Streaming Architectures for Medical Image Reconstruction

by

Brendan Lee West

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2021

Brendan Lee West

westbl@umich.edu

ORCID iD: 0000-0002-4245-4620

To my parents, Harold West and Germaid West, whose dedication to my education taught me the value of lifelong learning.

ACKNOWLEDGEMENTS

When first starting my graduate work, I believed that earning a Ph.D. demonstrated a great achievement by a single individual. However, over time I have learned that this could not be further from the truth—earning a Ph.D. is not the achievement of an individual; it takes a village. Without the support of my family, friends, labmates, and advisor, I never would have achieved this dream. In particular I want to acknowledge Robert and Samantha Nantau, who welcomed me into their family and gave me a "home away from home" during my college years—you may never truly comprehend how much that meant (and still means) to me.

On the academic side, I first want thank my advisor, Thomas F. Wenisch, for believing in my potential when even I questioned it and for his somewhat strange (but undeniably useful...for pranks) association with unicorns. I would also like to thank my fellow Sanctuary Lab members: Akshitha Sriraman, Amirhossein Mirhosseini, Harini Muthukrishnan, Hossein Golestani, Ofir Weisse, Steve Zekany, and Vaibhav Gogte. Although we have been mostly apart for the past year or more, I will always remember our conversations, adventures, and pranks with fondness. Thank you all for making our lab environment so enjoyable.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Non-invasive imaging modalities have recently seen increased use in clinical diagnostic procedures. Unfortunately, emerging computational imaging techniques, such as those found in 3D ultrasound and iterative magnetic resonance imaging (MRI), are severely limited by the high computational requirements and poor algorithmic efficiency in current parallel hardware—often leading to significant delays before a doctor or technician can review the image, which can negatively impact patients in need of fast, highly accurate diagnosis. To make matters worse, the high raw data bandwidth found in 3D ultrasound requires on-chip volume reconstruction with a tight power dissipation budget—dissipation of more than 5 W may burn the skin of the patient. The tight power constraints and high volume rates required by emerging applications require orders of magnitude improvement over state-of-the-art systems in terms of both reconstruction time and energy efficiency. The goal of the research outlined in this dissertation is to reduce the time and energy required to perform medical image reconstruction through software/hardware co-design. By analyzing algorithms with a hardware-centric focus, we develop novel algorithmic improvements which simultaneously reduce computational requirements and map more efficiently to traditional hardware architectures. We then design and implement hardware accelerators which push the new algorithms to their full potential.

In the first part of this dissertation, we characterize the performance bottlenecks of high-volume-rate 3D ultrasound imaging. By analyzing the 3D plane-wave ultrasound algorithm, we reduce computational and storage requirements in Delay Compression. Delay Compression recognizes additional symmetry in the planar transmission scheme found in 2D, 3D, and 3D-Separable plane-wave ultrasound implementations, enabling on-chip storage of the reconstruction constants for the first time and eliminating the most power-intensive component of the reconstruction process. We then design and implement TETRIS, a streaming hardware accelerator for 3D-Separable plane-wave ultrasound. TETRIS is enabled by the TETRIS RESERVATION STATION (RS), a novel 2D register file that buffers incomplete voxels and eliminates the need for a traditional load-and-store memory interface. Utilizing a fully pipelined architecture, TETRIS reconstructs volumes at physics-limited rates (i.e., limited by the physical propagation speed of sound through tissue).

Next, we review a core component of several computational imaging modalities, the Non-uniform Fast Fourier Transform (NuFFT), focusing on its use in MRI reconstruction. We find that the non-uniform interpolation step therein requires over 99% of the reconstruction time due to poor spatial and temporal memory locality. While prior work has made great strides in improving the performance of the NuFFT, the most common algorithmic optimization severely limits the available parallelism, causing it to map poorly to the massively parallel processing available in modern GPUs and FPGAs. To this end, we create Slice-and-Dice, a processing model which enables efficient mapping of the NuFFT's most computationally-intensive component onto traditional parallel architectures. We then demonstrate the full acceleration potential of Slice-and-Dice with JIGSAW, a custom hardware accelerator which performs the non-uniform interpolations found in the NuFFT in time approximately linear in the number of non-uniform samples, irrespective of sampling pattern, uniform grid size, or interpolation kernel width.

The algorithms and architectures herein enable faster, more efficient medical image reconstruction, without sacrificing image quality. By decreasing the time and energy required for image reconstruction, our work opens the door for future exploration into higher-resolution imaging and emerging, computationally complex reconstruction algorithms which improve the speed and quality of patient diagnosis.

CHAPTER I

Introduction

Non-invasive medical imaging modalities, such as X-ray CT, magnetic resonance imaging (MRI), and ultrasound have become cornerstones in modern medicine. Today, doctors are able to see high-resolution images of a patient's internals, both statically and in real-time, without the costly and often dangerous invasive diagnostic procedures required in the past.

In recent years, medical imaging has started to experience an evolution from 2D to 3D and real-time imaging capabilities. 3D and real-time imaging offer a more detailed view of the region of interest, without the need to rely on experienced technicians to estimate the "whole" from a set of 2D slices, in a manner which enables faster, more accurate diagnosis. Unfortunately, while great strides have been made in new 3D imaging algorithms, these algorithms often bring exponential increases in raw data volume, computational complexity, and execution time.

While computer performance has improved, in many cases existing architectures are still too slow or inefficient to support real-time data acquisition and image reconstruction. Commercial hardware has been able to reconstruct 3D images for some algorithms, but these often suffer from severe computational bottlenecks—resulting in long reconstruction times and/or poor resolution. As an example, state-of-the-art 3D ultrasound systems can only reconstruct tens of volumes per second [44, 18, 14, 19], which is far too low for emerging applications which require tracking high-frequency motion [37, 23, 55, 50]. These performance problems are caused in part by the algorithms themselves: many algorithms in the medical domain are designed purely to enable a new imaging technique, without consideration of how the algorithm will actually be executed in hardware. To compound the problem, computer architects rarely design modern systems with the unique requirements of medical imaging reconstruction in mind. This has created a disconnect between imaging algorithm developers and hardware architects, with the result being that traditional architectures are far too inefficient to feasibly handle many real-time 3D imaging applications.

Many of the inefficiencies and computational bottlenecks found in reconstruction algorithms can be traced back to current architecture's load-and-store memory system. Medical imaging applications often have memory access patterns which do not mesh well with modern computer caches, leading to severe performance degradation and power consumption that in some cases make imaging entirely infeasible [1, 28, 44, 44, 14, 2]. To continue advancements in medical image reconstruction, algorithm developers and hardware architects must work together to enable new and improved imaging systems.

In this dissertation, we take an in-depth look at two of the most popular imaging modalities, ultrasound and MRI. Through careful benchmarking of each algorithmic component, we identify the primary bottlenecks encountered in traditional parallel systems, such as GPUs. We then partner with signal processing experts to rebuild the algorithms from the ground up. Using a hardware-centric focus, we develop novel algorithmic modifications and computational models which enable dramatic performance improvements when using traditional architectures.

While improved algorithms go a long way toward enabling emerging imaging modalities, not all applications can afford the space or power required by the racks of GPUs needed to reconstruct images in real-time. With this in mind, we demonstrate the full performance and power-saving potential of our algorithms by designing custom hardware accelerator architectures for 3D ultrasound and MRI reconstruction. We obviate the memory subsystem bottlenecks encountered in previous systems by doing away with the costly load-and-store paradigm to which standard architectures have historically adhered to. Using stall-free, fully-pipelined hardware implementations, we achieve orders of magnitude improvements in both performance (latency and throughput) and power efficiency versus traditional systems and prior state-of-the-art research.

1.1 3D Ultrasound

Ultrasound is an imaging modality widely used for non-invasive diagnostics because of its low transmit power, lack of ionizing radiation, and portability compared to X-ray and magnetic resonance imaging (MRI) [48, 42]. Relying on the principle of sonic transmission and reflection, ultrasound acts much like sonar and radar, transmitting sound waves into target volumes by exciting a transducer array with an electronic pulse. As the sound waves pass through the volume, partial reflections occur when the waves encounter tissue interfaces; these reflections are then sampled by the transducer array. The raw reflection signals collected by each transducer are used to reconstruct an image through a process called beamforming, wherein the signals are filtered to coherently sum reflections that originate from the numerous focal points within the image volume.

Beamforming is the most computationally expensive aspect of ultrasound imaging. Identifying the samples within each receive signal that correspond to each focal point is computationally expensive, as it requires numerous trigonometry calculations to calculate the delay constants (indexes into the received signal). In 2D systems, these delays can be trivially calculated ahead of time and stored in on-chip look-up tables to save power and improve performance. Unfortunately, an exponential increase in the number of delays for emerging 3D imaging applications precludes this possibility. Worse, the data rate found in 3D systems—often over 500 Gbps—is too high to offload to an external machine for reconstruction, instead requiring beamforming to occur within the ultrasound probe handle. This in turn adds another layer of complexity: the probe handle's power dissipation must be kept under 5 W, as higher dissipation may burn the skin of the patient [44].

Prior works have made great strides in lowering the computational complexity of ultrasound algorithms [64, 66, 63], delay approximation [44, 14], and custom accelerator architectures [44, 14, 19, 18]. Unfortunately, even the most performant of these are only able to reconstruct tens of volumes per second. Emerging 3D ultrasound applications in cardiac imaging [37], vector flow imaging [23, 55], and shear wave elastography [50] require extremely high volumetric image acquisition rates to track high frequency motion—orders of magnitude higher than possible using prior approaches.

In the first half of this dissertation, we discuss our work on 3D ultrasound algorithmic optimizations and hardware architectures. We first introduce Delay Compression, a novel decomposition of the 3D plane-wave beamforming algorithm's most computationally intensive component: calculation of the round-trip delay constants. Applicable to 2D, 3D, and 3D-Separable beamforming variants, Delay Compression reduces the number of delay constants requiring distinct calculation by a factor equal to the number of receive transducers in the ultrasound probe head—1024× using our 3D system parameters. Such a dramatic reduction in computational requirements results not only in the algorithm mapping more efficiently to traditional hardware accelerators, such as GPUs, but also enables first-of-its-kind custom hardware solutions for applications with extremely tight power constraints and performance requirements.

Next, we describe our work on TETRIS, a novel hardware accelerator for separable ultrasound beamforming that implements Delay Compression and enables volume acquisition rates up to the physical limits of acoustic propagation delay. TETRIS operates in a streaming fashion—without requiring on-chip storage of the entire receive signal—reconstructing volumes in real-time. For a representative imaging task, TETRIS generates physics-limited ~13,000 volumes per second in an estimated ~2 W system power budget. The TETRIS beamformer has an unprecedented power efficiency of 2.03 tera-beamforming operations per watt (tera-BOPS)—an increase in efficiency of nearly 3× compared to the prior work.

1.2 Magnetic Resonance Imaging

Magnetic resonance imaging (MRI) is a computational imaging modality popular in radiology due to its ability to enable high-resolution non-invasive imaging of anatomy. While it is desirable to visualize physiological processes in a non-invasive manner for diagnostic and post-operative care purposes, MRI historically suffers from long and often uncomfortable scan times—MRI scans can generate sound levels up to 110 decibels and take upwards of 90 minutes [13, 53].

To reduce scan time, MRI applications often rely on non-uniform, frequency-domain sampling trajectories such as radial, spherical, and spiral. Unfortunately, these irregular sampling patterns preclude use of the highly optimized Fast Fourier Transform (FFT) operations commonly found in reconstruction implementations. Instead, applications with irregular sampling patterns turn to the Non-uniform Fast Fourier Transform (NuFFT), an extension of the FFT which enables computation over irregular data. The NuFFT transforms data between the frequency and image domains using a three-step process: (1) interpolation of the non-uniform samples onto or from a uniform grid, (2) a standard [uniform] FFT, and (3) apodization, or weighting of the uniform data.

The time required to compute the NuFFT is dominated by the non-uniform interpolation component, which is responsible for over 99% of the reconstruction time. Because each non-uniform sample affects a window of non-contiguous memory locations, MRI reconstruction algorithms often require nearly random reads and writes. With little spatial and temporal locality available, NuFFT computation is severely limited by poor cache and memory bandwidth utilization [47, 30, 32, 28, 3, 2].

Prior work has offered algorithmic optimizations which focus on the NuFFT's performance by optimizing the memory bandwidth utilization during the non-uniform interpolation step. The most popular approach is a process called *binning* [24, 43]. Binning breaks the uniform grid into *tiles*, presorting the non-uniform samples into *bins* based on which tiles their interpolation windows intersect. Tile–bin pairs are then processed sequentially, with the tile held in on-chip memory to reduce the number of cache misses.

While the algorithmic optimizations available through binning have led to significant speedups in both traditional processors and custom FPGA architectures [47, 38, 32, 31, 12, 3, 2], binning severely limits the amount of available parallelism. Paired with the overhead of presorting the non-uniform samples and maintaining the bins, current NuFFT implementations are insufficient to enable emerging MRI techniques requiring thousands or even millions of NuFFT calls [6, 10].

In the second half of this dissertation, we focus on optimizing the non-uniform interpolations found in MRI reconstructions. We develop a novel grid decomposition model, Slice-and-Dice, that obviates the presorting operations required by prior optimizations and exposes more parallelism to existing hardware architectures. Using a tiled memory layout, Slice-and-Dice results in a 1600× reduction in the number of boundary check operations when using our system parameters. When implemented on a GPU, Slice-and-Dice achieves significant improvements in terms of cache hit rate, latency, and occupancy compared to prior state-of-the-art GPU implementations.

To highlight Slice-and-Dice's full potential when paired with a custom memory subsystem and computational pipeline, we implement JIGSAW, a streaming hardware architecture for NuFFT acceleration. JIGSAW uses a mixture of fixed- and floating-point functional units to form 2D array of stall-free pipelines, performing interpolations with M non-uniform samples in approximately M cycles, irrespective of sampling pattern, interpolation kernel width, or uniform grid size. To enable end-to-end NuFFT acceleration, JIGSAW implements a 2D radix-2 FFT and IFFT operations on top of the Slice-and-Dice tiled memory layout. Processing up to eight FFTs in parallel, JIGSAW achieves unprecedented NuFFT performance while simultaneously reducing power requirements by orders of magnitude.

CHAPTER II

Delay Compression: Reducing Delay Calculation Requirements for 3D Plane-Wave Ultrasound

2.1 Introduction

Ultrasound is computational imaging modality often used in medical diagnosis. Its low cost, lack of ionizing radiation, and no known dangers or side effects provides a safe and economical imaging method compared to modalities such as X-ray and MRI [48, 42], with the added benefit of being amenable to parallel hardware.

Relying on the principle of sonic transmission and reflection, ultrasound acts much like sonar and radar. In its most simplistic form, visualized in Figure 2.1, an array of transducers located in the tip of an ultrasound probe handle transmits sound waves into the target volume by exciting the transducers with an electronic pulse. These sound waves pass through and partially reflect at tissue interfaces or scatter at inhomogeneities, with the scattered wave radiating outward in an expanding spherical geometry. Reflections and scattered waves are then captured by the transducer array by sampling the time-dependent intensity of the signal; different tissue inhomogeneities and interfaces result in varying intensities. The raw reflection signals collected by each transducer are then used to reconstruct a volumetric image through a process called beamforming. In beamforming, the received signals are

The work described in this section was published in DAC 2019 [60], IUS 2019 [58], and IEEE TC 2020 [59].

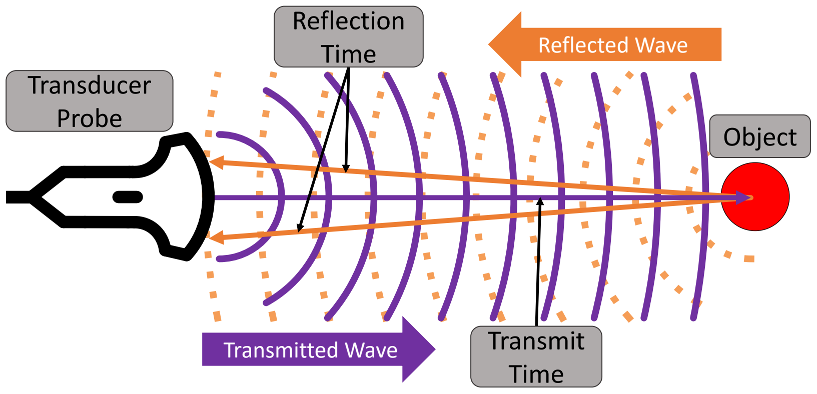


Figure 2.1: Ultrasound basics: (1) a transducer probe transmits sound waves through a target medium. (2) these waves come into contact with a target object, they cause partial reflections of the sound waves which are then captured by the transducer probe. (3) the captured reflections are then beamformed to reconstruct the image.

filtered to coherently sum reflections that originate from numerous focal points within the image volume.

Beamforming is the most computationally expensive aspect of ultrasonic imaging. In beamforming, the samples within each receive signal corresponding to each focal point are identified based on the round-trip time-of-flight and the speed of sound. This distance calculation requires using numerous trigonometry calculations for each focal point. Moreover, there is substantial data sharing and reuse, as each sample may contribute to numerous voxels in the final volumetric image. Hence, although calculating each voxel is in principle embarrassingly parallel, performing these computations efficiently while exploiting data sharing is difficult.

Time-delay beamforming, one of the primary beamforming methods, calculates a time delay for the propagation path from each transmit element to each focal point and then to each receive element. For a target image of size M_xM_z focal points and a receive array of N_x transducers, the number of required delays, and the associated computational complexity, is $M_xM_zN_x$. After sampling the reflected signals, the samples are commonly interpolated to improve axial resolution without the power overhead of faster analog-to-digital converters (ADCs) [4]. After interpolation, samples corresponding to each pixel are selected using the time delays. The selected values then undergo a process called apodization, in which

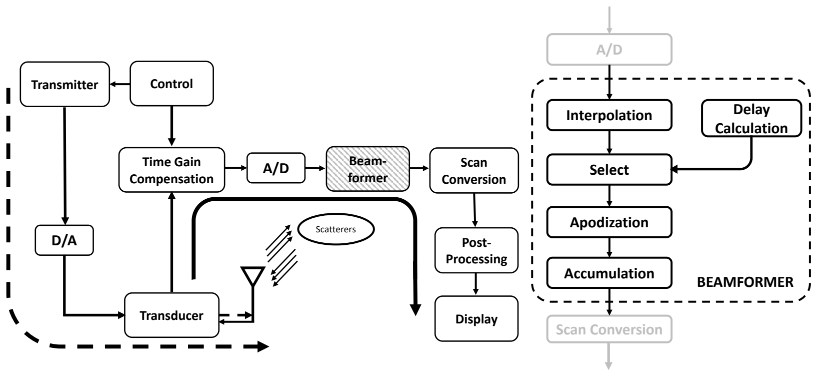


Figure 2.2: In the transmit mode, the waveforms are converted to analog form by a D/A converter, amplified, and fed to the transducer elements. In the receive mode, the received signals undergo time gain compensation to reduce the dynamic range and are then digitized by analog-to-digital converters (ADC). The digitized signals are beamformed to form each scanline. After scan conversion and post-processing techniques, the B-mode image is reconstructed.

the samples are multiplied by a weighting function. The weighting function accounts for any lateral distance between the receive transducer and the transducer aligned with the focal point in question, thereby improving focus and suppressing side lobes. Apodized samples are then summed to form the final pixel. A flowchart showing a generic time-delay beamforming algorithm is shown in Figure 2.2.

2.1.1 Transition from 2D to 3D

In 2D imaging, delays can be trivially precalculated and stored in lookup tables [34], but the scale of 3D imaging has typically precluded such precomputation due to the vast number of delays required. Moreover, the transition from 2D images to 3D volumes comes with an exponential increase in computational complexity and raw data bandwidth. Given a 2D receive aperture of 32×32 transducers and a 40 MHz sampling rate using 12-bit resolution, the raw data bandwidth is nearly 500 Gbps per second. This bandwidth is too high for a power-constrained platform, such as an ultrasound transducer probe, to transmit to an ultrasound machine for processing in real-time—therefore requiring beamforming to be tightly coupled to the transducers. This tight coupling complicates matters by requiring that beamforming occur within the ultrasound probe handle, in close contact with the patient's skin. If the probe handle dissipates more than ~5 W, including the beamformer, the resulting heat could burn the patient's skin [44].

2.1.2 Difficulties of High-Volume-Rate Ultrasound

Emerging 3D ultrasound applications in cardiac imaging [37], vector flow imaging [23, 55], and shear wave elastography [50] require extremely high volumetric image acquisition rates to track high frequency motion. In 2D this has been achieved by precalculating the round-trip delay values used in the beamforming process and storing them in look-up tables (LUTs) instead of calculating them on the fly, or by offloading the data to external, power-hungry machines to handle beamforming. However, due to the increased complexity of 3D systems, these are no longer options—there are too many delays to store in on-chip look-up tables, and the raw data bandwidth is too high to offload to a larger, more powerful machine. To make matters worse, these applications also require large imaging apertures to capture sufficient resolution in a large region of interest. The combination of a large receive aperture (32×32 elements), large imaging volume (32×32×$>$1500 focal points), substantial precision requirement (10-12 bit ADCs), and enormous volume rate ($>$10,000 volumes / sec) pose a daunting computational requirement—straight-forward delay-and-sum beamforming is intractable in real-time even with a large array of GPUs.

2.1.3 Reducing Computational Burden

To reduce the computational complexity of 3D ultrasound, researchers have proposed separable beamforming algorithms [64, 66, 63], which approximate traditional beamforming by splitting it into two stages. Separable algorithms perform beamforming consecutively along the two lateral image axes—first along the X-dimension, and then along the Y-dimension. The two-stage process reduces the 2D grid of contributing transducers (multiplicative complexity) for each voxel—a 3D pixel in the output volume—to two 1D arrays of contributing transducers (additive complexity), while sacrificing little image clarity. Despite such algo-

Table 2.1: 3D Plane-Wave System Parameters.

Property	Value
Speed of sound (tissue), m/s	1540
TX / RX Pitch, µm	192.5 / 385
Transmit aperture size, transducers	128×96
Receive aperture size, transducers	32×32
Beamforming aperture size, transducers	32×32
Number of scanlines per stage	32×32
Stage 1 ADC input length, points	3,077
Stage 1 scanline output length (M_z), points	2,089
Stage 2 scanline output length (M_z), points	1,679
Maximum imaging depth, cm	6
Center frequency, MHz	4
6 dB transducer bandwidth, MHz	2
ADC sampling rate, MHz	40

rithmic advances, however, naïve time-delay beamforming often remains impractical for 3D applications, as delay calculations involve complicated trigonometry.

2.1.4 Contributions

In this chapter, we build upon the work in [64, 66], which proposed a separable variant of the 3D plane-wave algorithm to reduce the computational complexity of plane-wave imaging. To make the algorithm's computational complexity even more tractable, and therefore more amenable to hardware implementation, we present Delay Compression. Delay Compression is a novel refinement of the 2D plane-wave imaging algorithm that enables drastic reduction in the number of unique round-trip computations per volume. We then extend this reduction to 3D for both non-separable and separable variants, demonstrating that this approach is not limited to the 2D case. Our time-delay decomposition dramatically decreases hardware complexity, increases performance, and lowers power requirements. In our 3D non-separable system, Delay Compression achieves a 1024× reduction in the number of delays computed, while our 3D-Separable implementation results in an asymptotic reduction in the number of delays computed for the first and second stages—up to 1008× and 1006× reduction in the first and second stages when using the aperture dimensions in Table 2.1, respectively.

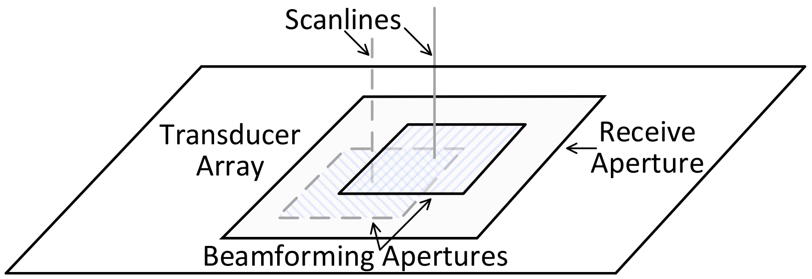


Figure 2.3: 3D plane-wave transmit and receive setup; overlapping beamforming apertures “step” across the receive aperture, each creating a single scanline.

With the number of unique delays reduced by over three orders of magnitude, precalculation and on-chip storage of delay constants for 3D systems becomes feasible for the first time, enabling new architectural approaches to high-volume-rate ultrasound imaging.

2.2 Background

Plane-wave imaging is an ultrasound variant that utilizes a planar transmission scheme. A large 2D array of transducers receive and sample the reflected signals, with each transducer outputting a *channel* of samples. The receive array is divided into smaller, overlapping 2D sub-arrays called beamforming apertures, as seen in Figure 2.3. The channels within each beamforming aperture contribute to voxels along a single *scanline*, or column of voxels in the final volume. In this paper, we assume a receive array of M_xM_y transducers, beamforming apertures of N_xN_y transducers, and a depth of M_z, resulting in a final volume size of $M_xM_yM_z$ voxels.

Managing delay constants poses a significant challenge in many 3D ultrasound algorithms, and plane-wave is no different. For separable plane-wave imaging with angled compounding [64], which can produce high quality images with plane-wave transmits, each firing angle requires a combined $M_xM_zN_xN_y + M_xM_yM_zN_y$ delay constants for the two beamforming stages. In Equations (2.1), (2.2), α and β are the elevational and lateral angles of the plane, respectively. (m_x, m_y, m_z) are the coordinates of each focal point, (n_x, n_y) are the coordinates of each receive transducer within the beamforming aperture, and (x_0, y_0, z_0)

is an arbitrary point on the plane at its origin; in this formulation, there is little symmetry between beamforming apertures due to the varying firing angle of the plane.

$$d_{tx} = (m_x - x_0)\sin\alpha\cos\beta + (m_y - y_0)\sin\alpha\sin\beta + (m_z - z_0)\cos\alpha \quad (2.1)$$

$$d_{rx} = \sqrt{(m_x - n_x)^2 + (m_y - n_y)^2 + m_z^2} \quad (2.2)$$

The high number of required delay constants is one of the principle difficulties of 3D ultrasound in general, as it prohibits large beamforming apertures, receive volumes, and depth combinations. Prior works [18, 19, 44, 46, 14, 15] have used specialized hardware to approximate delays or share small components of the computation between focal points to reduce the hardware requirements for beamforming, but all of these methods fall orders of magnitude short of the target 1,000+ volumes per second for plane-wave applications. Similarly, precomputing the delays and placing them in on-chip storage—or streaming them from off-chip storage—is also prohibitive due to the sheer quantity required (millions of delays per volume for large receive apertures); none of the prior approaches are sufficient for high-volume-rate beamforming.

2.3 Delay Compression

To combat the computational requirements of high-volume-rate beamforming, we propose *Delay Compression*. Delay Compression recognizes additional symmetry between the round-trip times for all focal points, thereby allowing us to drastically reduce the total number of computations required for each volume. At a high level, Delay Compression works by breaking the transmission component of the round-trip distance calculation into two pieces, which we call d_{tx1} and d_{tx2}. d_{tx1} is a set of unique distances from the plane-wave's origin to the first focal point along each scanline, where $m_z = 0$, hereafter referred to as the *base* of each scanline, while d_{tx2} is a set of distances calculated from when the plane is touching the base of a single scanline to each focal point along that scanline. One d_{tx1} distance corresponds to each scanline, and the set of d_{tx2} distances is shared between all scanlines—the d_{tx1} distance acts as an offset for the d_{tx2} distances within each beamforming aperture. To thoroughly explain Delay Compression, we will work our way through its function in plane-wave's 2D, 3D non-separable, and finally 3D-Separable [64, 66] variants.

2.3.0.1 Delay Compression in 2D

In its most simplistic form, plane-wave ultrasound fires a planar pulse directly into the target medium with an elevational angle of 0°, $M_xM_zN_x$ round-trip delays, and a final image size of M_xM_z. In this case the delay decomposition is straight-forward, as all of the transmission and reflection distances are equal—the distance from the plane's origin to the base of each scanline, d_{tx1}, is 0 for all scanlines. Equations (2.3), (2.4) demonstrate that the transmission distance does not vary across scanlines, so round-trip distance calculations can be shared—reducing the number of unique delays to M_zN_x.

$$d_{tx} = m_z - z_0 \tag{2.3}$$

$$d_{rx} = \sqrt{(m_x - n_x)^2 + m_z^2} \tag{2.4}$$

Figure 2.4 illustrates the symmetry. In the figure, the transmission distance is identical for each focal point that lies at the same depth m_z. With this in mind, distances a, b, and c need to be computed only once for all scanlines, resulting in a single shared set of distance values. The same is true for the reflection distances: since each beamforming aperture is centered on the scanline it produces, the distances from each focal point on the scanline to each transducer within the beamforming aperture are identical across apertures.

However, whereas a planar pulse with an elevational angle of 0° leads to straight-forward

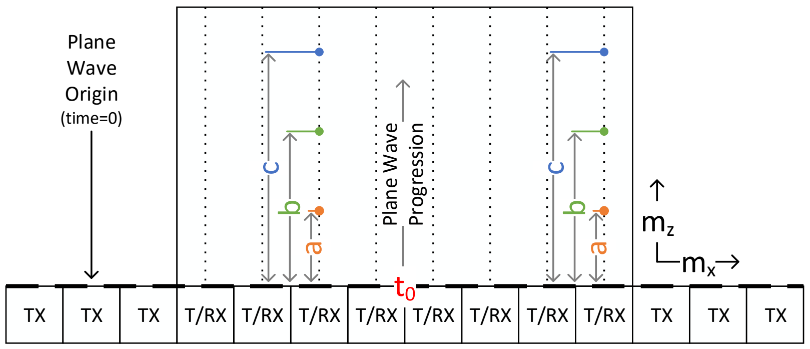


Figure 2.4: X-Z slice of the image space showing a 2D component of Delay Compression. Unique transmit distances a, b, and c represent the shared d_{tx} distances from the plane to focal points m_z along each scanline.

symmetry for delay reduction, there are cases, such as improving image quality through coherent compounding, which require transmits at varying elevational angles—resulting in $M_xM_zN_x$ delay calculations. Equations (2.5), (2.6) show how the transmission and reflection distances are calculated for planar transmits with non-0° elevation angle α.

$$d_{tx} = (m_x - x_0)\sin\alpha + (m_z - z_0)\cos\alpha \tag{2.5}$$

$$d_{rx} = \sqrt{(m_x - n_x)^2 + m_z^2} \tag{2.6}$$

For angled transmits, the transmit distance for each focal point at the same m_z depth depends on the scanline to which the focal point belongs. To recover the symmetry available in the 0° case, we use Delay Compression to decompose the transmission distance into two components, which we call d_{tx1} and d_{tx2}. d_{tx1} addresses the angle dependency by capturing the unique distance from the plane-wave's origin to the base of each scanline, from which the distances to each focal point are again equal across scanlines. By setting the value m_z in Equation (2.5) to 0, we find the distance from the plane-wave's origin to the first focal point in each scanline. We then set m_x equal to x_0 and z_0 equal to 0, essentially transforming the origin of the coordinate system to the base of the scanline. The 2D variant of the transmit distance decomposition can be seen in Equations (2.7), (2.8).

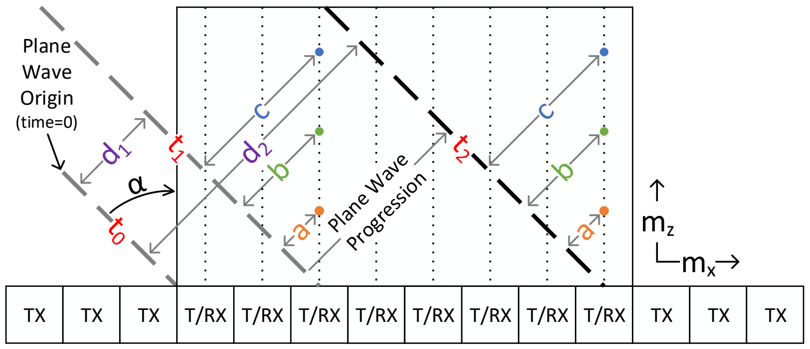


Figure 2.5: X-Z slice of the image space showing a 2D component of Delay Compression. Unique d_{tx1} distances are shown in d_1 and d_2, while a, b, and c represent the shared d_{tx2} distances from the plane to focal points m_z along each scanline.

$$d_{tx1} = (m_x - x_0)\sin\alpha - z_0\cos\alpha \tag{2.7}$$

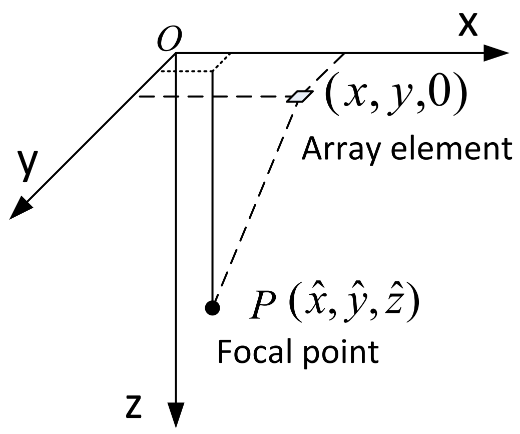
$$d_{tx2} = m_z\cos\alpha \tag{2.8}$$

Figure 2.5 again illustrates our delay decomposition, this time with an elevational angle of α. In this example, d_{tx1} distances d_1 and d_2 vary, but the d_{tx2} distances a, b, and c are constant across scanlines. By computing one d_{tx1} value for each scanline and a single set of d_{tx2} values, all of the unique transmission distances can be recomputed by simply adding the d_{tx1} values to the set of d_{tx2} values—reducing the number of unique delays to $M_x + M_zN_x$.

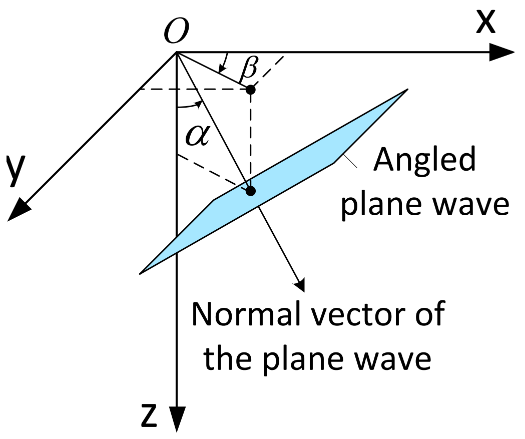
2.3.0.2 Delay Compression in 3D

3D plane-wave ultrasound requires drastically more round-trip calculations per volume than the 2D case. The coordinate system used for 3D plane-wave ultrasound includes both an elevational angle α and lateral angle β, as shown in Figure 2.6. This additional angle results in the number of delay calculations increasing to $M_xM_yM_zN_xN_y$.

3D Non-Separable. We implement Delay Compression in the 3D non-separable plane-wave algorithm much like 2D, decomposing the distance calculation into a base component and a distance along the scanline, sharing the latter across all scanlines. First, we again set m_z to 0 in order to calculate the distance d_{tx1}. We then set m_x to x_0, m_y to y_0, and m_z to



(a) Coordinate system



(b) Firing plane angles

Figure 2.6: Coordinate system and angle definition of 3D plane-wave system with coherent compounding.

0—translating the coordinate system to the base of the scanline and facilitating calculation of d_{tx2}. This breakdown can be seen in Equations (2.9), (2.10).

$$d_{tx1} = (m_x - x_0)\sin\alpha\cos\beta + (m_y - y_0)\sin\alpha\sin\beta - z_0\cos\alpha \tag{2.9}$$

$$d_{tx2} = m_z\cos\alpha \tag{2.10}$$

With this decomposition, the unique delays are reduced to $M_xM_y + M_zN_xN_y$—nearly a 1024x reduction under our system parameters.

3D-Separable. Separable 3D plane-wave imaging [66, 64] is a technique designed to reduce calculations by splitting beamforming into two sequential steps—first performing beamforming along the X-axis, and then along the Y-axis. The original implementation of separable 3D plane-wave imaging entails a computational complexity of $M_xM_zN_xN_y$ for the first stage and $M_xM_yM_zN_y$ for the second stage. Although this reduction is already significant, Delay Compression can further reduce the number of required calculations to only $M_xM_y + M_zN_x$ for the first stage and $M_xM_y + M_zN_y$ for the second stage. By using 3D-Separable plane-wave imaging and Delay Compression in tandem, 3D ultrasound complexity is reduced to a point where on-chip integration is achievable without compromising image

Table 2.2: Delay Calculations Required — per angle.

2D Flat	53,728
2D Angled	1,719,296
2D Angled Compressed	53,760
3D	1,760,559,104
3D Compressed	1,720,320
3D-Separable (stage 1)	68,452,352
3D-Separable (stage 2)	55,017,472
3D-Separable Compressed (stage 1)	67,872
3D-Separable Compressed (stage 2)	54,752

quality or exceeding the power requirements of the ultrasound probe handle.

Delay Compression drastically reduces the number of delay calculations required in 2D, 3D, and 3D-Separable variants of the plane-wave algorithm. By reducing the number of calculations, far less work is needed in order to reconstruct images or volumes—directly affecting the hardware complexity and power requirements for the reconstruction process. The exact number of delay calculations required by each variant when using our system parameters is shown in Table 2.2.

2.4 Evaluation

2.4.1 Image Quality

We verify Delay Compression image quality using visual comparison of 2D volume slices and the calculated Contrast-to-Noise Ratio (CNR) of simulated cyst phantoms. CNR is a standard metric for comparing ultrasound image quality [41, 54, 5] where the contrast resolution of the echoic (tissue) and anechoic (cyst) regions are compared under the effects of clutter and receiver noise (outside interference that produces image artifacts); higher CNR values indicate better image quality. The purpose of this validation is to confirm that Delay Compression's decomposition does not degrade image quality relative to that of traditional double-precision floating-point calculations.

Table 2.3: Anechoic Cyst CNR — 13-angle compounding.

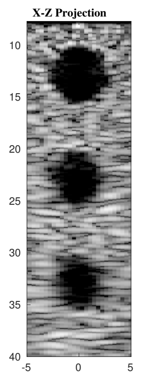
3D Implementation	Cyst 1	Cyst 2	Cyst 3
Non-Separable (doubles)	3.0916	3.1008	2.5766
Non-Separable Compressed (doubles)	3.0916	3.1008	2.5766
Separable (doubles)	3.0687	3.1141	2.5646
Separable Compressed (doubles)	3.0466	3.1006	2.6017

To validate that Delay Compression does not reduce image quality, we recreate the simulations described by Yang et al. [64] with our new Delay Compression method in Field II [20, 22]. Figure 2.7 and Table 2.3 show the results of simulations with both non-separable and separable beamforming algorithms with anechoic cysts at depths of 13 mm, 23 mm, and 33 mm to demonstrate image quality when using the different plane-wave algorithm variants. Non-separable beamforming reconstructs the volume in a single step, where each voxel has $32 * 32 = 1024$ contributing samples, whereas separable beamforming reconstructs the volumes in two sequential steps, with $32 + 32 = 64$ contributing samples per voxel. The system parameters used for these simulations are given in Table 2.1; the firing angles are $(\alpha, \beta) \in \{(0°,0°),(3°,0°),(6°,0°),(9°,0°),(3°,90°),(6°,90°),(9°,90°),(3°,180°),$ $(6°,180°),(9°,180°),(3°,270°),(6°,270°),(9°,270°)\}$.

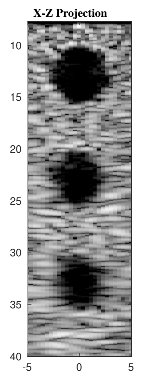
As seen in Figure 2.8, Delay Compression reduces calculation requirements by over 1000× for all methods. In terms of per-firing-angle storage or bandwidth savings (assuming 14-bit delay values), Delay Compression saves ~2.9 MB for 2D, ~3 GB for 3D non-separable, and ~216 MB for 3D-Separable implementations under our system parameters. This drastic reduction enables on-chip storage of delay constants for 3D applications for the first time, allowing new architectures and algorithmic approaches to be considered.

2.5 Conclusion

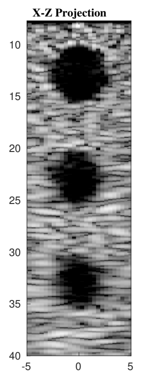
In this chapter, we proposed Delay Compression, a novel delay decomposition for plane-wave ultrasound that drastically reduces the number of delay calculations required



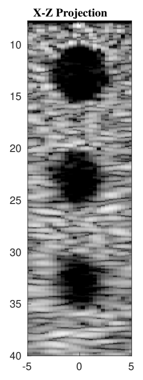
(a) Non-separable (doubles); 1,760,559,104 delay constants per angle



(b) Non-separable Delay Compression (doubles); 1,721,344 delays per angle



(c) Separable (doubles); 123,469,824 delay constants per angle



(d) Separable Delay Compression (doubles); 122,624 delay constants per angle

Figure 2.7: 2D slices of simulated 3D cyst phantoms using non-separable and separable plane-wave beamforming with 13-angle coherent compounding.

for the 2D, 3D, and 3D-Separable variants of the plane-wave algorithm. By splitting the planar transmit component of the round-trip distance calculation into a per-scanline offset and sharing the second component between scanlines, Delay Compression offers reductions in the number of unique delay constants required of up to 1024× when using our system parameters. The computational complexity reductions offered by Delay Compression enable pre-computation of the delays required by 3D systems for the first time, opening the door to new architectural designs and algorithmic innovations.

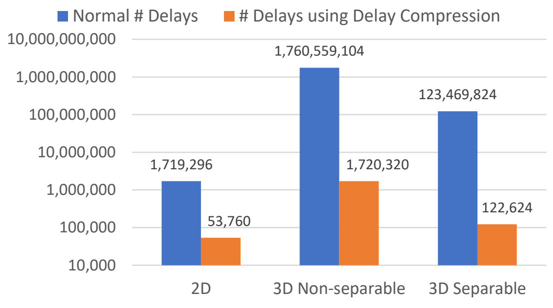


Figure 2.8: Delay requirements across plane-wave algorithm variants. M_z is 1,679 points for both 2D and 3D non-separable, while M_z for 3D-Separable has 2,089 points for the first stage and 1,679 points for the second stage.

CHAPTER III

TETRIS: A Streaming Accelerator for Physics-Limited 3D Plane-Wave Ultrasound Imaging

3.1 Introduction

3D ultrasound is an imaging modality that has seen increased use in medical applications due to its low risk potential compared to X-ray and MRI [45]. Emerging 3D ultrasound applications in cardiac imaging [37], vector flow imaging [23, 55], and shear wave elastography [50] require extremely high volumetric image acquisition rates to track high frequency motion. These applications also require large imaging apertures to capture sufficient resolution in a large region of interest. Worse, the combination of a large receive aperture (32×32 elements), large imaging volume (32×32×~1680 focal points), substantial precision requirement (10-12 bit ADCs), and enormous volume rate (up to 13,020 volumes per second) poses a daunting computational requirement—straight-forward delay-and-sum beamforming is intractable even with a rack full of GPUs. The transducers' raw data rate is so high (~500 Gbps) that beamforming must be tightly coupled to the transducers. However, this tight coupling implies that beamforming must occur in the ultrasound probe handle, in close contact with the patient's skin. If the beamforming system dissipates more than approximately 5 W, it could burn the patient [44].

The work described in this section was published in DAC 2019 [60] and IEEE TC 2020 [59].

Because beamforming—identifying the samples within each receive signal that correspond to each focal point—is the most computationally expensive aspect of 3D ultrasound [44], substantial work has been done on algorithms and computational architectures to make it tractable [65, 66]. Straight-forward time-delay beamforming calculates a time delay for the propagation path from each transmit element to each focal point to each receive element. For a target volume of size $M_xM_yM_z$ focal points and a receive array of N_xN_y transducers, the number of required delays, and the associated computational complexity, is $M_xM_yM_zN_xN_y$. To reduce computational complexity, researchers have proposed separable beamforming algorithms [64, 66, 63], which approximate traditional beamforming by splitting it into two stages, beamforming consecutively along the two lateral image axes. The two-stage process reduces the 2D grid of contributing transducers for each voxel (multiplicative complexity) to two 1D arrays of contributing transducers (additive complexity), while sacrificing little image clarity.

3.1.1 3D Ultrasound in Custom Hardware

Despite such algorithmic advances, however, naïve time-delay beamforming remains impractical, as delay calculations involve complex trigonometry. In 2D imaging, delays can be trivially stored in lookup tables [34], but the scale of 3D imaging has typically precluded such precomputation due to the vast number of delays required. Prior works have proposed a variety of algorithms [66, 64] and custom hardware architectures [44, 14, 19, 18] to reduce the computational burden of the delay calculation. Sonic Millip3De [44] used a multiplexed 128×96 transducer array, sampling 1024 elements per firing, and approximated the delay values using a piecewise function. The key to Sonic Millip3De's approach was that the delta function between adjacent focal points formed a smooth curve, which could be approximated accurately in hardware using only additions. The piecewise approximation function, paired with an iterative approach to image reconstruction, produced up to 32 volumes per second while consuming only 15 W of power [63]. Hager et al. [14] designed Ekho, a beamformer which fully samples a 100×100 element array and reduces computational burden through a novel computational sharing technique. Somewhat similar to Sonic Millip3De's piecewise approximation, Ekho implements a hybrid square-root algorithm to share portions of the delay calculation between focal points, requiring only incremental computation for each successive round-trip calculation. With this approach, Ekho is able to achieve 15 volumes per second in a power budget of 30.3 W. An approach that uses a look-up table and approximates the delay calculations to reduce complexity was proposed for another 100×100 array in [19, 18], relying on a piecewise approximation of the square root function. Using a combination of on-the-fly approximation and table-based steering, Ibrahim et al. achieve a rate of 15-30 volumes per second using a state-of-the-art Field Programmable Gate Array (FPGA). Unfortunately, while all of these systems demonstrated novel approaches to low-power distance calculation, all fall orders of magnitude short in achieving the 1,000+ volumes per second target for cardiac, vector flow imaging, and shear wave elastography applications.

3.1.2 Contributions

In this chapter, we describe TETRIS, a digital beamforming accelerator for high-volume-rate 3D plane-wave imaging. The TETRIS architecture is built around two primary components: First, TETRIS implements Delay Compression, our novel delay decomposition scheme that builds upon prior separable beamforming [64, 66] to exploit additional symmetries that arise in plane-wave imaging. With the number of unique delays reduced by over three orders of magnitude through Delay Compression, precalculation and on-chip storage of delay constants for 3D systems became feasible for the first time. In contrast to the other works, where delays were approximated using on-the-fly calculations, TETRIS uses this decomposition to reconstruct image volumes in a streaming fashion with no delay calculation or approximation performed on-chip—drastically reducing power and hardware complexity requirements. TETRIS takes Delay Compression a step further by converting precomputed delays into a specialized bit vector encoding, enabling on-chip storage or streaming of the delay constants as needed. With no on-the-fly calculation of delays, TETRIS effectively eliminates the most computationally intensive aspect of prior beamforming accelerators.

Second, TETRIS incorporates a novel hardware functional unit, the TETRIS RESERVATION STATION (RS), which aligns time-delayed samples across receive channels for the beamsum operation. We name this unit after the popular 80s-era video game because its operation is conceptually similar to the game—as sample values "fall to the bottom" of a conceptual 2D grid, a delay-aligned horizontal row is "cleared" and sent for accumulation when the row is fully populated. The TETRIS RS removes the need for on-chip storage of the receive signal, enabling arbitrary imaging depth and facilitating volume rates that reach physics limits.

By joining the Delay Compression decomposition with the TETRIS RESERVATION STATION (RS) in specialized processing pipelines, TETRIS achieves physics-limited beamforming rates for a fixed sampling rate. This means that image reconstruction is not limited by computational power, but rather the round-trip acoustical propagation speed of sound. The theoretical volume rate for any imaging depth can be approximated by dividing the round-trip distance for a given imaging depth by the speed of sound, or using the equation $1/(depth * 2/c)$, where c is the speed of sound (1540 m/s in human tissue). Given a 3077-sample ADC channel data obtained for a 6 cm imaging depth, TETRIS reconstructs 13,000 volumes per second; shallower/deeper imaging results in higher/lower volume rates, respectively.

In summary, we contribute:

- A *bit vector encoding* for compressed delays, making on-chip storage of precomputed delays feasible.
- The TETRIS RS, a novel functional unit that buffers incomplete voxels for single-pass, depth-agnostic, physics-limited volume reconstruction.

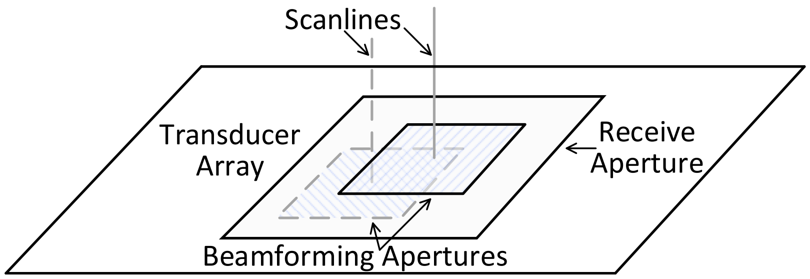


Figure 3.1: 3D plane-wave transmit and receive setup; overlapping beamforming apertures “step” across the receive aperture, each creating a single scanline.

- TETRIS, a streaming plane-wave beamforming accelerator for separable plane-wave ultrasound capable of physics-limited volume reconstruction with a power consumption of only 645 mW in 16 nm technology.

3.2 Background

Plane-wave ultrasound imaging uses a flat-plane (unfocused) transmission scheme to achieve high volume rates. A large 2D transducer array samples the reflected signals, with each receive *channel* digitizing the signal from one transducer. The large receive array is divided into smaller, overlapping 2D sub-arrays called *beamforming apertures* (see Figure 3.1). The channels within each beamforming aperture contribute to a single *scanline*—an axial column in the final volumetric image. The receive array comprises M_xM_y transducers, while the beamforming apertures include N_xN_y transducers; with a depth of M_z, the resulting image is $M_xM_yM_z$ voxels.

Managing delay constants poses a significant challenge in many 3D ultrasound algorithms, and plane-wave is no different. For separable plane-wave imaging with angled compounding [64], each firing angle requires a combined $M_xM_zN_xN_y$+$M_xM_yM_zN_y$ delay constants for the two beamforming stages. The exact delay calculations for each focal point and firing angle are given in Equations (3.1) and (3.2). In these equations, (m_x, m_y, m_z) are the coordinates of each focal point, (n_x, n_y) are the coordinates of each receive transducer within the beamforming aperture. The plane’s firing angle is defined by α—the angle between the z

axis and the normal vector, β—the angle between the x axis and the projection of the normal vector on the xy plane, and (x_0, y_0, z_0)—an arbitrary point on the plane at its origin. In this formulation, there is little symmetry between beamforming apertures due to the varying angle of the transmit plane. The transmit distance d_{tx} represents how far the plane must propagate to reach a point within the target volume, while d_{rx} represents the distance the reflection must travel to reach the transducers within the associated beamforming aperture.

$$d_{tx} = (m_x - x_0)\sin\alpha\cos\beta + (m_y - y_0)\sin\alpha\sin\beta + (m_z - z_0)\cos\alpha \tag{3.1}$$

$$d_{rx} = \sqrt{(m_x - n_x)^2 + (m_y - n_y)^2 + m_z^2} \tag{3.2}$$

The high number of required delay constants is one of the difficulties of 3D ultrasound in general, as it precludes combinations of large beamforming aperture, receive volume size, imaging depth, and volume formation rate. On-chip delay storage, or streaming them from off-chip storage, is prohibitive due to their sheer quantity, and the complexity of the trigonometric equations makes calculating them on the fly impractical due to power constraints.

3.2.1 Delay Compression

Prior works [44, 63, 66] attempt to reduce on-chip delay storage by approximating the delays using a piece-wise function and an associated on-chip delay calculation unit for each channel. However, this approach overlooks an opportunity for delay sharing between beamforming apertures in plane-wave imaging. For example, in the approach of Yang et al. [64], some components of the distance calculation do not actually depend on the original position or angle of the transmitted plane-wave. In fact, the reflection distance equation is identical for every beamforming aperture, and the transmit distance differs only up to the point where the plane-wave touches the first focal point along each scanline (where m_z is 0); after this point, the calculation is identical for each scanline in the volume. This

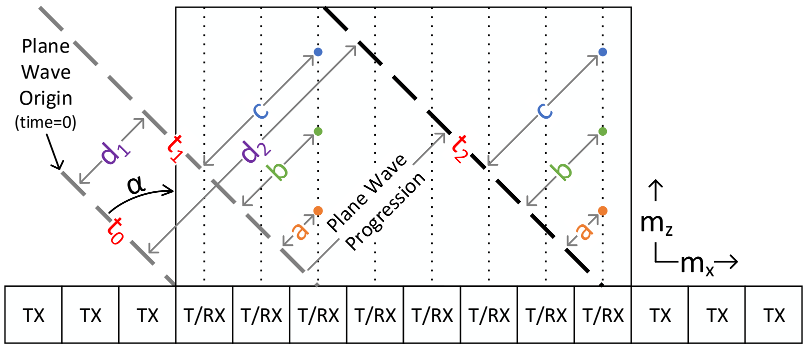


Figure 3.2: X-Z slice of the image space showing a 2D component of Delay Compression. Unique d_{tx1} distances are shown in d_1 and d_2, while a, b, and c represent the shared d_{tx2} distances from the plane to focal points m_z along each scanline.

decomposition, which we call Delay Compression, is visualized in Figure 3.2. The initial distance d_{tx1} is calculated by setting m_z to 0 in Equation 3.1, giving us the distance from the plane's origin to the first focal point in each scanline. We then calculate d_{tx2} by setting x_0, y_0, and z_0 equal to m_x, m_y, and 0 respectively, eliminating terms that depend on x and y:

$$d_{tx1} = (m_x - x_0)\sin\alpha\cos\beta + (m_y - y_0)\sin\alpha\sin\beta - z_0\cos\alpha \tag{3.3}$$

$$d_{tx2} = m_z\cos\alpha \tag{3.4}$$

By breaking the transmit distance into two components, d_{tx1} and d_{tx2}, we gain additional symmetry between beamforming apertures. Distance d_{tx1} measures from the plane's origin to the "base" of each scanline (where m_z is 0), while d_{tx2} measures from that point to each focal point along the line. This symmetry allows us to share a single d_{tx2} calculation across all beamforming apertures.

Whereas Delay Compression can be applied to both separable and non-separable plane-wave beamforming algorithms, we focus on the separable variant, as it significantly reduces the computational—and therefore the power—requirements of a beamforming hardware accelerator. For the separable algorithm, Delay Compression reduces the number of delays to M_xM_y+M_zN_x and $M_xM_yN_x$+M_zN_y for the first and second stages, respectively. For the imaging parameters we evaluate (see Table 3.1), Delay Compression yields savings of 1008×

Table 3.1: 3D Plane-Wave System Parameters.

Property	Value
Speed of sound (tissue), m/s	1540
TX / RX Pitch, µm	192.5 / 385
Transmit aperture size, transducers	128×96
Receive aperture size, transducers	32×32
Beamforming aperture size, transducers	32×32
Number of scanlines per stage	32×32
Stage 1 ADC input length, points	3,077
Stage 1 scanline output length (M_z), points	2,089
Stage 2 scanline output length (M_z), points	1,679
Maximum imaging depth, cm	6
Center frequency, MHz	4
6 dB transducer bandwidth, MHz	2
ADC sampling rate, MHz	40

and 636× for the first and second stages. The significant reduction makes it feasible to store the delay constants on-chip or stream them from memory without exorbitant bandwidth requirements, which in turn enables new approaches to beamforming hardware design.

3.3 TETRIS Microarchitecture

We next describe TETRIS, our 3D-Separable plane-wave beamforming accelerator, which uses Delay Compression and the TETRIS RESERVATION STATION (RS) to enable single-pass, depth-agnostic, physics-limited volume reconstruction in a fully streaming architecture.

Digital time-delay beamforming generally comprises five steps: (1) digitizing the input channels, (2) up-sampling received signals via interpolation, (3) selecting—through calculation of the round-trip delays—the samples corresponding to each voxel, (4) apodizing selected samples to improve focus and suppress side lobes, and (5) summing apodized samples across channels to obtain voxel intensity. Time-delay beamforming is amenable to massive parallelization across both channels and voxels. Nevertheless, the load-store interface of conventional data parallel architectures, such as GPUs, is a poor match for

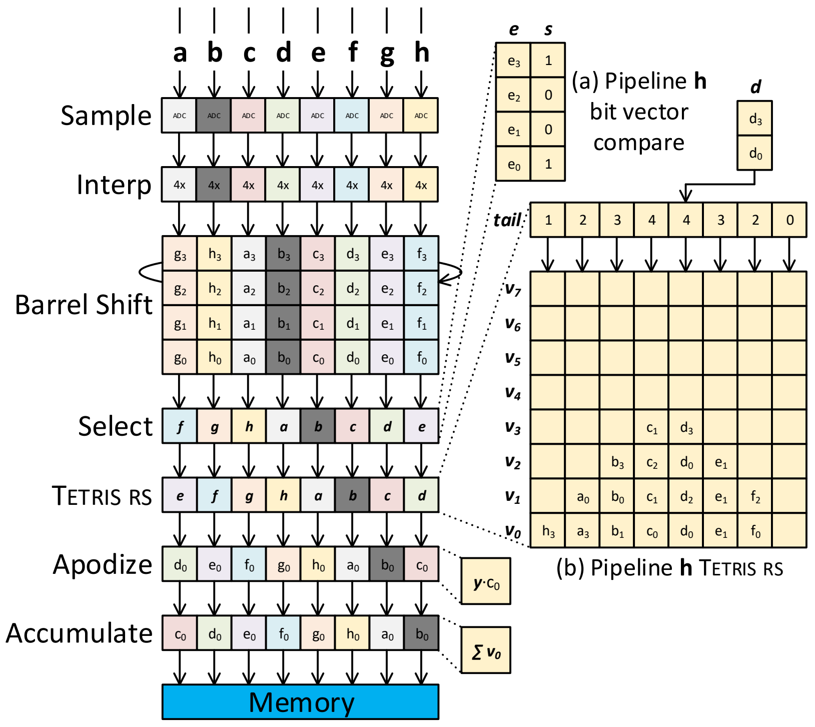


Figure 3.3: Scaled-down depiction of TETRIS with one slice of eight pipelines (a-h). New samples arrive at a fixed rate, while the core of the pipeline is clocked 8× faster, allowing samples to rotate across all pipelines via the barrel shift stage. Samples matching each focal point are selected according to the pre-computed bit-vector (a) in the select stage, and are then passed to the TETRIS RS (b) for temporary storage. As the "bottom" row fills, samples are sent to the apodization and accumulation modules to compute a single voxel; once filled, the bottom row is "dropped" and erased.

beamforming. If channel data are mapped to memory, each volume requires more than 100 million address calculation and load operations, rendering volume rates in excess of 1,000 per second impossible.

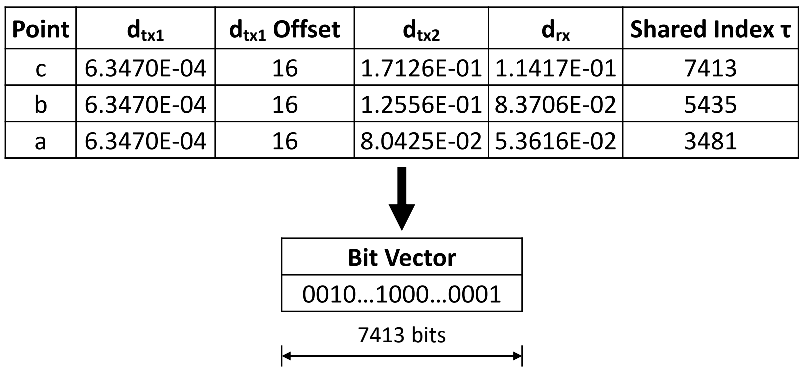
Prior 3D beamforming hardware [45] instead abstracts each incoming channel as a stream and selects samples that correspond to consecutive focal points along a scanline as the samples arrive from memory, without explicit loads or address calculation. Nevertheless, prior designs required large functional units to calculate the time-delay delta between consecutive focal points along a scanline. These delay estimation units dominate the hardware and limit performance to at most a few tens of volumes per second [63].

TETRIS achieves high volume rates within the tight power constraints of a hand-held system by exploiting Delay Compression to enable delay precomputation, eliminating the need for on-the-fly approximation. TETRIS comprises 1024 pipelines broken into 32 slices of 32 pipelines each, wherein each slice processes one row (or column) of transducer channels and each pipeline processes a single scanline; a simplified diagram depicting one slice with eight pipelines (labeled **a** through **h**) is shown in Figure 3.3. TETRIS exploits that, under separable beamforming, only the receive channels in the same row and slice contribute to a corresponding row of scanlines, while channels in different rows/slices do not interact—this dramatically simplifies the data sharing requirements between pipelines.

3.3.1 Pipeline Stages

The following subsections describe each of the TETRIS pipeline stages that comprise the five steps of digital time-delay beamforming; the chosen ordering of the stages ensures that no duplicate work occurs.

ADC Sampling, Interpolation, and Data Sharing. In each slice, incoming 12-bit samples arrive in each channel at 40 MHz from the analog front-end (top of Figure 3.3). The samples are then up-sampled via 4× linear interpolation to create 4-wide sample vectors; this up-sampling improves axial resolution while keeping power requirements of the ADCs from becoming prohibitive. Since samples from each vector potentially contribute to focal points in multiple scanlines, all 32 of these 4-sample vectors must be processed by all 32 pipelines. The "barrel shift" stage rotates the sample vectors laterally across pipelines within a slice, shifting by one position each cycle. The interpolator and subsequent stages are clocked 32× faster (1.28 GHz) than the ADC sampling rate (40 MHz), so each sample vector visits each pipeline once before the next input sample arrives from the ADCs. This 32× increase in processing frequency vs sampling frequency allows for beamforming aperture sizes of up to 32; i.e., 32 samples contribute to each output voxel, one from each channel. For smaller beamforming apertures, the clock speed multiplier can be lowered—for example, a beamforming aperture size of 20 only requires a clock speed of 800 megahertz in order to propagate the required data to each channel before the next sample arrives. Figure 3.3



Point	d_{tx1}	d_{tx1} Offset	d_{tx2}	d_{rx}	Shared Index τ
c	6.3470E-04	16	1.7126E-01	1.1417E-01	7413
b	6.3470E-04	16	1.2556E-01	8.3706E-02	5435
a	6.3470E-04	16	8.0425E-02	5.3616E-02	3481

Figure 3.4: Bit vector construction example: first the distances from the plane-wave's origin to the focal points are calculated and converted into an index into the sample stream. These indexes are then converted into the offset and bit vector, with a 1 in the bit vector corresponding to each index value and a 0 corresponding to all values between indexes. These points lie along a single scanline; other scanlines may have different d_{tx1} offsets depending on the firing angle.

shows an example state after six rotations, where vector **a** is in pipeline **c**.

Sample Selection using Bit Vectors. Each cycle, the "select" stage in each pipeline determines if any of the samples in the current vector contribute to focal points in its scanline. By using Delay Compression, we are able to compute the delays offline, with the d_{tx1} offsets being pre-loaded into each pipeline's select stage. While storage of the raw delay values is now possible, we recognize that storing direct indices is inefficient due to the need for designing the memory structure to fit the largest (i.e., the greatest number of bits) values, even though the difference from one value to another is very small. Given that a 4× interpolation factor was used, on average only one out of four samples will correspond to a focal point. Therefore, in order to further reduce hardware complexity, we store the shared delays (the sum of d_{tx2} and d_{rx}) in a bit vector, where a one indicates that a sample corresponds to a focal point, while a zero indicates the sample should be discarded (i.e., it falls between focal points). This representation removes the need to allocate space based on the largest possible value, and further reduces hardware complexity by removing the need for additional decrementers, turning the selection process into a simple Boolean check. Figure 3.4 shows an example of the shared bit vector and a d_{tx1} offset for a single receive channel. As incoming samples are presented to the select stage, the d_{tx1} offsets are

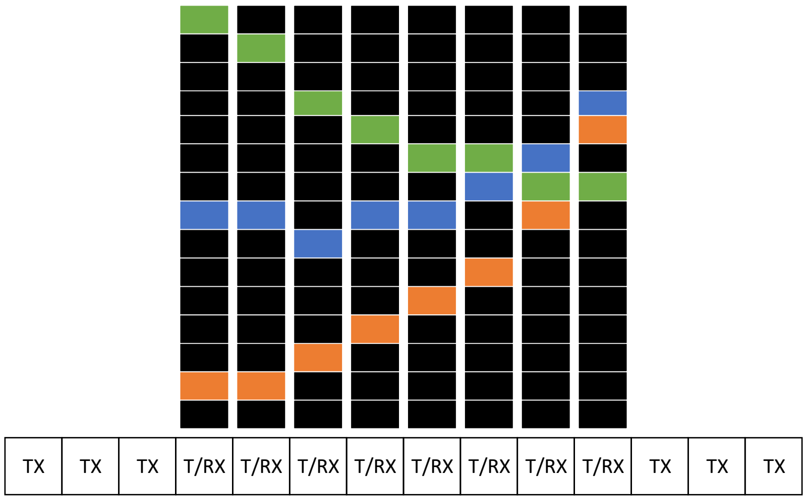


Figure 3.5: Example channel data; each colored curve represents a reflection from a unique focal point in the image space.

decremented; all samples are discarded while the offset is non-zero. Once the offsets reach zero, the selection process begins using the shared bit vector, which can be stored on-chip or streamed from off-chip and broadcast to the pipelines in parallel. The selection process for the **e** vector in pipeline **h** is depicted in Figure 3.3(a); bits that are set in the delay vector **s** indicate which corresponding samples should be selected.

TETRIS RESERVATION STATION (RS). Selected samples then flow into the TETRIS RS in each pipeline; the one for pipeline **h** is shown in Figure 3.3(b). Due to the curvature of the reflected signals, shown in Figure 3.5, the samples for each voxel still arrive offset in time, so a mechanism is needed to store them until summation can occur. As anywhere from zero to four samples may be selected each cycle (depending on depth), the temporary storage must allow for a variable number of writes per cycle without wasting space or incurring write port conflicts. The TETRIS RS is a 4-way banked register file logically arranged as a 2D grid, where columns correspond to input channels and rows are split round-robin across banks. There is a single write port that writes to up to four consecutive entries (i.e., across the four banks) in a single column each cycle, and a single read port that reads a single entry from the 2D array.

Conceptually, the unit acts as a 2D circular buffer, with one column allocated for each input data channel in the beamforming aperture. Arriving samples “fall down” the column corresponding to their receive channel to fill in the lowest-indexed empty cells (much like falling Tetris blocks). A tail pointer indicating the row index for the next write is maintained for each column, and a single column is written each cycle (up to 4 entries). Incoming samples arrive along with an index indicating from which input channel they are arriving; this index is used to determine to which column the samples should be written. Each row holds samples corresponding to a specific voxel, with a single global head pointer that points to the current row (or voxel) being processed. As the “bottom” row of the array fills, samples are read out, one per cycle, and sent to the apodization unit. Once the entire row has been read, the global head pointer is incremented, “dropping” the completed row (again, much like the game Tetris). Since each voxel will consist of the same number of samples, there is no need to reset the values in the dropped row—they will simply be overwritten by future values when the tail pointer for each column wraps back around.

Apodization and Accumulation. As samples are read from the TETRIS RS, they are apodized to reduce side lobes and improve focusing. Apodization involves multiplying each sample with a channel-specific weight; the set of channel-specific weights are pre-loaded into each apodization unit, and do not change unless the beamforming aperture size is reset. Although a 4× interpolation factor is used, only a single multiplier is required for each apodization unit since the TETRIS RS will only output a single sample per cycle.

After apodization, samples then pass to the accumulation unit, which sums samples from a particular TETRIS RS row. Once an entire row has been accumulated—as indicated by a counter reaching the configured beamforming aperture size—the final value is sent to either the second beamforming stage or to memory. Our approach minimizes apodization and accumulation operations, which account for a large portion of the system’s power.

3.3.2 TETRIS RS + Multiply-and-Accumulate.

One argument against using a 2D structure for buffering incoming samples is that, in the near-field (i.e., small imaging depths), the curvature of the reflected wave fronts arriving at the transducer array is high. As a consequence, the center columns in the 2D reservation station structure fill before the outer columns, creating an imbalance in the capacity requirement across columns. In situations where near-field imaging is required, the "height" of the 2D structure must ensure that the center columns do not overflow before the edges are complete—an inefficient use of storage since there may be many registers that are never used in the upper corners of the structure. Instead of a 2D register file structure, we can integrate the TETRIS RS, apodization unit, and accumulation unit into a single TETRIS Multiply-and-Accumulate (MAC) unit with a single storage array. Conceptually, the single column is still addressed as if it were a 2D structure, while the physical implementation comprises a linear register file. This implementation allows for far fewer storage slots since apodization and accumulation occur as the samples arrive, rather than staging them in temporary memory.

However, since we now must be able to apodize and accumulate up to four samples per cycle—as there is no way to buffer incoming samples from the select unit—four MAC units are required per pipeline. The required size of the new accumulation array depends on several factors, including the beamforming aperture size, the minimum imaging depth, and the location of the plane-wave's origin. In this work we use an accumulation array with 64 entries, which is sufficient to support the "height" of the reflection curvature for a minimum imaging depth of 6.5 mm.

3.3.3 System Integration

We next describe how the TETRIS beamforming accelerator can be integrated into a complete hand-held 3D ultrasound system. Like prior designs [44, 64], we rely on 3D die stacking to enable integration of analog, digital, and memory components with sufficient interconnect bandwidth; through-silicon vias (TSVs) allow for efficient transmission of the raw data between stacked layers. The complete beamforming system requires a 4- or 5-layer die-stacked structure comprising a layer of 128×96 capacitive micromachined ultrasonic transducers (CMUTs) on the face of the ultrasound probe, the ADCs and their associated amplifiers, one or two TETRIS beamforming accelerators, and a 12.5 MB eDRAM layer that acts as a buffer for completed volumes and facilitates compounding.

Since the separable plane-wave algorithm performs beamforming along the X- and Y-dimensions in sequence, TETRIS can be deployed in two different ways to address this requirement. In the first approach, the die stack includes a single TETRIS accelerator. In this configuration, the single TETRIS accelerator processes each volume twice, with the intermediate volume buffered in the eDRAM array until the first stage has completed and then passed back into the accelerator to handle the second stage. However, this approach halves the attainable volume acquisition rate, as the transducers must remain idle while the accelerator processes the intermediate volume from the eDRAM array. The second approach is to create a system which includes two TETRIS accelerators, rotated 90° with respect to each other to accommodate beamforming in both lateral dimensions in sequence. While having two TETRIS accelerators also doubles the power consumption of the beamforming layer, this design can achieve a physics-limited volume rate while still meeting the target power budget.

3.4 Evaluation

We first demonstrate why a hardware accelerator is needed for plane-wave ultrasound by reviewing beamforming performance on existing computational platforms. We then validate the TETRIS design in two parts. First, we evaluate the power and area requirements of a TETRIS system in an industrial 16 nm node, showing that we meet the 5 W power budget deemed safe for contact with human skin. Second, we confirm the image quality of the hardware design against ideal beamforming using Field II [21, 20, 22], a widely-used

Table 3.2: System Performance — volumes per second.

Beamforming Aperture:	20	25	32	Power*
TETRIS (2×)†	13,020	13,020	13,020	1.29 W
Nvidia Titan XP (doubles)	103.3	91.5	75.1	33.81 kW
Nvidia Titan XP (floats)	108.2	96.7	81.1	33.81 kW
TI C6678 DSP [29]	36.7	29.4	23.0	5.79 kW

Matlab extension for modeling ultrasound physics, and verify that JIGSAW's 12-bit fixed-point pipeline does not reduce image quality when compared to standard implementations.

3.4.1 Why a[nother] Specialized Accelerator?

To demonstrate why new approaches are needed to achieve physics—rather than computation—limited volume acquisition rates for 3D ultrasound, we briefly consider what it would take to perform beamforming on conventional computational platforms. The overall performance of the conventional platforms and the TETRIS system are compared in Table 3.2.

DSP. The TI C6678 is a state-of-the-art Digital Signal Processor (DSP) that is specifically marketed for medical imaging applications [52]. A study performed with the C6678 found that a single DSP can output a 4096-point scanline for a 512-transducer array in 740 μs when using a state-of-the-art algorithm and a 1.0 GHz clock frequency [29]. We scale this result to estimate the time per scanline for our separable plane-wave algorithm, estimating that the C6678 requires 23.58 μs for the first stage and 18.96 μs for the second stage, resulting in a total of 42.54 μs per scanline for our target system and imaging parameters (see Table 3.1). Using this back-of-the-envelope calculation, we estimate the C6678 will produce a single volume in 43.56 ms. To achieve 13,000 volumes per second—where the round-trip time of the transmitted signal is the limiting factor—requires the use of 568 DSPs, with a staggering 5.79 kW power consumption [51]. The need for explicit memory addressing and load instructions makes a DSP-based solution untenable for such volume rates.

GPU. To evaluate performance on high-end Graphics Processing Units (GPUs), we implement 3D plane-wave beamforming on an Nvidia Titan XP. Since there is no known method of offloading data to an external machine for high-volume-rate processing, we assume zero-cost instantaneous data transfer and consider only computation in our comparison. We implement our separable plane-wave algorithm with Delay Compression in CUDA to match the algorithm implemented by TETRIS as closely as possible, storing the precomputed delays in an in-memory lookup table. To improve efficiency, we use four separate GPU kernels: stage 1 interpolation, stage 1 beamforming, stage 2 interpolation, and stage 2 beamforming. In the interpolation kernels, each GPU thread performs 4x interpolation for a single sample, yielding 32×32×3077 threads in the first stage and 32×32×2089 threads in the second stage. The beamforming kernels compute a single voxel per thread, resulting in 32×32×2089 threads in the first stage and 32×32×1679 threads in the second stage.

Using this implementation, the Titan XP reconstructs a single volume in 12.34 ms using single-precision floating-point, yielding 81 volumes per second. However, matching the physics-limited volume rate of 13,000 per second still requires 161 Titan XPs, with a total power consumption of nearly 34 kW. TETRIS is drastically more efficient than the GPU because its streaming architecture does not require explicit address calculation and load/store instructions to access channel data, and because it operates using 12-bit fixed-point precision rather than 32- or 64-bit floating point.

While there has been prior work on hardware-accelerator ultrasound beamforming [44, 18, 66, 14, 34, 19], none of the prior accelerators target volume acquisition rates residing within the same order of magnitude as TETRIS.

3.4.2 Hardware Analysis

We implement TETRIS in SystemVerilog and synthesize it using an industrial 16 nm standard cell library and SRAM compiler. We report power for a 1.28 GHz operating frequency, which is fast enough to process all ADC data in real-time and achieve physics-limited volume rates for the full 32×32 receive aperture at any imaging depth; smaller apertures can be processed at lower frequencies (and therefore lower power).

Power & Area. Power and area results vary significantly based on whether one implements the TETRIS RS's 2D register file structure or the linear MAC array. When using the TETRIS RS with 32 rows and 32 columns, synthesis reports that the TETRIS beamformer requires approximately 780 mW of power and 11.9 mm^2 of area. In contrast, the optimized TETRIS RS+MAC implementation eliminates the 2D register file structure, the apodization unit, and the accumulate unit, replacing them with a four-wide MAC array with a 64-element linear array. This results in a power consumption of 645 mW and 11.3 mm^2 of area.

To understand the differences in power and area between the two designs, we discuss the hardware units required by each implementation. Whereas the MAC array requires four MAC units per pipeline, compared to the 2D structure's single unit, it needs a much smaller storage buffer. By eliminating the ~1.5 kB of SRAM used for each of the 1024 pipelines—which accounts for nearly 40% of the area and power in the 2D register file-based design—a 64-entry register-based MAC buffer and three additional MAC units per pipeline still achieves a net decrease in area and power.

Performance. Given a 1.28 GHz operating frequency, a 32×32 receive aperture, and 32×32 beamforming apertures (i.e., samples from 32 channels contribute to each voxel in each dimension), TETRIS can process all incoming data in a streaming fashion. By calculating the time it takes the sound wave to travel from the transducer array to the maximum imaging depth and back, we find that the TETRIS beamformer can reconstruct approximately 13,000 volumes per second when using our system parameters.

TETRIS's power efficiency can be found by calculating the theoretical maximum beamforming operations per second (BOPS), as defined in [14], and dividing by the beamformer's power consumption. Given a 1.28 GHz operating frequency, each of the 1024 channels in TETRIS can output one completed voxel every 32 cycles in steady state when using a 32×32-channel beamforming aperture. Due to our using a separable beamforming algorithm, only 64 beamforming operations are required to create each voxel—32 delay-and-sums

Table 3.3: Beamforming Operations per Second/Watt.

	BOPS	BOPS/W	Technology
TETRIS	2.62 T	2.03 T	16 nm
Ekho [14]	2.98 T	98.4 G	28 nm
Sonic Millip3De [44]	2.01 T	684.8 G	11 nm

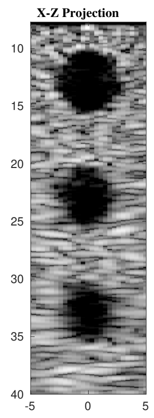
Table 3.4: Anechoic Cyst CNR — 13-angle compounding.

3D Implementation	Cyst 1	Cyst 2	Cyst 3
Non-Separable (doubles)	3.0916	3.1008	2.5766
Non-Separable Compressed (doubles)	3.0916	3.1008	2.5766
Separable (doubles)	3.0687	3.1141	2.5646
Separable Compressed (doubles)	3.0466	3.1006	2.6017
Separable Compressed (12-bit TETRIS)	3.0316	3.1027	2.5993

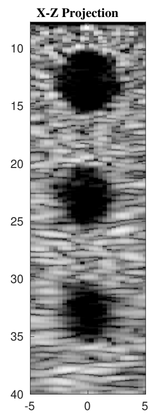
per stage. This leads to a BOPS value of 1.28G×1024×64/32, or 2.62 tera-BOPS. After dividing the tera-BOPS value by the two-TETRIS-layer power consumption of 1.29 W—645 mW per stage—we obtain a power efficiency value of 2.03 tera-beamforming operations per watt (TBOPS/W). Compared to Ekho [14] and Sonic Millip3De [44], the only other fully-integrated beamforming solutions, our 16 nm TETRIS beamformer is over 20× more efficient than Ekho, implemented in 28 nm technology, and nearly 3× more efficient than Sonic Millip3De, which reported scaled power estimates for an 11 nm technology. These results are outlined in Table 3.3.

3.4.3 Image Quality

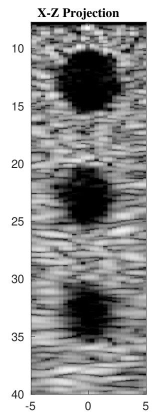
We verify TETRIS image quality using visual comparison of 2D volume slices and the calculated Contrast-to-Noise Ratio (CNR) of simulated cyst phantoms. CNR is a standard metric for comparing ultrasound image quality [41, 54, 5] where the contrast resolution of the echoic (tissue) and anechoic (cyst) regions are compared under the effects of clutter and receiver noise (outside interference that produces image artifacts); higher CNR values indicate improved image quality. The purpose of this validation is to confirm that TETRIS's 12-bit data path does not degrade image quality relative to double-precision floating-point



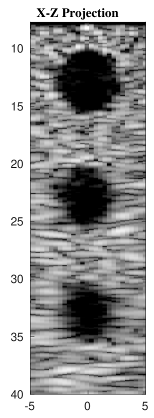
(a) Non-separable (doubles); 1,760,559,104 delay constants per angle



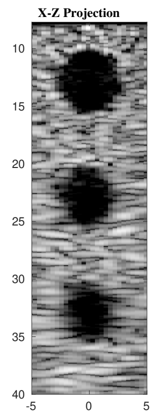
(b) Non-separable Delay Compression (doubles); 1,721,344 delays per angle



(c) Separable (doubles); 123,469,824 delay constants per angle



(d) Separable Delay Compression (doubles); 122,624 delay constants per angle



(e) Separable Delay Compression (12-bit TETRIS); 122,624 delay constants per angle

Figure 3.6: 2D slices of simulated 3D cyst phantoms using non-separable and separable plane-wave beamforming with 13-angle coherent compounding.

calculations.

Using our hardware simulator, we feed the raw receiver data into the sample stage of each pipeline and compare the output to the same Matlab baseline used for validating Delay Compression. As shown in Table 3.4 and Figure 3.6, the fixed-point hardware simulation produces images and CNR values indistinguishable from those using double-precision floating-point. These results demonstrate that the image quality degradation using our Delay Compression technique and fixed-point hardware implementation results in negligible

Table 3.5: Estimated Full-System Power — each column represents a different layer in the 3D stack.

	CMUTs (128×96)	ADCs (32×32)	TETRIS MAC #1	TETRIS MAC #2	eDRAM (12.5MB)	Total
Power	300 mW	143 mW	645 mW	645 mW	325 mW	2.1 W
Area	432 mm^2	4.8 mm^2	11.3 mm^2	11.3 mm^2	4.5 mm^2	N/A

changes when compared to ideal beamforming.

3.4.4 Full-System Power

We estimate full-system power using a combination of hardware synthesis and previously published results. The area and power for each layer is shown in Table 3.5. Our design assumes the same 128×96 transducer array as [44], with elements spaced λ apart, where λ is the wavelength in tissue, i.e. 385µm for a 4 MHz excitation frequency and 1,540 meters per second speed of sound (λ=c/f). While our receive aperture is only 32×32 elements, the larger transmit array avoids roll-off at the edges of the imaging volume when transmitting angled planes. We estimate ADC power using a recently published implementation in 28 nm technology [35], scaling the power linearly as we require only a 40 MHz sample rate rather than the published 100 MHz. We estimate the area and power requirements of the 12.5 megabyte eDRAM based on reported results for a 22 nm technology [16]; the eDRAM power is based off of scaling the design to account for the proper array size and bandwidth.

3.5 Conclusion

In this chapter, we have described TETRIS—the first beamforming accelerator for separable 3D plane-wave ultrasound capable of achieving physics-limited volume rates. TETRIS is enabled by Delay Compression, a novel approach to delay decomposition which enables precomputation of the 3D delay constants for the first time. TETRIS is paired with the TETRIS RS—conceptually arranged as a 2D register structure that sorts the incoming samples and sums relevant samples as they arrive, avoiding the need to store the received signal and allowing for a streaming design. Through synthesis of our RTL-level design in a 16 nm standard cell library, we show that TETRIS achieves single-stage, physics-limited volume rates with a power consumption of only ~645 mW. Furthermore, our system integration estimates indicate that TETRIS can be combined with the required supporting layers to form a full 3D ultrasound system with a power consumption of ~2 W, remaining well within the tight 5 W power constraint for safe contact with human skin.

CHAPTER IV

Slice-and-Dice: Rethinking Non-uniform FFT Interpolations in MRI for Modern Parallel Architectures

4.1 Introduction

One of the ten most influential signal processing algorithms of the 20th century [36], 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 [11, 25]—and specialized instructions and hardware units embedded in modern processors. However, the conventional FFT is applicable only to data that is uniformly sampled—and therefore unable to support computational imaging modalities such as magnetic resonance imaging (MRI) [28, 49, 1, 9, 38, 61, 12], computed tomography [39, 62], synthetic aperture radar [27, 17], and radio astronomy [56, 26] that use non-uniform sampling to enable reduced imaging scan time or irregular sensor placement.

To enable quick processing of an irregular data set, the Non-uniform FFT (NuFFT) extends the FFT to non-uniform sampling patterns [7]. The NuFFT is computed using a three-step process: (1) interpolation between non-uniform samples and a uniform grid,

The work described in this section was published in IPDPS 2021 [57].

hereafter referred to as “gridding” for simplicity, (2) apodization, or weighting of the uniform samples, 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 non-uniform interpolation step [47, 30, 32, 28], 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’s computation time using a representative 2D data set [40]. 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 NuFFT implementations. With the rise in real-time [10] and iterative image reconstruction techniques [6]—particularly in 3D, wherein millions of NuFFTs are iteratively performed to reconstruct a single volume—NuFFT performance is key to computing answers quickly and enabling emerging applications.

4.1.1 Algorithmic Optimizations

In an attempt to overcome these challenges, many optimized variants of gridding have been proposed to improve performance of the NuFFT [1, 24, 47, 38, 32, 31, 12, 3, 2]. 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 [24, 43]. These tiles are configured such that they are small enough to fit in an on-chip cache or memory [32, 3, 2]. Binning sequentially processes tile–bin pairs, improving memory bandwidth use by reducing the number of cache evictions caused by spatially the diverse reads and writes. Obeid et al. build upon this approach in [38], recognizing that in many sampling patterns the density of samples varies across the grid, leaving unused entries in the uniformly-sized bins associated with the lower-density regions. To eliminate this wasted space, they implement dynamically-sized bins to accommodate varying sample density without wasting memory. An alternative approach using matrix operations is found in the Michigan Image Reconstruction Toolbox (MIRT) [8]. MIRT performs fast and accurate NuFFT operations in Matlab using a collection of open source algorithms for MRI reconstruction and related imaging problems. Widely used in the imaging community, MIRT relies on Matlab's optimized matrix processing and compiled executables to efficiently perform gridding using both interpolation table and sparse matrix implementations. However, despite these algorithmic optimizations enabling gridding performance improvements ranging up to 40× when using commodity hardware accelerators such as GPUs [47, 12] and FPGAs [32, 31, 3, 2], they only partially mitigate the bottleneck—NuFFT computation time remains dominated by gridding.

4.1.2 Contributions

The prior works suffer from three distinct disadvantages: (1) they require a pre-processing step to sort the non-uniform samples, (2) they process each non-uniform sample up to four times when the interpolation kernel overlaps multiple tiles, and (3) they severely limit parallelism by only processing a single tile–bin pair at a time. As an alternative, in this chapter we present Slice-and-Dice, a novel approach to NuFFT gridding that completely removes the binning step in a manner that maps efficiently to massively-parallel hardware, such as a GPU or ASIC design. Slice-and-Dice departs from the conventional approach by breaking the target grid into tiles and stacking them to create *dice*. Rather than processing a single tile at a time, as found in binning, Slice-and-Dice processes contributions from an input sample to all tiles in the dice in parallel—conceptually “binning on-the-fly”—in a manner that mitigates the memory bandwidth utilization problems encountered by prior works and increasing parallelism. By using a stacked tile model and operating on all tiles in parallel, Slice-and-Dice achieves a 1600× reduction in nonuniform-to-uniform boundary checks using our system parameters. When implemented on a GPU, Slice-and-Dice achieves average gridding speedups of over 342× and 20× when compared to the CPU baseline [8] and state-of-the-art GPU implementation [12], respectively. When run as part of the complete NuFFT algorithm, our GPU gridding implementation results in end-to-end speedups of over 85× vs the CPU baseline and 6× the prior work, with gridding and the FFT contributing approximately equal computation time.

4.2 Background

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, precluding the use of the FFT. Instead, applications util zing irregularly sampled data 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.

4.2.1 Non-uniform Discrete Fourier Transform

A generalization of the forward Discrete Fourier Transform, the Non-uniform Discrete Fourier Transform (NuDFT) allows for computation over non-uniform input data. Following the notation found in [47, 30], 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

$$f_j = \sum_{\boldsymbol{k}\in\{0,\ldots,N-1\}^d} \hat{f}_{\boldsymbol{k}}\, \mathrm{e}^{-2\pi i \boldsymbol{k}\cdot\boldsymbol{x}_j}, \quad j = 0,\ldots,M-1 \tag{4.1}$$

The adjoint NuDFT is similarly defined as

$$\hat{h}_{\boldsymbol{k}} = \sum_{j=0}^{M-1} \hat{f}_j\, \mathrm{e}^{2\pi i \boldsymbol{k}\cdot\boldsymbol{x}_j}, \quad \boldsymbol{k} \in \{0,\ldots,N-1\}^d \tag{4.2}$$

Equations (4.1) and (4.2) can also be written as matrix-vector products, as shown in the following equations:

$$\boldsymbol{f} = \boldsymbol{A}\hat{\boldsymbol{f}} \tag{4.3}$$

$$\hat{\boldsymbol{h}} = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{f}, \tag{4.4}$$

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.

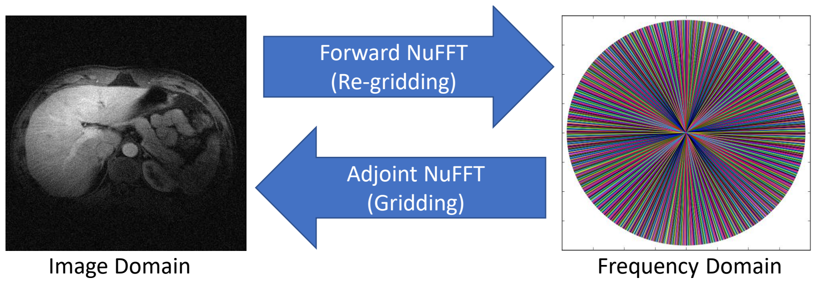


Figure 4.1: Each NuFFT variant comprises three steps. Forward: (1) pre-apodization, (2) FFT, (3) regridding. Adjoint: (1) gridding, (2) FFT, (3) de-apodization. Image data from [40].

4.2.2 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) non-uniform interpolation, (2) an FFT, and (3) apodization (i.e., amplitude weighting), 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, with the forward and adjoint NuFFTs both staples in image reconstruction. As shown in Figure 4.1, the forward NuFFT transforms image data to the frequency domain, while the adjoint NuFFT transforms frequency data to the image domain. The non-uniform interpolations dominate the time required to compute the NuFFT, accounting for upwards of 99.6% of the computation time [8].

Often called gridding in the adjoint NuFFT and regridding in the forward NuFFT, this interpolation step—visualized in Figure 4.2—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

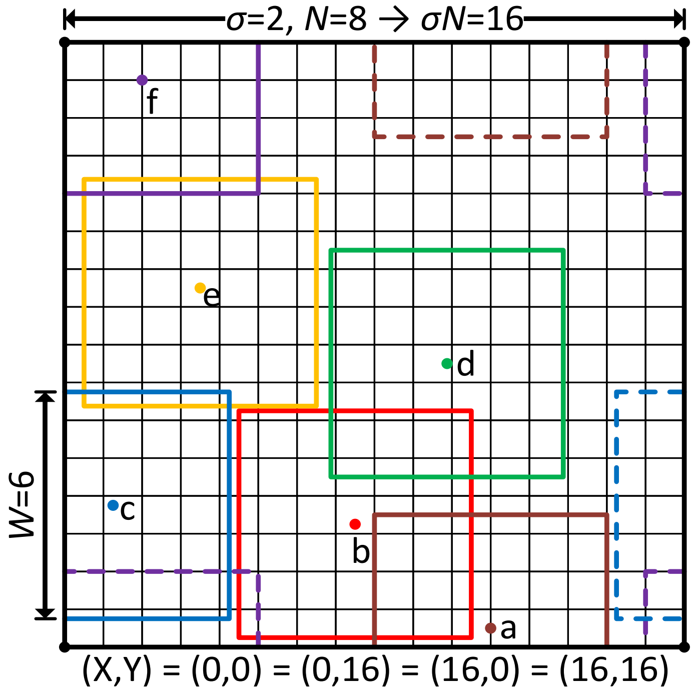


Figure 4.2: Uniform grid (flattened torus) with $d = 2$ dimensions, $M = 6$ input samples, base grid dimension $N = 8$, oversampling factor $\sigma = 2$, and interpolation kernel width $W = 6$.

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 is possible even for hardware with limited on-chip memory, reducing the amount of online computation required for each interpolation operation; these weights are commonly stored in a look-up table (LUT).

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 4.2, where the interpolation kernels of samples a, c, and f wrap to other sides of the grid, requiring complicated circular boundary checks to determine which uniform points are 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's interpolation accuracy, an oversampling factor σ, set to two in Figure 4.2, is multiplied by the uniform and non-uniform coordinates prior to the interpolation process. 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 σN FFT along each dimension, and (2) increasing gridding memory requirements as N or d grow. To alleviate these affects, 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 the computational complexity and causes the NuFFT to be even further dominated by the interpolation operation.

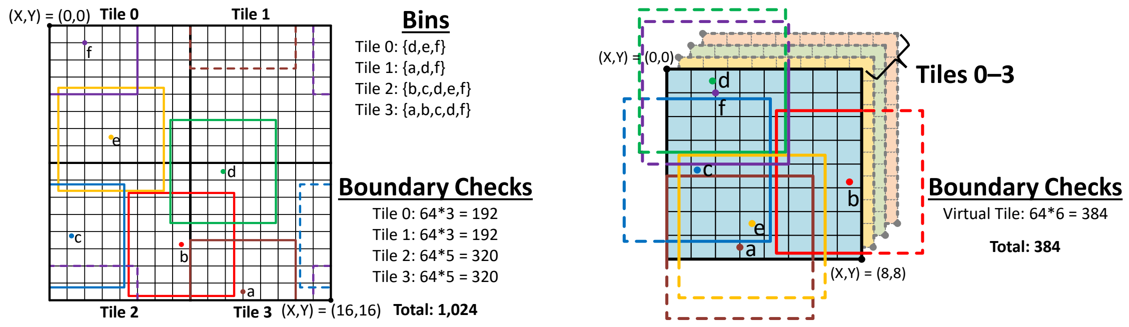
4.2.3 Traditional Gridding

As the dominant computational component of the NuFFT, optimizing the interpolation step of the NuFFT has been a focus of many previous works [1, 3, 2, 8, 9, 12, 28, 30, 32, 31, 38, 47]. 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; the forward NuFFT requires the inverse of this operation. 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 and system parameters. In particular, gridding time depends on the oversampling factor σ and the interpolation kernel width W. The interpolation kernel width W—commonly set to four or six—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 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) Binning (b) Slice-and-Dice

Figure 4.3: 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 location of the uniform point affected (i.e., "depth" in the stack). Comparatively, binning requires 3× more boundary checks.

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 4.3a, 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 boundary checks between every sample and every uniform point in the entire grid, samples in each bin must only be checked against each of 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 4.3a, 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 associated with each bin 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 4.3a, 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 may try to parallelize across titles to leverage massive Thread-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 result, neither CPUs nor GPUs are able to entirely hide the memory latency via ILP/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.

4.3 Slice-and-Dice Design

We next introduce the Slice-and-Dice NuFFT gridding acceleration model 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 synchronization-free processing of parallel threads or pipelines.

4.3.1 Rethinking Binning

Most recent NuFFT acceleration implementations use binning to improve 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 overlaps with 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) non-trivial wasted computation, and (3) additional memory overhead and 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-

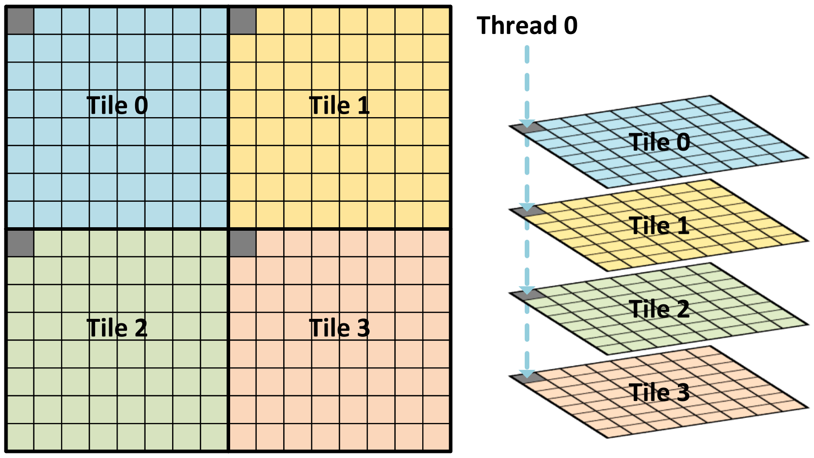
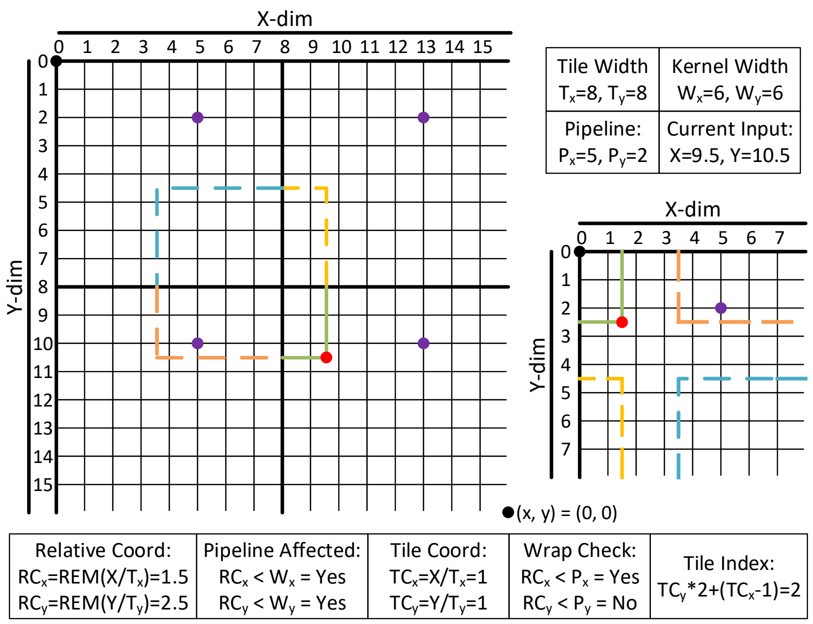


Figure 4.4: Slice-and-Dice virtually stacks tiles on top of each other, with a single thread processing all samples that affect a given position in any tile (i.e., a column in the stack). This drastically reduces the number of all-to-all boundary checks, as each sample must only be checked against each column.

based, output-driven parallel model and a unique decomposition of the sample coordinates. As shown in Figure 4.4, 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 4.4 will process the uniform grid points at relative position $(x,y) = (0,0)$ in each of the four virtual tiles. Coupled with the constraint $W \leq T$, this layout guarantees that each sample will only affect a single point in any column.

Efficiently handling boundary checks in the stacked tile model is enabled by a two-part decomposition of the input samples' coordinates, as shown in Figure 4.5. For coordinates ranging from $[0 : N)$ in each dimension, we perform the 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



Tile Width	Kernel Width
T_x=8, T_y=8	W_x=6, W_y=6
Pipeline:	Current Input:
P_x=5, P_y=2	X=9.5, Y=10.5

Relative Coord:	Pipeline Affected:	Tile Coord:	Wrap Check:	Tile Index:
RC_x=REM(X/T_x)=1.5	RC_x < W_x = Yes	TC_x=X/T_x=1	RC_x < P_x = Yes	TC_y*2+(TC_x-1)=2
RC_y=REM(Y/T_y)=2.5	RC_y < W_y = Yes	TC_y=Y/T_y=1	RC_y < P_y = No	

Figure 4.5: Slice-and-Dice in action. The thread assigned to handle uniform points with relative positions $(x, y) = (5, 2)$ is affected by an input sample with coordinates $(9.5, 10.5)$ if both relative coordinates are less than the interpolation kernel width. The thread is affected in tile $(0, 1)$ but the sample's coordinate is in tile $(1, 1)$, causing a wrap in the X dimension. If a wrap occurs, the corresponding tile coordinate is offset.

input sample affects one of 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 uniform grid—resulting in MT^d total checks. Since points are processed sequentially in 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 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 output-parallel implementation.

4.3.2 Simplifying Boundary Checks

Because the uniform grid is a torus in the frequency domain, care must be taken in all implementations to handle situations wherein the interpolation kernel "wraps" around to the other edge of the target grid. The stacked virtual tile structure of Slice-and-Dice exacerbates this issue, as wrapping now occurs along the edge of each virtual tile rather than only at the edges of the grid as a whole. Slice-and-Dice simplifies the handling of wraps by offsetting the input coordinates by $W/2$, which eliminates wraps to the right. The offset effectively halves the computational requirements of the boundary check itself, as it eliminates the need to calculate and compare bidirectional column-to-sample distance in each dimension. Rather than checking if the distance between a column and an input sample is less than $\pm W/2$, Slice-and-Dice only checks if the distance from the column to the sample in the positive direction is less than W. In Figure 4.5—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.

4.4 Evaluation

To evaluate the Slice-and-Dice model, we implement Slice-and-Dice in CPU and GPU variants with virtual tiles of dimension 8×8. We employ the Michigan Image Reconstruction Toolbox (MIRT) [8]—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 MIRT is on par with that of the well-known NFFT [30] library, but uses a gridding-based implementation, 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 [12]. Impatient is a GPU-accelerated framework for non-Cartesian sampling trajectories in MRI. Using an output-driven parallelism gridding approach com-

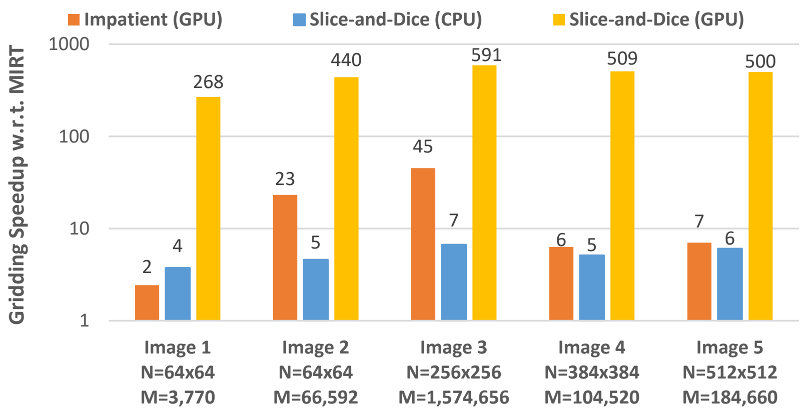


Figure 4.6: Gridding speedups of Slice-and-Dice CPU, Slice-and-Dice GPU, and Impatient [12], normalized to the Michigan Image Reconstruction Toolbox (MIRT). The grid size N and number of non-uniform samples M for each image are labeled; $\sigma = 2$ and $W = 4$ for all images.

bined 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 contains an Intel i9-9900KS with 128 GB of DDR4 3600 MHz memory for the CPU-based benchmarks, and an Nvidia RTX 2080 Ti for the GPU-based benchmarks. The CPU and GPU implementations of Slice-and-Dice use single-precision floating-point values to closely match the prior work. Functional verification and quality evaluation is performed against MIRT, which uses doubles.

4.4.1 Performance Comparison

We evaluate Slice-and-Dice's performance using five images of differing dimension and number of non-uniform samples. While Slice-and-Dice is primarily designed to accelerate the gridding operation found in the adjoint NuFFT, wherein non-uniform samples are interpolated onto a uniform grid, we include the results for a regridding variant which uses the tiled memory structure to obviate binning in the forward NuFFT as well. Figure 4.6 and Figure 4.8 show the standalone gridding and regridding performance of each implementation normalized to the performance of MIRT. End-to-end speedups for the adjoint and forward NuFFT are shown in Figure 4.7 and Figure 4.9. Since 3D gridding is a derivative of 2D

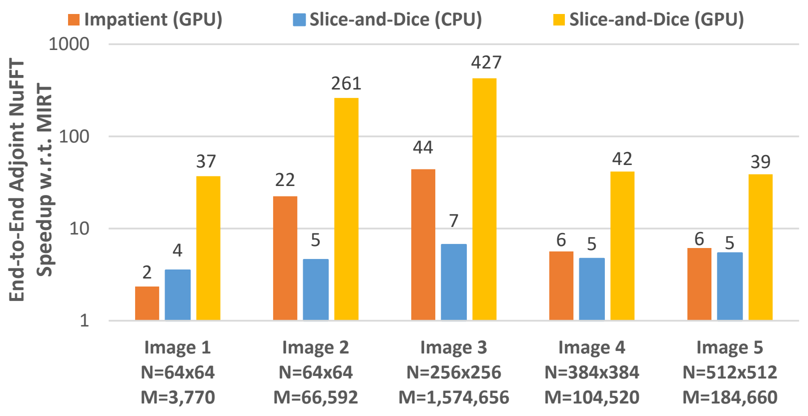


Figure 4.7: End-to-end adjoint NuFFT speedups of Slice-and-Dice CPU, Slice-and-Dice GPU, and Impatient [12], normalized to the Michigan Image Reconstruction Toolbox (MIRT).

gridding—serially operating on 2D slices—for the compared implementations, we only report results for the 2D case.

CPU. Slice-and-Dice CPU's performance is dominated by the GPU implementations when it comes to the larger problem sizes for gridding and regridding operations, but comes surprisingly close to matching that of Impatient in terms the end-to-end performance, as seen in Figure 4.7 and Figure 4.9. Slice-and-Dice CPU averages end-to-end speedups of approximately 5× for the adjoint and forward NuFFTs compared to the baseline. Limited by the available parallelism in the CPU's relatively low core count, its performance suffers from frequent memory stalls due to insufficient parallelism to hide pending accesses.

GPU. The Slice-and-Dice GPU implementation provides massive gains relative to the CPU implementation, as the CUDA threading model offers a way to hide the long external memory latencies through quick context switching among thread warps. Slice-and-Dice GPU's gridding kernel 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—necessary to isolate the threads from write contention—as

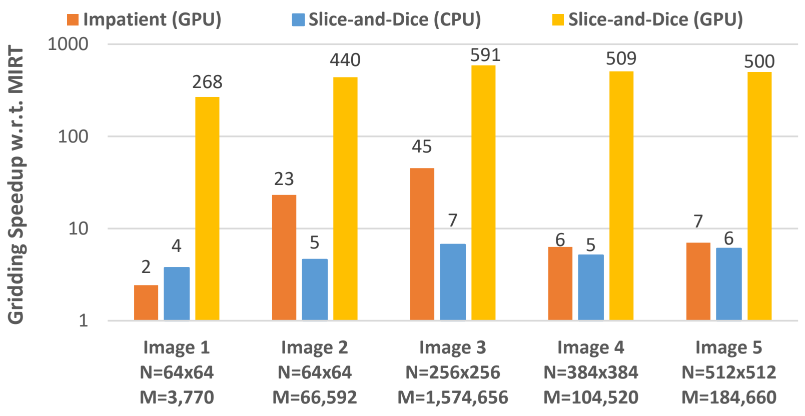


Figure 4.8: Regridding speedups of Slice-and-Dice CPU, Slice-and-Dice GPU, and Impatient [12], normalized to the Michigan Image Reconstruction Toolbox (MIRT). The grid size N and number of non-uniform samples M for each image are labeled; $\sigma = 2$ and $W = 4$ for all images.

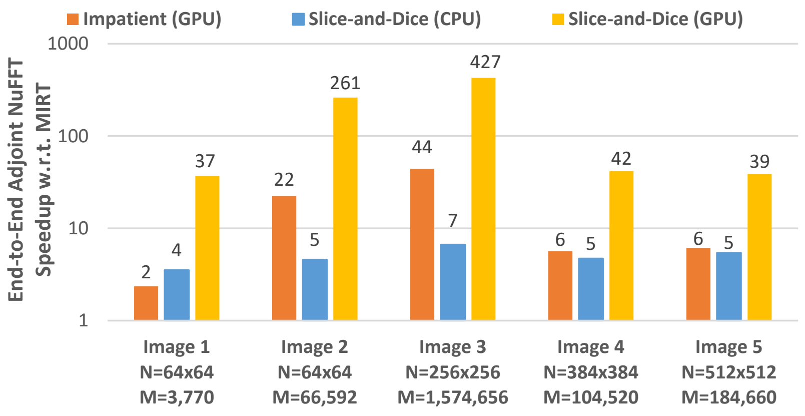


Figure 4.9: End-to-end forward NuFFT speedups of Slice-and-Dice CPU, Slice-and-Dice GPU, and Impatient [12], normalized to the Michigan Image Reconstruction Toolbox (MIRT).

multiple threads may now write to the same output location. We use atomic addition instructions to ensure proper synchronization and that no updates are inadvertently lost. In this manner, the Slice-and-Dice GPU implementation achieves an average gridding speedup of over 446× relative to the baseline and over 43× over Impatient [12] across all problem sizes, as shown in Figure 4.6. The end-to-end adjoint NuFFT performance improvements in Figure 4.7 follow a similar trend, with an average speedup of over 92× relative to the baseline and 10× over Impatient.

Parallelizing regridding for the forward NuFFT is handled differently from the "standard" Slice-and-Dice model used in gridding, as the write conflicts occur on the non-uniform sample set rather than the uniform grid. Slice-and-Dice GPU assigns each non-uniform sample to a thread, eliminating write conflicts for the accumulation operations. The stacked-tile memory structure of Slice-and-Dice is left intact, resulting in average regridding speedups of 262× MIRT and 9× Impatient; end-to-end forward NuFFT benchmarks show average speedups of 79× that of MIRT and 4× that of Impatient.

The significant increases in performance relative to the prior work can be attributed to several reasons: (1) Slice-and-Dice GPU uses a lookup table for interpolation weights, while Impatient [12] calculates them during processing, (2) Slice-and-Dice GPU achieves an L2 hit rate of ~99% across several image problem sizes compared to Impatient's ~73%, (4) Slice-and-Dice operates on all uniform grid points in parallel for both the adjoint and forward operators, while Impatient operates serially on tile–bin pairs, and (5) 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.

4.5 Conclusion

In this chapter we described Slice-and-Dice, a model for NuFFT gridding that maps efficiently to traditional parallel hardware architectures. Slice-and-Dice eliminates pre-processing of the non-uniform data by performing binning on the fly, breaking the uniform grid into multiple tiles and operating on all tiles simultaneously. When implemented on a GPU, Slice-and-Dice achieves average non-uniform interpolation speedups of over 342× and 20× when compared to the CPU baseline and GPU state-of-the-art implementations for representative problems. Slice-and-Dice GPU achieves end-to-end NuFFT speedups of over 85× the CPU baseline, and 6× the state-of-the-art GPU implementation, resulting in non-uniform interpolation and the FFT contributing approximately equal computation time.

CHAPTER V

JIGSAW: Non-uniform Fast Fourier Transform Acceleration for MRI Reconstruction

5.1 Introduction

The Non-uniform Fast Fourier Transform (NuFFT) is a core component of many image reconstruction applications, including magnetic resonance imaging (MRI) [28, 49, 1, 9, 38, 61, 12], computed tomography [39, 62], synthetic aperture radar [27, 17], and radio astronomy [56, 26]. Compared to the standard Fast Fourier Transform (FFT), which requires uniformly spaced data, the NuFFT allows these applications to use non-uniform sampling patterns which can enable reduced imaging scan time or irregular sensor placement [7].

To enable quick processing of an irregular data set, the NuFFT extends the FFT to non-uniform sampling patterns through a three-step process: (1) interpolation from non-uniform to uniform coordinates (or vice versa), (2) a normal [uniform] FFT, and (3) apodization, or weighting, of the uniform data. Fast, efficient, and accurate NuFFT operations are of paramount importance for many 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.

NuFFT performance is dominated by the interpolation operation, which is known as

The work described in this section was partially published in IPDPS 2021 [57].

"regridding" in the forward NuFFT and "gridding" in the adjoint NuFFT. For a given sample set with uniform grid length N, dimensions d, a non-uniform sample set of size M, and an interpolation kernel window of size W, the computational *complexity* of this operation is MW^d. While the computational complexity appears fairly tractable, the NuFFT's computation *time* leaves much to be desired: as recently as a decade ago, this interpolation step accounted for upwards of 90% of the NuFFT's computation time, with less than 10% due to the FFT itself [47, 30, 32, 28], and in the last chapter we found that gridding now accounts for upwards of 99% of computation time in modern systems. The reason that non-uniform interpolation dominates computation time is fairly straightforward: each non-uniform sample in the data set, which can be randomly ordered, affects (or is affected by) a window of non-contiguous points on the uniform grid. With randomly ordered samples in memory, 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 non-uniform interpolation implementations.

5.1.1 Reducing Computational Burden

In an attempt to overcome these challenges, many optimized implementations of non-uniform interpolation have been proposed to improve performance of the NuFFT [1, 24, 47, 38, 32, 31, 12, 3, 2]. 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 [24]. These tiles are often configured such that they are small enough to fit in an on-chip cache or memory [32, 3, 2]. Binning sequentially processes tile–bin pairs, improving memory bandwidth use by reducing the number of cache evictions caused by spatially diverse reads and writes. Binning is common in both software and hardware accelerator works [47, 38, 32, 31, 12, 3, 2], with GPU and FPGA implementations among the best in terms of performance and power efficiency.

In the previous chapter we described Slice-and-Dice, an alternative approach to non-uniform interpolation optimization which obviates the need to presort samples and reduces the number of boundary distance checks through a two-part decomposition of the non-uniform coordinates. Slice-and-Dice breaks the uniform grid into tiles, as in binning, but then stacks them to create “dice.” The contributions from each non-uniform sample are then applied to all points in the dice in parallel in a manner which maps far better to commodity hardware accelerators, such as GPUs. When implemented on a GPU, Slice-and-Dice achieves average non-uniform interpolation speedups of over 342× and 20× when compared to the CPU baseline [8] and state-of-the-art GPU [12] implementations for representative problems. Slice-and-Dice GPU achieves end-to-end NuFFT speedups of over 85× the CPU baseline and 6× the state-of-the-art GPU implementation, resulting in the FFT being the bottleneck for the first time.

5.1.2 Parallelism Through Hardware

Armed with algorithmic optimizations like binning, researchers turned to highly parallel hardware for additional performance gains. To better utilize the parallelism available in modern architectures, GPU implementations [47, 12] commonly turn to output-driven parallelism, wherein each uniform point in a tile is assigned to a single thread—improving performance through drastically increased parallelism at the cost of increasing the computational complexity due to the required all-to-all boundary checks. In [47], Sørensen et al. implemented a fast gridding algorithm on commodity GPUs. In their model, each thread is responsible for the interpolation operations affecting a set of neighboring points, with all points held in registers during the computation. After all of the non-uniform samples have been processed and their effects transferred to the set of uniform points, the registers are written to memory, reducing the number of global writes to N^d. Operating on sets of eight points in parallel, Sørensen et al. achieve a 20-85× speedup versus the contemporary state-of-the-art CPU implementation for various problem sizes and sampling trajectories.

More recently, a Toeplitz-based MRI reconstruction strategy was implemented on GPUs in [12]. Rather than direct matrix inversion, gridding with binning was used for both the forward and adjoint NuFFT, reducing the computational complexity from MN^d to $MW^d + N^d log(N^d)$. To reduce data races caused by multiple non-uniform samples affecting the same uniform points, an output-driven parallelism approach was used, in which every uniform grid point is 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 [61].

Due to limited hardware resources, FPGA implementations [32, 31, 3, 2] process the non-uniform samples serially, parallelizing each sample's W^d contribution to the tile currently being processed. One of the earliest FPGA implementations [32], proposed by Kestur et al., used a Geometric Tiling approach to partition the non-uniform source samples into small sets, called *source tiles*, based on which region of the uniform grid they affected. The uniform grid was then broken into slightly larger *target tiles*, such that the edge of the source tile was $W/2$ points from the edge of the target tile on each side in order to accommodate the interpolation kernel width. This choice required target tiles to have an overlap, while source tiles were non-overlapping. Non-uniform samples were dynamically added to linked lists associated with their relevant tile, ensuring maximum memory-write throughput of the samples through use of contiguous local memory storage and thereby improving spatial locality. A trajectory-optimized variant was later implemented [31].

As an alternative to linked lists, Cheema et al. design an FPGA gridding accelerator in [3], which implements binning using a set of fixed-sized FIFOs. Designed for random, unsorted sample trajectories, the FPGA reads non-uniform samples from external memory and sorts them into the FIFOs, with a fill-rate based arbiter determining which FIFO should be processed and the corresponding tile then loaded into on-chip memory. They later optimize memory bandwidth utilization using a trajectory-specific arbiter to process FIFOs in an order predetermined by the sampling trajectory of the input data [2]. Use of a priori knowledge of the sampling trajectory allows for 20-25% reduced logic in the arbiter and reduces the number of stalls in the pipeline, resulting in a 2-3× increase in gridding performance. However, the limited hardware available on the FPGA continued to restrict the number of parallel operations achievable to the interpolation window size W. With a fixed interpolation window width $W = 4$, 16 parallel MAC operations were achieved.

While these hardware implementations resulted in significant performance improvements, they all suffer from the severely limited parallelism imposed by the binning model, processing of samples multiple times due to overlapping interpolation kernels or tiles, and suboptimal memory bandwidth utilization for random or unknown trajectories.

5.1.3 Contributions

While the prior work has demonstrated significant improvements in performance using commodity hardware accelerators, all of them rely on some form of binning—thereby limiting the available parallelism, processing of some samples multiple times due to interpolation window overlaps, and requiring additional processing and memory overhead to handle the bins. In this chapter we present JIGSAW, a streaming accelerator architecture for the NuFFT which supports gridding, regridding, FFT2, and IFFT2 operations. JIGSAW implements Slice-and-Dice with a custom stack-of-tiles memory layout. Using a small set of computational pipelines, JIGSAW performs non-uniform interpolations 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 relies on a mixture of 16-bit fixed-point and 32-bit floating-point functional units to simultaneously decrease hardware complexity and lower area and power requirements. With a fully pipelined architecture and sufficient caching to handle all outstanding memory requests, JIGSAW experiences no computational stalls—leading to average non-uniform interpolation speedups of over 1058× relative to the CPU baseline [8] and over 60× when compared to the state-of-the-art GPU implementation [12] across all problem sizes tested.

The 2D radix-2 FFT/IFFT implementation used by JIGSAW achieves an average speedup of 15× versus the baseline, resulting in end-to-end NuFFT speedups of over 558× vs the CPU baseline and 41× the state-of-the-art GPU implementation. Implemented in SystemVerilog and synthesized using an industrial 16 nm node, JIGSAW requires an area of ~13 mm^2 and a power budget of only ~181 mW across all operational modes—nearly 1800× more power efficient than the GPU Slice-and-Dice implementation described in the previous chapter.

5.2 Background

When dealing with irregularly sampled data, computation of the Fourier Transform cannot be performed with a conventional FFT algorithm. Instead, computational imaging applications like MRI [28, 49, 1, 9, 38, 61, 12], computed tomography [39, 62], synthetic aperture radar [27, 17], and radio astronomy [56, 26] must rely on the NuFFT, which supports data that does not lie on a uniform grid. To understand the algorithmic differences imposed by non-uniform data, in this section we will first look at the Non-uniform Discrete Fourier Transform (NuDFT), which can be used to directly compute the Fourier Transform of non-uniform data. We then take an in-depth look at the NuFFT, which allows for efficient approximation of the NuDFT by combining an interpolation step—used to map the non-uniform data to and from a uniform grid—with the traditional Fast Fourier Transform.

5.2.1 Non-uniform Discrete Fourier Transform

A generalization of the forward Discrete Fourier Transform, the NuDFT allows for computation over non-uniform—or non-equispaced—input data. Following the notation found in [47, 30], 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

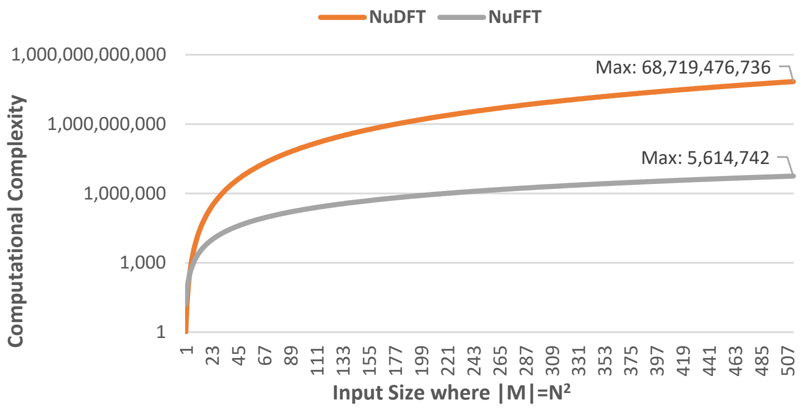


Figure 5.1: Computational complexity of the unoptimized 2D NuDFT and NuFFT.

uniform points k in $\{0,\ldots,N-1\}^d$, the forward NuDFT is used to compute

$$f_j = \sum_{\boldsymbol{k}\in\{0,\ldots,N-1\}^d} \hat{f}_{\boldsymbol{k}} \mathrm{e}^{-2\pi i \boldsymbol{k}\cdot\boldsymbol{x}_j}, \quad j = 0,\ldots,M-1 \tag{5.1}$$

The adjoint NuDFT is similarly defined as

$$\hat{h}_{\boldsymbol{k}} = \sum_{j=0}^{M-1} \hat{f}_j \mathrm{e}^{2\pi i \boldsymbol{k}\cdot\boldsymbol{x}_j}, \quad \boldsymbol{k} \in \{0,\ldots,N-1\}^d \tag{5.2}$$

As shown in Figure 5.1, direct calculation of these operations requires MN^d floating-point operations, which is too expensive for many applications even for small problem sizes.

5.2.2 Non-uniform Fast Fourier Transform

The NuFFT extends the traditional FFT to support non-uniform data, providing approximate solutions to the NuDFT with far lower computational complexity and memory requirements. Using three steps, (1) apodization, (2) an FFT, and (3) a non-uniform interpolation, the NuFFT computes approximately 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 combina-

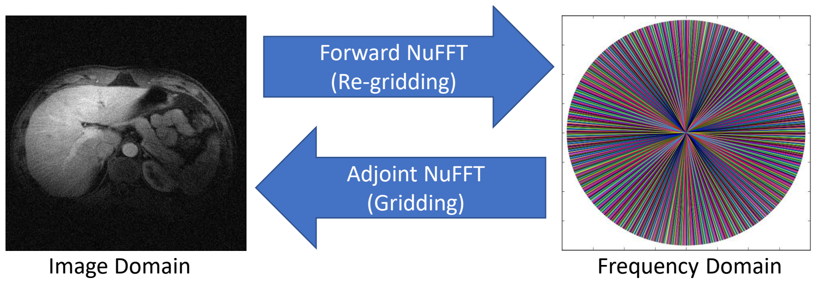
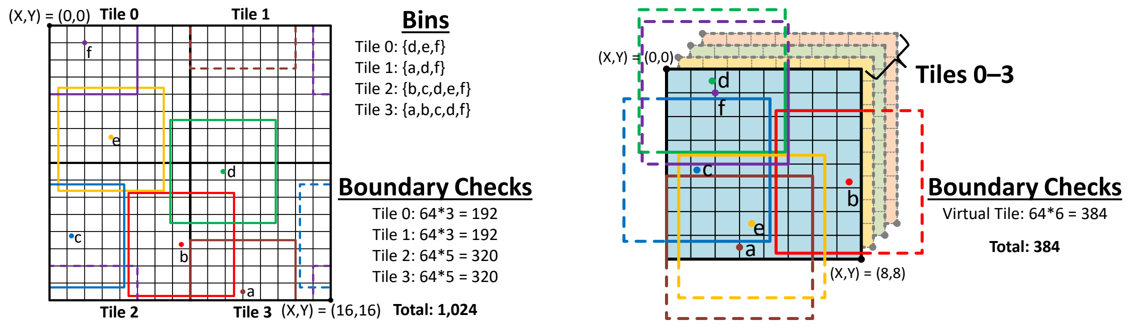


Figure 5.2: Each NuFFT variant comprises three steps. Forward: (1) pre-apodization, (2) FFT, (3) regridding. Adjoint: (1) gridding, (2) FFT, (3) de-apodization. Image data from [40].

tions of uniform and non-uniform inputs and outputs, with the forward and adjoint NuFFTs a core component in many image reconstruction algorithms. As shown in Figure 5.2, the forward NuFFT transforms image data to the frequency domain, while the adjoint NuFFT transforms frequency data to the image domain. The time required to compute the NuFFT is dominated by steps (2) and (3), with step (3) historically accounting for 95% or more of the computation time [47, 30, 32, 28]. This interpolation step, often called re-gridding in the forward NuFFT and gridding in the adjoint NuFFT, transforms the data between uniform grids of N^d points and M non-uniform samples using a kernel width of W for each dimension. Due to the periodicity of the complex exponential functions, the uniform grid is really a flattened torus, so any sample that lies within $W/2$ of an "edge" will involve interpolation using "neighbors" that must be determined using periodic boundary conditions, as illustrated in sample points a, c, and f in Fig. 5.3. The interpolation kernel itself can be one of a variety of windowing functions, including Kaiser-Bessel, Gaussian, B-spline, Sinc, and more; the choice of windowing function is application-specific.

To improve the NuFFT interpolation accuracy, an oversampling factor σ, often set to two, is multiplied by the input grid dimensions prior to the interpolation process. While crucial for accuracy, oversampling comes with two undesirable traits: (1) increased FFT computation time, as an N sample FFT now becomes a σN FFT along each dimension, and (2) increasing memory requirements as N or d grow. To alleviate these affects, Beatty et al. proposed a method of gridding using smaller oversampling factors, i.e., $\sigma \leq 2$. However, when the



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

Figure 5.3: Binning vs Slice-and-Dice. Binning presorts data into bins corresponding to which tiles the samples' interpolation windows intersect. In this example, binning performs a boundary check between each of the 64 uniform points in a tile and each non-uniform sample in the 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 of the stack (i.e., between columns) and each sample to determine which columns are affected. We then calculate the tile coordinates (i.e., "depth" in the stack) to determine the location of the uniform point affected. Using our stacked tile approach, Slice-and-Dice requires only 384 boundary checks versus 1,024 required by binning.

oversampling factor σ is reduced, the interpolation kernel must be widened (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 causes the NuFFT to be even further dominated by the interpolation operation.

5.2.3 Traditional Gridding

Because the non-uniform interpolation operation dominates the computation, optimizing the interpolation—hereafter referred to as "gridding" for simplicity—has been a focus of many previous works [1, 3, 2, 8, 9, 12, 28, 30, 32, 31, 38, 47]. To understand why gridding dominates NuFFT computation time, we first consider a typical gridding implementation and its basic parameters. As an example, we use the adjoint NuFFT, wherein gridding is performed to map non-uniform frequency data onto a uniform grid before 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 the target system and other implementation parameters. In particular, gridding time depends on the oversampling factor σ, interpolation kernel width W, and the interpolation table oversampling factor L. The interpolation kernel width W—commonly set to four or six—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]. The final parameter, L, defines the granularity of the coordinates considered by the interpolation kernel window; i.e., there are WL discrete weights in the interpolation kernel window, and coordinates are rounded to the nearest weight. By constraining the kernel granularity in this manner, offline precomputation of the discrete kernel weights becomes possible, reducing the amount of online computation required for each interpolation operation; these weights are commonly stored in a look-up table (LUT). Through modification of these three parameters, gridding's computation time, memory requirements, and error can be adjusted based on application requirements.

5.2.3.1 Gridding Parallelism

Whereas gridding has tractable computational complexity, modern hardware realities—notably the significantly slower memory speed versus processor speed—can lead to considerable slowdowns based on the implementation. Since non-uniform samples often arrive in effectively random order and each affects a window of W uniform points that are discontiguous in memory, gridding commonly suffers from poor memory locality.

The simplest possible gridding implementation processes the randomly-ordered non-uniform samples serially. The serial approach avoids 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 naïve approach is not parallel.

Instead, GPU and FPGA implementations commonly turn to output-oriented parallelism, wherein one thread accumulates all sample values that affect a single grid point. This approach does not need any synchronization among threads, since each thread modifies disjoint memory locations. However, a naïve output-parallel implementation requires 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 about as expensive as the interpolation operation (a table lookup and multiply-accumulate) and the vast majority of the checks will fail, since each sample affects only a small number of grid points. As such, naïve output-parallel gridding is inefficient.

5.2.3.2 Reducing Computational Burdens via Binning

To reduce the number of boundary checks required by output-driven parallelism, modern implementations instead rely on a form of geometric tiling known as *binning*. Binning, visualized in Figure 5.3a, 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 corresponding to the tiles that they affect. In Figure 5.3a, Tile 0 has an associated subset consisting of points $\{d,e,f\}$, Tile 1 has an associated subset $\{a,d,f\}$, and so on. Processing the points in this manner allows for significant execution time improvement, as the tile associated with each set remains cached while the subset is processed.

Whereas binning greatly improves spatial locality and reduces the memory stalls, it still suffers from several factors that cost 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. 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 (e.g., sample f in Figure 5.3a). Third, and perhaps most important, even if bins fit in on-chip caches (or another local memory structure), that memory does

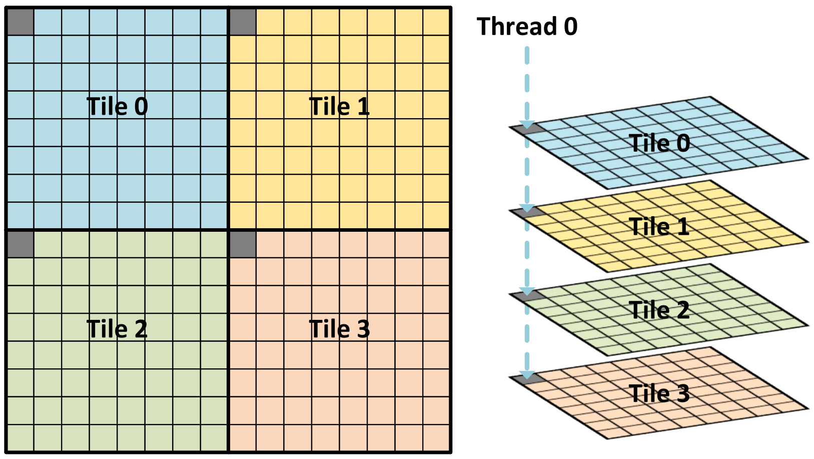


Figure 5.4: Slice-and-Dice virtually stacks tiles on top of each other, with a single thread processing all samples that affect a given position in any tile (i.e., a column in the stack). This drastically reduces the number of all-to-all boundary checks, as each sample must only be checked against each column.

not necessarily provide single-cycle access—binning *reduces* stalls due to pending memory reads, but does not eliminate them. Our objective is to eliminate the duplicate sample processing, pre-processing, and remaining memory stalls encountered using binning.

5.2.4 Slice-and-Dice Design

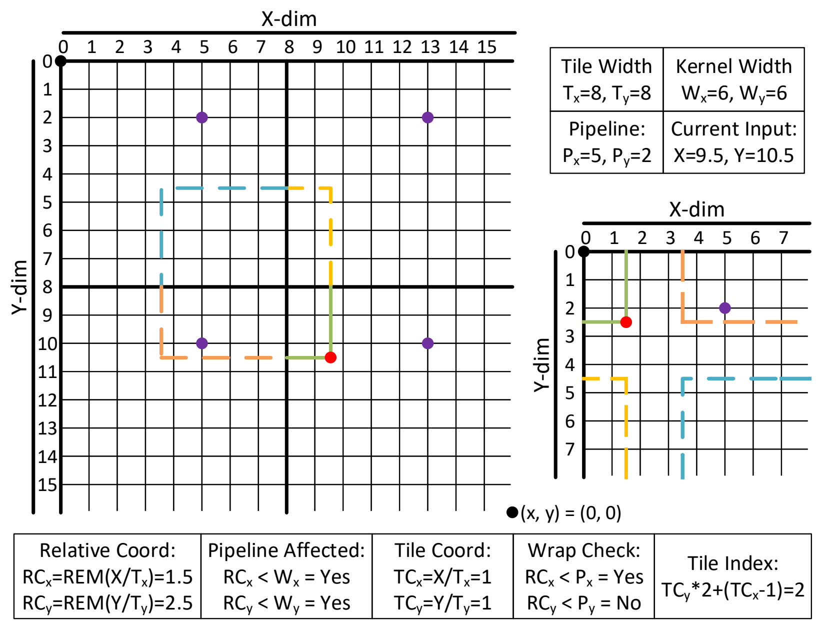
We next review the Slice-and-Dice NuFFT acceleration model, its design choices, and the streaming approach, which enable substantial performance gains in conventional parallel architectures and enable highly optimized hardware accelerator designs.

5.2.4.1 Rethinking Binning

Most recent acceleration approaches 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 overlaps adjacent tiles. While binning does reduce the amount of unnecessary boundary checks, it requires non-trivial precomputation time, additional memory, and overhead to manage the bins.

As an alternative to binning, in the previous chapter we introduced Slice-and-Dice, a novel gridding optimization technique which obviates the need for an additional binning step. We implement Slice-and-Dice by leveraging a stacked-tile, data-driven parallel model and a unique decomposition of the sample coordinates. As shown in Figures 5.3b and 5.4, 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* and mapped to a 2D block of T^d parallel threads. Each thread processes a single position in each of the virtual tiles, which correspond to a column in the dice; that is, Thread 0 will process the target point at relative position $(x,y) = (0,0)$ in each of the four virtual tiles shown in Figure 5.4.

Handling boundary checks in the virtual tile model relies on a novel decomposition of the input samples' coordinates. For coordinates ranging from $[0 : N)$ in each dimension, we perform a two-part decomposition by dividing the coordinate by the virtual tile's dimension, with the quotient referred to as the *tile coordinate* and the remainder as the *relative coordinate*. Using an arrangement in which the target grid points accessed by each thread are placed in a single contiguous array, we determine the index into that array by finding the *global tile address*, which is a combination of the tile coordinates in each dimension—much like calculating the total linear index in GPU programming. The relative coordinate, on the other hand, indicates where the sample lies within the virtual tile, and whether the current input sample affects the uniform points assigned to a given thread. By using the relative coordinate in this manner, we reduce the computational complexity of Slice-and-Dice compared to that of standard output-driven parallelism. 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 only MT^d total checks. Even with an all-point to all-column comparison, use of the Slice-and-Dice binning-free model results in a computational complexity reduction of N^d/T^d vs a naïve no-binning approach.



Tile Width	Kernel Width
T_x=8, T_y=8	W_x=6, W_y=6
Pipeline: P_x=5, P_y=2	Current Input: X=9.5, Y=10.5

Relative Coord:	Pipeline Affected:	Tile Coord:	Wrap Check:	Tile Index:
RC_x=REM(X/T_x)=1.5	$RC_x < W_x$ = Yes	TC_x=X/T_x=1	$RC_x < P_x$ = Yes	TC_y*2+(TC_x-1)=2
RC_y=REM(Y/T_y)=2.5	$RC_y < W_y$ = Yes	TC_y=Y/T_y=1	$RC_y < P_y$ = No	

Figure 5.5: Slice-and-Dice in action. The thread assigned to handle uniform points with relative positions $(x, y) = (5, 2)$ is affected by an input sample with coordinates $(9.5, 10.5)$ if both relative coordinates are less than the interpolation kernel width. The thread is affected in tile $(0, 1)$ but the sample's coordinate is in tile $(1, 1)$, causing a wrap in the X dimension. If a wrap occurs, the corresponding tile coordinate is offset.

5.2.4.2 Simplifying Boundary Checks

Because the frequency domain target grid is a torus, care must be taken to handle situations where the interpolation kernel "wraps" around to the other edge of the target grid. As seen in Figure 5.5, the stacked tile structure of Slice-and-Dice exacerbates this issue, as wrapping now occurs along the edge of each tile rather than only at the edges of the grid as a whole. As an example, consider a 1D tile of size $T = 8$ and an interpolation width $W = 6$. For an input sample with a coordinate value of 2, there will be a wrap to the left, while for a coordinate value is 6, there will be a wrap to the right. Tracking wraps in both directions is undesirable for efficient implementations, as it requires a far more complex boundary check.

Slice-and-Dice simplifies the handling of wraps by offsetting the input coordinates by $W/2$, which eliminates wraps to the left. As an added benefit, the offset significantly reduces the complexity of the boundary check itself, as it eliminates the need to calculate distance in the negative direction for each dimension. Rather than checking if the wrapped distance between a tile's thread and input sample is less than $W/2$ in either direction, Slice-and-Dice only checks if the distance in the positive direction is less than W. Figure 5.5 shows a complete example for a single non-uniform sample, demonstrating the coordinate decomposition, wrap handling, and global tile address calculation.

5.2.5 A Streaming Approach

As discussed in previous sections, a primary bottleneck in many gridding acceleration works is the memory system, whether it be a lack of bandwidth or stalls due to non-contiguous memory accesses. Commodity hardware accelerators, such as GPUs, try to hide the latency of memory reads using massively parallel approaches, but do not fully overcome the challenges presented by the NuFFT's interpolation. The Slice-and-Dice model offers a processing structure which allows for efficient implementation in hardware, as there is no interaction between adjacent threads (or pipelines). With minimal communication and sufficient on-chip buffering, custom streaming architectures can do away with the traditional load and store memory hierarchy and thereby attain orders of magnitude better performance and power efficiency.

5.3 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. The Slice-and-Dice model's stacked slice memory layout eliminates unnecessary hardware interaction in custom accelerators. With a single 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.

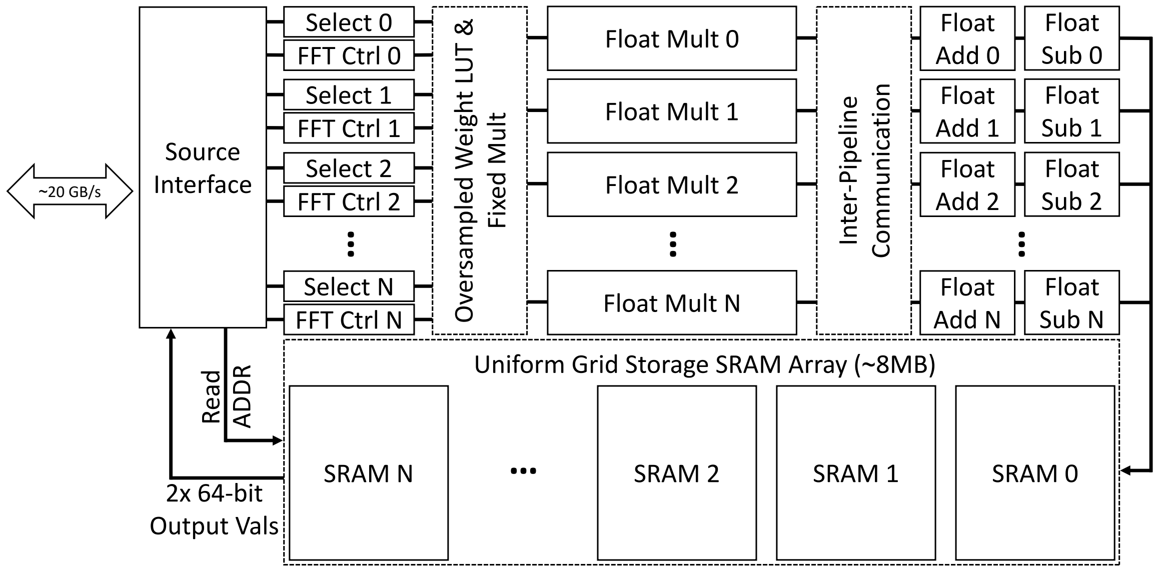


Figure 5.6: JIGSAW microarchitecture. Operating at 1.0 GHz, non-uniform samples arrive on a 128-bit bus and are passed serially through the pipelines, which are connected in the Source Interface. Pipelines are logically arranged in a 2D grid, with FFT2 and IFFT2 communication occurring across rows and columns. Each pipeline has a local SRAM array to hold the uniform points in its column.

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 NuFFT acceleration. As a hardware implementation of Slice-and-Dice, JIGSAW comprises two components. The first is a set of pipelines which fully process an input stream of non-uniform samples in a single stall-free pass, accumulating contributions to or from the uniform target grid columns, which are stored in private memory arrays. These pipelines enable non-uniform interpolation runtime linear in the number of non-uniform samples, independent of ordering of the input samples, uniform grid size, or interpolation window width. The second component is a custom FFT implementation which supports 2D radix-2 FFT and IFFT operations in JIGSAW's stacked-tile memory hierarchy, allowing acceleration of the two most computationally-intensive components of the NuFFT. In this section, we describe the basic hardware implementation of JIGSAW's non-uniform interpolation pipelines and its custom FFT2/IFFT2 implementation.

As illustrated in Figure 5.6, JIGSAW comprises a set of identical pipelines broken into several functional units. For the remainder of this chapter, we assume a target grid with

Table 5.1: 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
Weight Bit Width	16-bit

$N = 1024$ points in each dimension, a virtual tile size with $T = 8$ points in each dimension, an interpolation window width of $W = 6$, and an interpolation table oversampling factor of $L = 32$. JIGSAW's pipelines are logically arranged as a 2D grid of dimensions T^2 to match the virtual tile size; the range of supported runtime parameters are listed in Table 5.1. To enable functional verification and obtain power/area synthesis estimates, we implement JIGSAW in SystemVerilog using these parameters.

5.3.1 Fixed-Point

Fixed-point is a representation often used in hardware accelerators due to the decreased complexity of functional units relative to floating-point variants. Hardware implementations of Slice-and-Dice benefit from using fixed-point representations of the input sample coordinates through bit manipulation and optimizing the dimensions of the virtual tiles. In JIGSAW, we utilize a mixture of fixed-point and floating-point hardware modules to accommodate the need for both high precision and high dynamic range, which varies significantly between operational modes. Mathematical operations with at least one operand being data (either non-uniform sample data or uniform grid data) are performed in floating-point, while operations involving only weights are performed in fixed-point. Using fixed-point weights results in a 50% reduction in storage requirements and reduced hardware complexity for the weight combination required in the non-uniform interpolation operations; weights are dynamically converted to floating-point values prior to any operations involving data.

5.3.2 JIGSAW Pipeline Microarchitecture

Each of JIGSAW's pipelines $T_{x,y}$ is comprised of seven functional units. The individual functionality of each unit is described below, followed by descriptions of how the units are ordered and used for the gridding, regridding, FFT2, and IFFT2 operations.

Select. The select unit is responsible for determining whether a non-uniform sample affects or is affected by the grid points assigned to a given pipeline. To determine whether the pipeline should process a given non-uniform sample, the select unit breaks the non-uniform coordinates into the relative and tile coordinates defined by Slice-and-Dice. The unit then calculates the distance from the indices assigned to the pipeline to the relative coordinates. For each dimension, the select unit performs the distance calculation in hardware using three 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 (3) subtracting the pipeline's index from the sum. The resulting value is the forward distance (i.e., left to right in 1D) from the pipeline's column to the input sample. The select unit then compares the distance against the interpolation window size to determine whether the window intersects a point in the column. If the distance in each dimension is less than the interpolation window size, the sample proceeds to be processed by this pipeline.

For any sample which must be processed, the select unit uses the previously truncated coordinate bits (the tile coordinate) to determine the appropriate entry in the uniform memory array. Interpolation windows that overlap 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 checks 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 5.5. The updated tile coordinates are then combined to form the global linear address in the uniform memory array.

The select unit next calculates the interpolation weight addresses. The weight 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 weight addresses by multiplying the previously calculated distances by the table oversampling factor—32 in this example—and rounding to the nearest integer.

FFT/IFFT Control. The FFT/IFFT unit is responsible for generating all of the addresses required by the FFT2 and IFFT2 operations, both along rows and across columns. JIGSAW's tiled memory layout, optimized for non-uniform interpolation operations, presents a unique problem when it comes to the FFT, as each pipeline only has access to its local memory array. To minimize the hardware complexity when transmitting data between pipelines in multiple dimensions, JIGSAW utilizes a decimation-in-time radix-2 FFT implementation, wherein the FFT is broken into $log_2(N)$ stages of "butterfly" operations. Each butterfly reads two points from the uniform arrays, multiplies one point by a weight, or "twiddle factor," and then adds the result to and subtracts the result from the second value before storing the results.

Due to the nature of the radix-2 FFT algorithm and the stacked-tile memory layout in JIGSAW, the amount of inter-pipeline communication is dictated by the dimensions of the JIGSAW pipeline array. Given an FFT of length N and an array of pipelines T in a single dimension, each pipeline can perform $log_2(N) - log_2(T)$ stages internally—meaning that both uniform points are contained within its private memory array—whereas $log_2(T)$ stages require communication with other pipelines to obtain one of the two values required for the butterfly operation. In our $T = 8$ pipeline, this results in three stages requiring communication.

The uniform memory addresses, the twiddle factor (or weight) addresses, and the communication distance for each dimension are generated by the FFT/IFFT unit. Based on the operational mode, these control signals are passed to the other modules in the JIGSAW pipeline to set up the datapath required for each stage of the FFT2 and IFFT2 operations.

Uniform Memory Lookup & Accumulation. The uniform memory & accumulation unit functions as a wrapper for the pipeline's associated Slice-and-Dice column, storing 64-

bit floating-point complex values (32 bits for each real and imaginary component). Adders are colocated with the local SRAM arrays that store the partial sums to support gridding's accumulation operation without superfluous external caching mechanisms. By accepting read, write, and accumulation commands, the uniform memory & accumulation unit is used for all supported JIGSAW operations, namely gridding, regridding, FFT2, and IFFT2.

Weight Lookup & Combination. The weight lookup & combination unit is a wrapper containing a dual-ported SRAM capable of storing up to 512 32-bit fixed-point complex weights (16 bits for each real and imaginary component). Given that the dynamic range of the weights is extremely small for all supported operations—ranging from -1 to 1—fixed-point storage and combination allows for significant reductions in weight storage and hardware complexity. The weight lookup unit operates in two modes: (1) a read-two-combine-and-output mode, used for the gridding and regridding operations, and (2) a standard read-one-and-output mode, used for the FFT2 and IFFT2 operations. For the first mode, capacity for 512 weights enables an oversampling factor of up to $L = 64$ given a maximum interpolation kernel width of $W = 8$. One weight is read for each dimension, based on the distance calculated in the select unit. These complex weights are then multiplied to form the final interpolation 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 [33]. For the second mode, one weight is read from the SRAM and immediately output to be used in the FFT2 and IFFT2 butterfly operations.

Complex Multiplication. The complex multiplication unit multiplies the complex weight and the complex data, again using Knuth's method [33]. To ensure that the large dynamic range found between the different operation modes are supported, a custom implementation of single-precision floating-point is used for the multiplication, addition, and subtraction operations. Since the weights are stored and combined in a fixed-point representation, each complex multiplication unit is capable of converting the 32-bit fixed-point complex weights to 64-bit floating-point complex weights, where 32 bits are used for

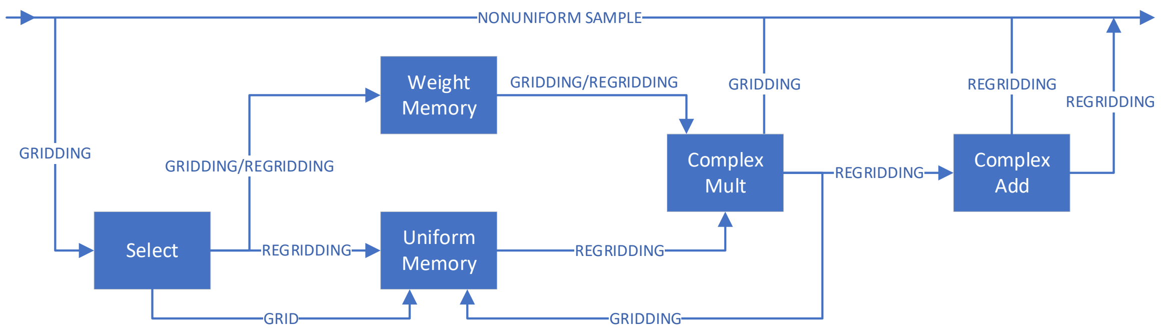


Figure 5.7: JIGSAW non-uniform interpolation datapath.

each of the real and imaginary components, respectively.

Complex Addition. The complex addition unit adds the real and imaginary components of two floating-point operands, returning the sum. To simplify the hardware, a custom single-precision floating-point implementation was created in which many of the edge cases found in IEEE 754-compliant hardware are removed. Subnormals, for example, saturate to the smallest "normal" value that can be stored in standard single-precision floating-point. Similarly, values with a magnitude too large to store in the available number of bits are saturated to the largest value available.

Complex Subtraction. The complex subtraction unit subtracts the real and imaginary components of two floating-point operands, returning the difference. The individual floating-point subtractions are implemented by using adders and flipping the second operand's sign bit.

5.3.3 Non-uniform Interpolations in JIGSAW

To support both forward and adjoint NuFFT operations, JIGSAW's pipelines reconfigure the internal datapath based on the current operation being performed. The datapaths used in the NuFFT's non-uniform interpolations, found in the gridding and regridding modes, can be seen in Figure 5.7. To maximize functional module reuse and avoid write conflicts, each of JIGSAW's pipelines are connected together to form a chain through which non-uniform samples pass in series. For both operations, the select unit determines which samples contain

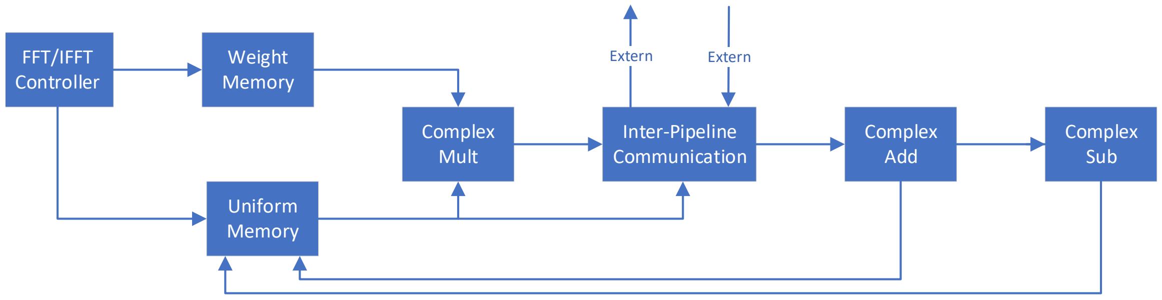


Figure 5.8: JIGSAW FFT2/IFFT2 datapath.

a contribution to a point in the given pipeline (or vice versa) and generates all corresponding addresses for the weight and uniform data memory units.

The weight lookup unit receives two addresses from the select unit, indicating the two weights which must be combined. In the gridding operation, the complex multiplication occurs between the non-uniform sample data and the combined weight, whereas in the regridding operation the complex multiplication involves data read from the uniform memory array and the combined weight. After the complex interpolation is complete, the value is either sent to the uniform memory array for accumulation with the value stored there (for gridding) or sent to the complex addition unit to be added to the non-uniform sample's data value. The non-uniform sample is then sent to the next pipeline, where the process begins anew.

5.3.4 FFT2/IFFT2 in JIGSAW

The 2D FFT and IFFT operations in JIGSAW utilize a custom radix-2 decimation-in-time implementation. The datapath—shown in Figure 5.8—for both operations is the same, with the only difference being that the real and complex components of each sample are swapped when reading from or writing to the uniform memory arrays when performing the inverse variant. These operations utilize six of the seven functional units to implement the butterfly operation, with only the select unit being idle while processing an FFT2 or IFFT2.

Due to the tiled memory layout and private memory arrays, inter-pipeline communication is required for $log_2(T)$ stages in the FFT2 and IFFT2 operations. The FFT/IFFT controller unit uses the stage index and the internal pipeline index to determine which pipeline it will communicate with. In each of the $log_2(T)$ stages, pipelines are paired—meaning that no pipeline must ever communicate with more than one other pipeline in any given stage. For an example, consider an FFT or IFFT in the first dimension with pipeline $T_{0,0}$, $log_2(N) = log_2(1024) = 10$ stages, and $log_2(T) = log_2(8) = 3$. For the first seven stages all operands are available in the pipeline's private memories, therefore requiring no external communication. For the last three stages (8 through 10), the pipeline must transmit and receive one value between each of the following pipelines ion the X-dimension: in stage 8 pipeline $T_{0,0}$ communicates with pipeline $T_{4,0}$, in stage 9 pipeline $T_{0,0}$ communicates with pipeline $T_{2,0}$, and in stage 10 pipeline $T_{0,0}$ communicates with pipeline $T_{1,0}$; identical communication happens with pipelines in the Y-dimension for the FFT/IFFT second dimension.

JIGSAW handles communication between all required pipeline pairs by having each pipeline output the value stored in their private arrays, and rerouting all of the values back into the pipeline in each dimension; i.e., each pipeline sees 8 values in each dimension, and the current stage determines which value is used for internal calculations. In each stage, the FFT/IFFT controller outputs new data and weight addresses each cycle, resulting in deterministic, constant-time performance determined only by the length of the FFT. All internal signals are delayed to account for transmission time, enabling fully-pipelined operations—once the pipeline has filled, one butterfly operation is computed each cycle.

5.3.5 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 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 processed 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 latency.

5.4 Evaluation

To evaluate our JIGSAW microarchitecture, we compare our CPU and GPU implementations of Slice-and-Dice to the SystemVerilog ASIC variant with virtual tiles of dimension 8×8. We employ the Michigan Image Reconstruction Toolbox (MIRT) [8]—a matrix-vector 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 [30] library, but uses a gridding-based implementation which allows for a more direct comparison.

To highlight how JIGSAW performs versus a state-of-the-art GPU implementation, we compare to Impatient [12]. 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 all cores locked to 5.0 GHz and 128 GB of DDR4 3600 MHz memory for the CPU-based benchmarks, and an Nvidia RTX 2080 Ti for the GPU-based benchmarks. The CPU and GPU implementations of Slice-and-Dice use single-precision floating-point values to closely match the prior work, while the ASIC implementation uses a 32-bit pipeline with a combination of fixed-point and floating-point functional units. We measure the performance of our JIGSAW ASIC implementation using

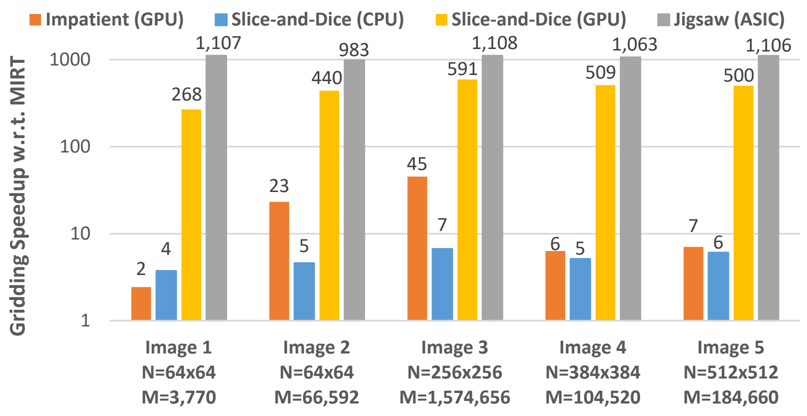


Figure 5.9: Gridding speedups of Slice-and-Dice CPU, Slice-and-Dice GPU, JIGSAW, and Impatient [12], normalized to the Michigan Image Reconstruction Toolbox (MIRT). The grid size N and number of non-uniform samples M for each image are labeled; $\sigma = 2$ and $W = 4$ for all images.

individual tests for each of the NuFFT's components, estimating latency using its synthesized 1.0 GHz clock frequency and the number of cycles required to complete the operation and drain the pipelines when simulating each input set. Functional verification and quality evaluation is performed against MIRT's output using doubles.

For power and area analysis, we synthesize our SystemVerilog implementation of JIGSAW using an industrial 16 nm node and a 1.0 GHz clock speed. Each operational mode's power is estimated individually using separate Switching Activity Interchange Format (SAIF) files generated by simulations with the largest supported problem sizes.

5.4.1 Performance Comparison

We evaluate JIGSAW's performance using five images of differing dimension and number of non-uniform samples. Figure 5.9 and Figure 5.11 show the standalone gridding and regridding performance of each implementation normalized to the performance of MIRT. End-to-end speedups for the adjoint and forward NuFFT are shown in Figure 5.10 and Figure 5.12. 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.

CPU. Slice-and-Dice CPU's performance is dominated by the GPU implementations when it comes to the larger problem sizes for gridding and regridding operations, but comes

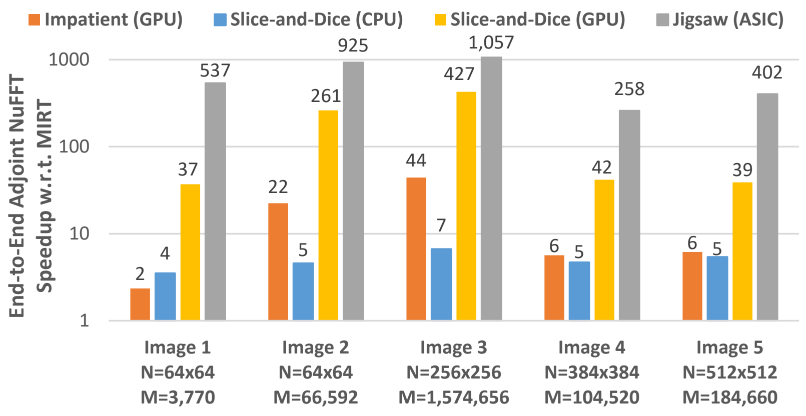


Figure 5.10: End-to-end adjoint NuFFT speedups of Slice-and-Dice CPU, Slice-and-Dice GPU, JIGSAW, and Impatient [12], normalized to the Michigan Image Reconstruction Toolbox (MIRT). The grid size N and number of non-uniform samples M for each image are labeled; $\sigma = 2$ and $W = 4$ for all images.

surprisingly close to matching Impatient in terms the end-to-end performance, as seen in Figure 5.10 and Figure 5.12 where the number of non-uniform samples is not significantly higher than N^2. Slice-and-Dice CPU averages end-to-end speedups of approximately 5× for the forward and adjoint NuFFTs compared to the baseline. Limited by the available parallelism in the CPU's relatively low core count, its performance suffers from frequent memory stalls due to insufficient parallelism to hide pending accesses.

GPU. The Slice-and-Dice GPU implementation provides massive gains relative to the CPU implementation, as the CUDA threading model offers a way to hide the long external memory latencies through quick hardware context switching among thread warps. Regridding for the forward NuFFT is handled differently from the "standard" Slice-and-Dice model used in gridding, as the write conflicts occur on the non-uniform sample set rather than the uniform grid. Slice-and-Dice GPU assigns each non-uniform sample to a thread, eliminating write conflicts for the accumulation operations. The stacked-tile memory structure of Slice-and-Dice is left intact, resulting in average regridding speedups of 262× MIRT and 9× Impatient; the end-to-end forward NuFFT benchmarks shown in Figure 5.12 demonstrate average speedups of 79× that of MIRT and 4× that of Impatient.

Slice-and-Dice GPU's gridding kernel uses blocks of 8×8 threads, where each thread is

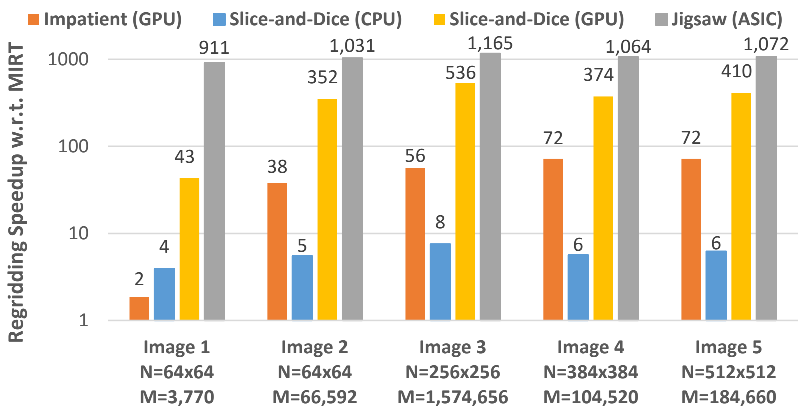


Figure 5.11: Regridding speedups of Slice-and-Dice CPU, Slice-and-Dice GPU, JIGSAW, and Impatient [12], normalized to the Michigan Image Reconstruction Toolbox (MIRT). The grid size N and number of non-uniform samples M for each image are labeled; $\sigma = 2$ and $W = 4$ for all images.

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—necessary to isolate the threads from write contention—as multiple threads may now write to the same output location. We use atomic accumulation operations to ensure proper synchronization and that no updates are inadvertently lost. In this manner, the Slice-and-Dice GPU implementation achieves an average gridding speedup of over 446× relative to the baseline and over 43× over Impatient [12] across all problem sizes, as shown in Figure 5.9. The end-to-end adjoint NuFFT performance improvements in Figure 5.10 follow a similar trend, with an average speedup of over 92× relative to the baseline and 10× over Impatient.

The significant increases in performance relative to the prior work can be attributed to several reasons: (1) Slice-and-Dice GPU uses a lookup table for interpolation weights, while Impatient [12] calculates them on the fly, (2) Slice-and-Dice GPU achieves an L2 hit rate of ~99% across several image problem sizes compared to Impatient's ~73%, (4) Slice-and-Dice operates on all uniform grid points in parallel for both the forward and adjoint operators, while Impatient operates serially on tile–bin pairs, and (5) Slice-and-Dice GPU utilizes

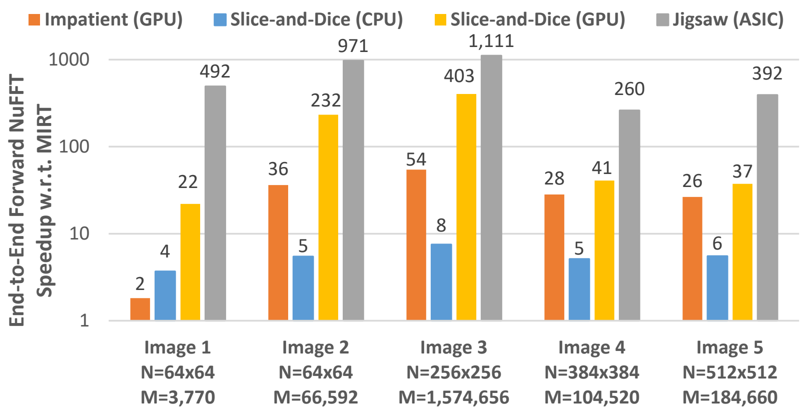


Figure 5.12: End-to-end forward NuFFT speedups of Slice-and-Dice CPU, Slice-and-Dice GPU, JIGSAW, and Impatient [12], normalized to the Michigan Image Reconstruction Toolbox (MIRT). The grid size N and number of non-uniform samples M for each image are labeled; $\sigma = 2$ and $W = 4$ for all images.

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, Slice-and-Dice's ASIC implementation for 2D NuFFT acceleration, comprises a set of 8×8 pipelines with a reconfigurable datapath to support the gridding, regridding, FFT2, and IFFT2 operations required for iterative MRI reconstructions. Using a hybrid fixed- and floating-point pipeline with a custom memory subsystem to provide fixed read and write latencies, JIGSAW is fully pipelined and does not experience any hardware stalls. In the gridding and regridding modes, JIGSAW accepts a new non-uniform sample each cycle, passing the sample from one pipeline to the next. 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. Pipeline depth varies based on the operating mode. For regridding, a total pipeline depth of 1032 cycles and the synthesized 1.0 GHz clock speed guarantees that the runtime of an M-sample input is $M + 1032$ cycles, or $(M + 1032)$ ns. The pipeline depth for gridding is 83 cycles, resulting in a runtime of $M + 83$ cycles.

With a streaming architecture and no memory or computational stalls, JIGSAW offers

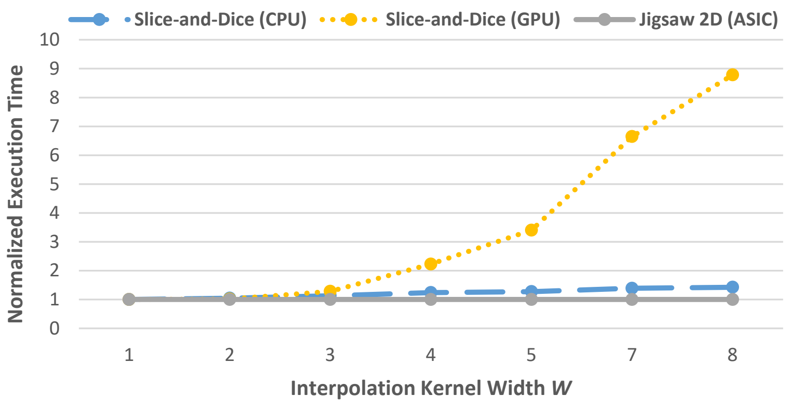


Figure 5.13: Effects of interpolation kernel width on performance of CPU, GPU, and ASIC Slice-and-Dice gridding implementations. Each implementation is normalized to itself, showcasing the increase in execution time with increasing window width.

orders of magnitude better performance than all prior works. Figures 5.9 and 5.11 demonstrate average gridding speedups of over 1000× relative to the baseline for both gridding and regridding operations. End-to-end NuFFT performance is also drastically increased in Figures 5.10 and 5.12, with speedups of over 558× compared to the baseline across both operations, and 58× and 29× for each the adjoint and forward NuFFTs versus the prior work, respectively.

5.4.2 Varying Interpolation Window Width

The interpolation window width, W, determines how many uniform grid points affect (or are affected by) each non-uniform sample. As described in [1], reducing the grid oversampling factor σ is desirable to reduce the memory constraints of the NuFFT. However, a reduction in σ requires an associated increase in W to keep the error constant, which in turn increases computation time in traditional systems. To demonstrate the effects of interpolation window width on Slice-and-Dice's performance, Figure 5.13 shows each variants' gridding performance for a range of widths, where lower values indicate less sensitivity.

With its fully-provisioned parallel pipeline structure, JIGSAW offers constant-time performance for all supported interpolation window widths. In comparison, the GPU variant experiences an 8.79× increase in execution time over the range of widths, while the CPU

Table 5.2: JIGSAW Synthesis Results in 16 nm Technology.

JIGSAW with 1.0 GHz Clock	Power	Area
Gridding w/ Uniform Grid 8MB SRAM	145 mW	13.0 mm^2
Gridding w/o Uniform Grid SRAM	84 mW	0.93 mm^2
Regridding w/ Uniform Grid 8MB SRAM	116 mW	13.0 mm^2
Regridding w/o Uniform Grid SRAM	56 mW	0.93 mm^2
FFT2/IFFT2 w/ Uniform Grid 8MB SRAM	181 mW	13.0 mm^2
FFT2/IFFT2 w/o Uniform Grid SRAM	50 mW	0.93 mm^2

variant experiences only a 1.42× increase. Although the occupancy increases from 82% to 94% over the tested range, the GPU variant suffers from lower streaming multiprocessor utilization—dropping from 64% with a window width of 1 to only 8% with a window width of 8. The CPU variant, which was already bottlenecked due to the low thread count, experiences far less sensitivity to the interpolation window width with IPC decreasing from 2.2 to only 1.4 over the range.

5.4.3 Power & Area

Table 5.2 reports our synthesis results. While the majority of the functional units are shared in each operating mode, the switching activity levels in each vary. Synthesis results both with and without uniform grid SRAM are given for each operational mode to illustrate the power required for just the combinational logic and lookup tables. JIGSAW has an estimated power consumption of up to 181 mW and an area of 12.97 mm^2. Approximately 93% of this area is used for the on-chip storage of the 1024×1024 uniform grid, which is also responsible for a significant portion of the power consumption across all operational modes—ranging from 42% for the gridding mode to 72% in the FFT2/IFFT2 modes. The ~30% difference is a result of the different activity factors for each module. In the gridding and regridding modes, the uniform grid SRAM is only accessed for a subset of the total number of samples (if the pipeline is affected by or affects a sample), while in the FFT2 and IFFT2 modes the SRAM is accessed twice in each cycle for the butterfly operations.

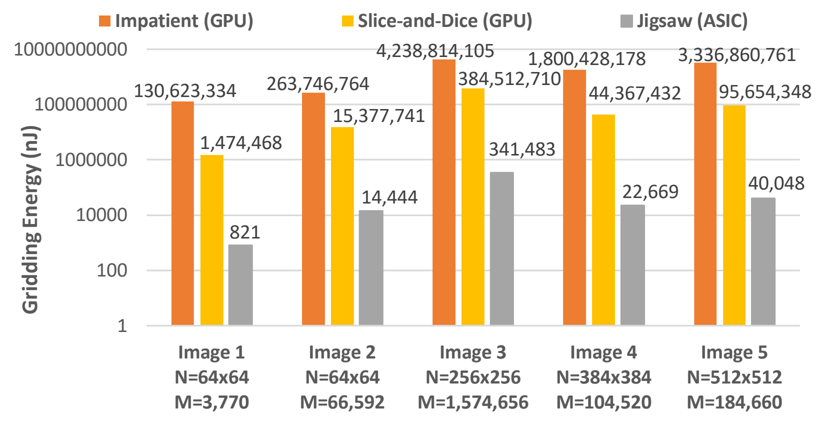
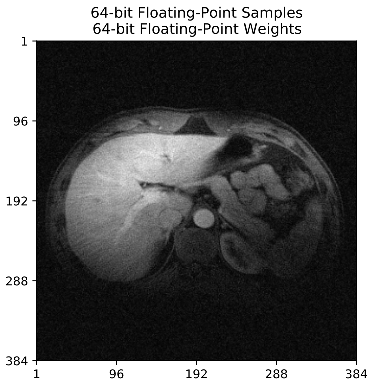


Figure 5.14: Energy requirements of Slice-and-Dice GPU, JIGSAW, and Impatient [12] gridding implementations across images.

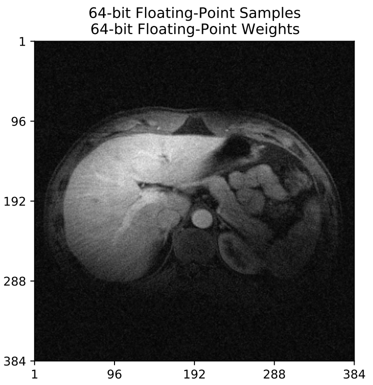
To showcase the energy efficiency achieved by JIGSAW, we compare it to the single-precision floating-point GPU gridding implementations of Slice-and-Dice and Impatient in Figure 5.14. Across all of the images tested, Impatient energy consumption averages 1.95 J, while Slice-and-Dice GPU averages 108.27 mJ. In contrast, JIGSAW consumes only 60.71 μJ—an energy reduction of over 32000× compared to Impatient and nearly 1800× compared to Slice-and-Dice GPU.

5.4.4 Image Quality

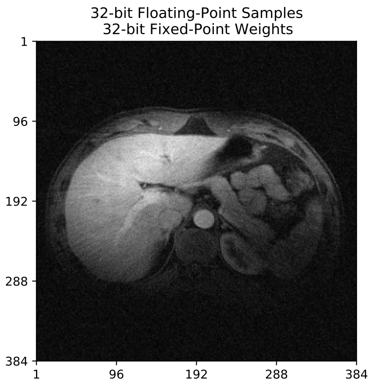
We verify JIGSAW image quality using visual comparison of 2D brain slices from [40] and their associated normalized root mean square error (NRMSE). Using our SystemVerilog implementation to simulate the hardware, we feed the non-uniform sample data into the JIGSAW pipeline, comparing the final output grid to that of our Matlab reference implementation, which uses doubles. As shown in Figures 5.15a–5.15d, our fixed-point weight 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 percentages in Figure 5.15e indicate significantly lower normalized error for our fixed-point implementation relative to a single-precision floating point implementation—$1/4$ the error while halving the ALU width and table storage requirements. These results demon-



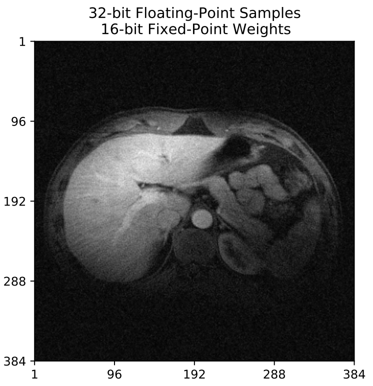
(a) $L = 1024$, Doubles



(b) $L = 32$, Doubles



(c) $L = 32$, 32-bit Fixed-Point



(d) $L = 32$, 16-bit Fixed-Point

Weight Precision	NRMSE %
32-bit Floating-Point	0.047
32-bit Fixed-Point	0.012
16-bit Fixed-Point	0.014

(e) $L = 32$, NRMSE percentages vs Doubles

Figure 5.15: Direct NuFFT reconstructions using various table oversampling factors and numeric precision. Numeric precision is given for the individual real and imaginary complex components; i.e., two doubles are used to store each complex table value in Figures 5.15a and 5.15b. Image data from [40].

strate that using JIGSAW's custom-precision hardware implementation results in negligible image quality degradation compared to the reference.

5.5 Conclusion

In this chapter, we described JIGSAW, a streaming accelerator implementation of Slice-and-Dice that performs non-uniform interpolations in runtime proportional to the number of non-uniform samples—irrespective of uniform target grid size, interpolation kernel width, or sampling pattern. Our reconfigurable datapath, paired with 32-bit floating-point data and 16-bit fixed-point weights, achieves orders of magnitude better performance and energy efficiency versus the prior work.

Our evaluation of JIGSAW demonstrates average non-uniform interpolation speedups of over 1058× and 60× when compared to the CPU baseline and state-of-the-art implementations, respectively. JIGSAW processes eight FFT or IFFT butterfly operations in parallel, enabling end-to-end forward and adjoint NuFFT speedups of 558× the CPU baseline and 41× speedups versus the prior work—resulting in the FFT being the computational bottleneck for the first time. Synthesized in 16 nm technology, JIGSAW achieves a 1.0 GHz clock frequency with an area of ~13 mm^2 and a power consumption of less than ~181 mW across all operational modes. JIGSAW's fixed-point streaming implementation is nearly 1800× more power efficient than Slice-and-Dice GPU.

CHAPTER VI

Conclusion

Computational imaging modalities continue to be widely used by medical professionals as safe, non-invasive diagnostic methods. As these modalities trend towards higher resolution, 3D capabilities, and faster scan times, emerging algorithms put tremendous strain on traditional hardware architectures, which are not optimized for the unique memory access and computational patterns found in medical image reconstruction. To enable state-of-the-art algorithms and handheld imaging systems, specialized hardware architectures are required to meet the stringent bandwidth, power, and latency requirements.

In this dissertation we described our work on 3D ultrasound, focusing on high-volume-rate imaging using the plane-wave algorithm. We first introduced Delay Compression, an algorithmic optimization for 3D plane-wave ultrasound which enables unprecedented decreases in computational complexity by sharing a single set of delay constants between all scanlines, achieving a 1024× reduction in the number of uniquely calculated delays. We then partnered Delay Compression with the TETRIS RESERVATION STATION (RS), a 2D register file which buffers incomplete voxels and eliminates the need to store the entire receive signal on-chip. These form the core of TETRIS, our custom accelerator architecture for 3D-separable plane-wave ultrasound capable of processing data streams in a single, on-the-fly pass—enabling volume generation at rates limited only by the physical propagation speed of sound in human tissue. Together, Delay Compression and TETRIS demonstrate the power of using hardware/software co-design to accelerate state-of-the-art algorithms in fields requiring immense computational bandwidth in an ultra-low-power package.

We then reviewed our work on MRI reconstruction using the forward and adjoint Non-uniform Fast Fourier Transform (NuFFT). By using our novel processing model, Slice-and-Dice, we achieved non-uniform interpolation speedups averaging over 250× the baseline on traditional architectures—an operation which previously consumed 99.6% of the NuFFT's computation time. To demonstrate the full potential of Slice-and-Dice for emerging iterative and real-time MRI reconstruction algorithms, we then described JIGSAW, a streaming accelerator architecture optimized for the non-uniform interpolation and 2D FFT/IFFT operations found in the NuFFT. By pairing Slice-and-Dice with a custom memory subsystem and processing pipeline, JIGSAW performs non-uniform interpolations with a computation time linear in the number of non-uniform samples, irrespective of the non-uniform sampling pattern, interpolation kernel width, or uniform grid size. JIGSAW achieves non-uniform interpolation and end-to-end speedups averaging over 1000× and 558×, respectively, when compared to the baseline across all problems sizes tested.

With these algorithmic optimizations and hardware architectures as a guide, we believe that architects and developers can usher in a new era of performant, power-efficient computational imaging algorithms and modalities to enable safer, faster, and more accurate clinical diagnoses.

6.1 Future Directions

While the algorithmic optimizations and hardware architectures presented in this dissertation provide orders of magnitude improvements in terms of performance and power efficiency on both commodity hardware accelerators and custom ASIC solutions, this work only represents a small fraction of the potential optimizations we believe are available.

In Section II we described our work on algorithmic optimizations for high-volume-rate 3D ultrasound. While Delay Compression enables significant reductions in computational complexity for ultrasound algorithms incorporating planar transmission schemes, the two-part decomposition as currently formulated is restricted to algorithms of this type. We hope that future algorithmic works will continue looking for similar optimizations, as they open the doors to new architectural approaches. Future work should focus on finding additional applications for Delay Compression and exploring similar symmetries in spherical transmission schemes.

On the hardware side, in Section V we described JIGSAW, our streaming accelerator architecture for NuFFT processing. The architecture discussed relies heavily on on-chip SRAM to buffer all of the “in-flight” values and provide fast, fixed-latency memory access, which can be difficult to map to semi-custom platforms such as FPGAs. JIGSAW in particular suffers from this problem as dimensionality grows beyond 2D, as the limited on-chip memory storage determines the maximum image dimension that can be supported. Substituting the on-chip storage with a load/store queue-like structure capable of handling variable-latency accesses to external memories will enable support of larger image sizes and dimensions, such as those found in “full” 3D reconstructions (i.e., not performed as a series of 2D slices).

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] P. J. Beatty, D. G. Nishimura, and J. M. Pauly. Rapid gridding reconstruction with a minimal oversampling ratio. *IEEE Transactions on Medical Imaging*, 24(6):799–808, June 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*, 64(7):1853–1864, July 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 *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sep. 2014.

[4] R.S.C. Cobbold. *Foundations of Biomedical Ultrasound*. Biomedical Engineering Series. Oxford University Press, 2007.

[5] J.J. Dahl, G.E. Trahey, and G.F. Pinton. The effects of image degradation on ultrasound-guided HIFU. In *Proc. of IEEE International Ultrasonics Symp.*, pages 809–812, Oct. 2010.

[6] J. A. Fessler. Model-Based Image Reconstruction for MRI. *IEEE Sig. Proc. Mag.*, 27(4):81–9, July 2010.

[7] J. A. Fessler and B. P. Sutton. Nonuniform fast Fourier transforms using min-max interpolation. *IEEE Trans. Sig. Proc.*, 51(2):560–74, February 2003.

[8] Jeffrey A Fessler. Michigan Image Reconstruction Toolbox (MIRT).

[9] Jeffrey A. Fessler. On NUFFT-based gridding for non-Cartesian MRI. *Journal of Magnetic Resonance*, 188(2):191 – 195, 2007.

[10] Jens Frahm, Dirk Voit, and Martin Uecker. Real-Time Magnetic Resonance Imaging: Radial Gradient-Echo Sequences With Nonlinear Inverse Reconstruction. *Investigative Radiology*, 54:1, 07 2019.

[11] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] Jiading Gai, Nady Obeid, Joseph Holtrop, Xiao-Long Wu, Fan Lam, Maojing Fu, Justin Haldar, Wen-mei Hwu, Zhi-Pei Liang, and Brad Sutton. More IMPATIENT: A Gridding-Accelerated Toeplitz-based Strategy for Non-Cartesian High-Resolution 3D MRI on GPUs. *Journal of parallel and distributed computing*, 73:686–697, 05 2013.

[13] Breaking the MRI Sound Barrier, Jul 2018.

[14] P. A. Hager, A. Bartolini, and L. Benini. Ekho: A 30.3W, 10k-Channel Fully Digital Integrated 3-D Beamformer for Medical Ultrasound Imaging Achieving 298M Focal Points per Second. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(5):1936–1949, May 2016.

[15] P. A. Hager, P. Vogel, A. Bartolini, and L. Benini. Assessing the area/power/performance tradeoffs for an integrated fully-digital, large-scale 3D-ultrasound beamformer. In *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*, pages 228–231, Oct 2014.

[16] F. Hamzaoglu, U. Arslan, N. Bisnik, S. Ghosh, M. B. Lal, N. Lindert, M. Meterelliyoz, R. B. Osborne, J. Park, S. Tomishima, Y. Wang, and K. Zhang. A 1 Gb 2 GHz 128 GB/s Bandwidth Embedded DRAM in 22 nm Tri-Gate CMOS Technology. *IEEE Journal of Solid-State Circuits*, 50(1):150–157, Jan 2015.

[17] 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 Transactions on Antennas and Propagation*, 60(5):2597–2602, May 2012.

[18] A. Ibrahim, P. Hager, A. Bartolini, F. Angiolini, M. Arditi, L. Benini, and G. De Micheli. Tackling the bottleneck of delay tables in 3D ultrasound imaging. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1683–1688, March 2015.

[19] A. Ibrahim, P. A. Hager, A. Bartolini, F. Angiolini, M. Arditi, J. Thiran, L. Benini, and G. De Micheli. Efficient Sample Delay Calculation for 2-D and 3-D Ultrasound Imaging. *IEEE Transactions on Biomedical Circuits and Systems*, 11(4):815–831, Aug 2017.

[20] J. A. Jensen. FIELD: A program for simulating ultrasound systems. *10th Nordicbaltic Conference on Biomedical Imaging, Vol. 4, Supplement 1, Part 1:351–353*, pages 351–353, 1996.

[21] J. A. Jensen. Simulation of advanced ultrasound systems using Field II. In *2004 2nd IEEE International Symposium on Biomedical Imaging: Nano to Macro (IEEE Cat No. 04EX821)*, pages 636–639 Vol. 1, April 2004.

[22] J. A. Jensen and N. B. Svendsen. Calculation of pressure fields from arbitrarily shaped, apodized, and excited ultrasound transducers. *IEEE Transactions on Ultrasonics Ferroelectrics and Frequency Control*, 39(2):262–267, Mar. 1992.

[23] Jørgen Jensen, S.I. Nikolov, Alfred Yu, and Damien Garcia. Ultrasound Vector Flow Imaging—Part II: Parallel Systems. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 63(11):1722–1732, Nov 2016.

[24] K. O. Johnson and J. G. Pipe. Convolution kernel design and efficient algorithm for sampling density correction. *Mag. Res. Med.*, 61(2):439–47, February 2009.

[25] Steven G. Johnson and Matteo Frigo. Implementing FFTs in Practice. In C. Sidney Burrus, editor, *Fast Fourier Transforms*, chapter 11. Connexions, Rice University, Houston TX, September 2008.

[26] 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. *A&A*, 586:A76, 2016.

[27] 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 *2011 IEEE RadarCon (RADAR)*, pages 592–596, May 2011.

[28] 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 *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 449–460, May 2012.

[29] Kerem Karadayi, Cheoljin Lee, and Yongmin Kim. Software-based Ultrasound Beamforming on Multi-core DSPs. Technical report, University of Washington, March 2011. `http://www.ti.com/lit/wp/sprabo0/sprabo0.pdf`.

[30] Jens Keiner, Stefan Kunis, and Daniel Potts. Using NFFT 3—A Software Library for Various Nonequispaced Fast Fourier Transforms. *ACM Trans. Math. Softw.*, 36(4), August 2009.

[31] 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 *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 585–590, June 2011.

[32] S. Kestur, S. Park, K. M. Irick, and V. Narayanan. Accelerating the Nonuniform Fast Fourier Transform Using FPGAs. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 19–26, May 2010.

[33] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA, 1997.

[34] B. Lam, M. Price, and A. P. Chandrakasan. An ASIC for Energy-Scalable, Low-Power Digital Ultrasound Beamforming. In *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 57–62, Oct 2016.

[35] C. Liu, M. Huang, and Y. Tu. A 12 bit 100 MS/s SAR-Assisted Digital-Slope ADC. *IEEE Journal of Solid-State Circuits*, 51(12):2941–2950, Dec 2016.

[36] G. Madey, X. Xiang, S. E. Cabaniss, and Y. Huang. Agent-based scientific simulation. *Computing in Science & Engineering*, 2(01):22–29, jan 2005.

[37] G. Matrone, A. S. Savoia, G. Caliano, and G. Magenes. Ultrasound plane-wave imaging with delay multiply and sum beamforming and coherent compounding. In *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 3223–3226, Aug 2016.

[38] Nadia M Obeid, Ian C. Atkinson, Keith R. Thulborn, and W-M. W. Hwu. GPU-Accelerated Gridding for Rapid Reconstruction of Non-Cartesian MRI. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, 2010.

[39] 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*, 25(5):582–589, May 2006.

[40] Ricardo Otazo, Emmanuel Candès, and Daniel K. Sodickson. Low-rank plus sparse matrix decomposition for accelerated dynamic MRI with separation of background and dynamic components. *Magnetic Resonance in Medicine*, 73(3):1125–1136, 2015.

[41] M. S. Patterson and F. S. Foster. The improvement and quantitative assessment of B-mode images produced by an annular array/cone hybrid. *Ultrasonic Imaging*, 5(3):195–213, 1983. PMID: 6356553.

[42] Jerry L. Prince and Jonathan M. Links. *Medical Imaging: Signals and Systems*. Pearson Prentice Hall, 2006.

[43] Christopher I. Rodrigues, David J. Hardy, John E. Stone, Klaus Schulten, and Wen-Mei W. Hwu. GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, page 273–282, New York, NY, USA, 2008. Association for Computing Machinery.

[44] R. Sampson, M. Yang, S. Wei, C. Chakrabarti, and T. F. Wenisch. Sonic Millip3De: A massively parallel 3D-stacked accelerator for 3D ultrasound. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 318–329, Feb 2013.

[45] R. Sampson, M. Yang, S. Wei, C. Chakrabarti, and T. F. Wenisch. Sonic Millip3De with dynamic receive focusing and apodization optimization. In *2013 IEEE International Ultrasonics Symposium (IUS)*, pages 557–560, July 2013.

[46] W. Simon, A. C. Yüzügüler, A. Ibrahim, F. Angiolini, M. Arditi, J.-P. Thiran, and G. De Micheli. Single-FPGA, scalable, low-power, and high-quality 3D ultrasound beamformer. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–2, Aug 2016.

[47] 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*, 27(4):538–547, April 2008.

[48] Stergios Stergiopoulos, editor. *Advanced Signal Processing: Theory and Implementation for Sonar, Radar, and Non-Invasive Medical Diagnostic Systems*. The Electrical Engineering and Applied Signal Processing Series. CRC Press, 2009.

[49] Brad Sutton, Douglas Noll, and Jeff Fessler. Fast, iterative image reconstruction for MRI in the presence of field inhomogeneities. *IEEE transactions on medical imaging*, 22:178–88, 03 2003.

[50] M. Tanter and M. Fink. Ultrafast imaging in biomedical ultrasound. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 61(1):102–119, January 2014.

[51] Texas Instruments. C6678 Power Consumption Model (Rev. D), Feb. 2014. `http://www.ti.com/lit/zip/sprm545`.

[52] Texas Instruments. TMS320C6678 Datasheet (Rev. E), Mar. 2014. `http://www.ti.com/lit/ds/symlink/tms320c6678.pdf`.

[53] MRI scan: How it's performed, Aug 2018.

[54] K. F. Üstüner and G L Holley. Ultrasound Imaging System Performance Assessment. American Assocation of Physicists in Medicine Annu. Meeting, 2003.

[55] S. Wei, M. Yang, J. Zhou, R. Sampson, O. D. Kripfgans, J. B. Fowlkes, T. F. Wenisch, and C. Chakrabarti. Low-Cost 3-D Flow Estimation of Blood With Clutter. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 64(5):772–784, May 2017.

[56] S. Wenger, M. Magnor, Y. Pihlström, S. Bhatnagar, and U. Rau. SparseRI: A Compressed Sensing Framework for Aperture Synthesis Imaging in Radio Astronomy. *Publications of the Astronomical Society of the Pacific*, 122(897):1367–1374, nov 2010.

[57] B. L West, J. F. Fessler, and T. F. Wenisch. Jigsaw: A Slice-and-Dice Approach to Non-uniform FFT Acceleration for MRI Image Reconstruction. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (to appear)*, pages 1–10, 2021.

[58] B. L. West, J. Zhou, C. Chakrabarti, and T. F. Wenisch. Delay Compression: Reducing Delay Calculation Requirements for 3D Plane-Wave Ultrasound. In *2019 IEEE International Ultrasonics Symposium (IUS)*, pages 1278–1281, 2019.

[59] B. L. West, J. Zhou, R. G. Dreslinksi, O. D. Kripfgans, J. B. Fowlkes, C. Chakrabarti, and T. F. Wenisch. Tetris: Using Software/Hardware Co-Design to Enable Handheld, Physics-Limited 3D Plane-Wave Ultrasound Imaging. *IEEE Transactions on Computers*, 69(8):1209–1220, 2020.

[60] Brendan L. West, Jian Zhou, Ronald G. Dreslinski, J. Brian Fowlkes, Oliver Kripfgans, Chaitali Chakrabarti, and Thomas F. Wenisch. Tetris: A Streaming Accelerator for Physics-Limited 3D Plane-Wave Ultrasound Imaging. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 189:1–189:6, New York, NY, USA, 2019. ACM.

[61] 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 *2011 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 69–72, March 2011.

[62] Daguang Xu, Yong Huang, and Jin U. Kang. GPU-accelerated non-uniform fast Fourier transform-based compressive sensing spectral domain optical coherence tomography. *Opt. Express*, 22(12):14871–14884, Jun 2014.

[63] M. Yang, R. Sampson, S. Wei, T. F. Wenisch, and C. Chakrabarti. Separable Beamforming For 3-D Medical Ultrasound Imaging. *IEEE Transactions on Signal Processing*, 63(2):279–290, Jan 2015.

[64] M. Yang, R. Sampson, S. Wei, T. F. Wenisch, B. Fowlkes, O. Kripfgans, and C. Chakrabarti. High volume rate, high resolution 3D plane wave imaging. In *2014 IEEE International Ultrasonics Symposium*, pages 1253–1256, Sept 2014.

[65] M. Yang, R. Sampson, T. F. Wenisch, and C. Chakrabarti. Separable beamforming for 3-D synthetic aperture ultrasound imaging. In *SiPS 2013 Proceedings*, pages 207–212, Oct 2013.

[66] Ming Yang, Richard Sampson, Siyuan Wei, Thomas F. Wenisch, and Chaitali Chakrabarti. High Frame Rate 3-D Ultrasound Imaging Using Separable Beamforming. *J. Signal Process. Syst.*, 78(1):73–84, January 2015.