC from IEEE Xplore. Restrictions apply.