2013

Using offline routing to implement a low latenc 3D FFT in a multinode FPGA system

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

USING OFFLINE ROUTING TO IMPLEMENT A LOW LATENCY 3D FFT IN A MULTINODE FPGA SYSTEM

By

BENJAMIN HUMPHRIES

B.S., Georgia Institute of Technology, 2006

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2013

Approved by

First Reader _____

Martin Herbordt, Ph.D.
Associate Professor of Computer Engineering

Second Reader_____

Ayse Coskun, Ph.D.
Assistant Professor of Computer Engineering

Third Reader _____

Douglas Densmore, Ph.D.
Assistant Professor of Computer Engineering

USING OFFLINE ROUTING TO IMPLEMENT A LOW LATENCY 3D FFT IN A MULTINODE FPGA SYSTEM

BENJAMIN HUMPHRIES

ABSTRACT

Applications that require highly parallel computing along with low latency communication due to strong scaling, such as a calculating a 3D FFT for Molecular Dynamics simulations, can be problematic for traditional high performance computing (HPC) clusters. A multinode FPGA array is a good solution for these types of problems due to the direct high speed connections and flexible internal fabric inherent in FPGAs. Offline routing uses precomputed routing information to direct packets and can avoid much of the switching and congestion communication overhead. Two architectures are explored here which show the feasibility of using offline routing techniques to reduce communication latencies in FPGA systems. The first architecture targets a single FPGA that was built for initial exploration and to show how the powerful and flexible a single FPGA can be. It attained a maximum clock frequency of 102MHz and latencies of 64us and 250 us for 3D FFT calculations of 32^3 and 64^3 data points respectively. The second architecture targets an FPGA that is intended to be the model for each node in the array. The best multinode version is based on a multilevel switching architecture. It has a maximum clock frequency of 134MHz. When scaled to a cluster, latencies project to 2.4us and 5.5us for 3D FFT calculations of 32^3 and 64^3 data points respectively. The two designs show the potential for using a single FPGA and multi-FPGA arrays for HPC applications where communication latency is critical to the application.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

1D	First Dimension
2D	Second Dimension
3D	Third Dimension
AXI	Advanced Extensible Interface
COTS	Commercial off the Shelf
FFT	Fast Fourier Transform
FIFO	First In, First Out
FPGA	Fully Programmable Gate Array
Gbps	Gigabits per Second
GHz	Gigahertz
GTH	Gigabit Transceiver H Class
GTX	Gigabit Transceiver X Class
HPC	High Performance Computing
HSIF	High Speed Interface
HSRX	High Speed Receiver
HSTX	High Speed Transmitter
IP	Intellectual Property
ISE	Integrated Software Environment
LUT	Look-up Table
Mbps	Megabit per Second
MHz	Megahertz

NOP	No Operation
PLL	Phase-Locked Loop
RAM	Random Access Memory
ROM	Read-Only Memory
RX	Receiver
SERDES	Serial-Deserializer
TX	Transmitter

I. INTRODUCTION

1. The Problem at a High Level

These days there are many high performance computing (HPC) clusters doing processing on topics that range from mathematical problems to weather modeling to Molecular Dynamics simulations (MD). Almost all of these applications are compute intensive and lend themselves to acceleration by non-traditional (for HPC) compute devices such as GPUs and FPGAs. A large fraction of the "top-500" supercomputers have such accelerators associated with their nodes and; perhaps even more importantly, these architectures are also becoming ubiquitous in departmental level clusters.

An important class of applications, that includes MD, is both compute intensive and requires strong scaling. That is, the problem size can not necessarily be profitably increased as more compute resources are applied. For this application class, accelerator-based clusters are problematic: most of the computation is occurring in the accelerator while the communication interface is in other parts of the node.

To attain the best performance possible on these types of applications, a fundamental modification must be made. In order to exploit the high parallelism of the application while attaining low latency, it is necessary for these accelerator clusters to avoid traditional general purpose CPU software and traditional communication networks. An ideal approach has two parts: (i) the nodes of the accelerator clusters to have direct

accelerator-to-accelerator communication and (ii) communication is made congestion-free by using precomputed (offline) routing.

2. The Problem at a Low Level

Four types of HPC clusters are being built: general purpose CPUs, GPUs, custom ASICs, and FPGAs. CPU-only and CPU/GPU clusters are ubiquitous. An ASIC-based cluster, Anton from DE Shaw, is achieving great results for MD, but in its current form is not likely to be applicable elsewhere. FPGA clusters such as the Novo-G at the University of Florida are promising, but are still mostly used for research. An objective of this work is to advance the state-of-the-art of FPGA-based clusters.

To summarize: all of these clusters allow for a high degree of parallelism. Both the general purpose CPU and CPU/GPU clusters are cost-effective, but suffer from relatively high latency communication times due to their indirect connections with other nodes. Even when high performance networks, such as Infiniband, are used, CPU and CPU/GPU clusters still suffer from latency issues; which makes them non-ideal for highly parallel, strongly scaled problems [9]. Custom ASIC nodes like the Anton processor are ideal for any application because they can be customized to the exact problem they target, however the multi-million dollar cost of custom ASICs make them impractical for most users [21].

An excellent candidate for direct accelerator-to-accelerator network with non-traditional general purpose CPU software would be an FPGA array. The software used in FPGAs resembles the short computation cycles of ASIC hardware, and the flexible

general high speed I/O connections make FPGAs ideal for highly parallel, strong scaled computing. FPGAs provide a nice middle ground between a fully custom node and a COTS microprocessor. FPGAs provide a flexible fabric that can perform the functions of any custom nodes with comparable performance [8, 30]. Additionally, FPGAs come with a low up front cost and reusability of COTS microprocessor nodes. Modern FPGAs now come with of large fabrics, many high speed connections, and specialized hard macro cells built into the fabric, such as microprocessors. FPGA vendors also provide a large array of optimized IP, similar to Intel's Math Kernel Library for microprocessors, for computation and communication functions. Most of the generated IP has the ability to be highly customized at various levels of protocol.

4. An Application for the Problem

An example application for a highly parallel, strong scaled problem is molecular dynamic simulations. These simulations have a high degree of parallelism due to the fact that there are many particles in each simulation that require force calculations that are independent of the other particles. The requirement of strong scaling is due to the fact that millions of time step iterations of each of these force calculations are required for the simulation to produce meaningful results. The 3D FFT calculation that is used to reduce the complexity of pairwise electrostatic interactions in the molecular dynamics simulation will be used as the example application to implement using offline routing in a multinode FPGA system [20, 21].

5. Contributions to the Problem

There are two major contributions to the problem of implementing a highly parallel, strongly scaled application using FPGAs. The first is to implement a 3D FFT calculation using a single FPGA. A single FPGA design provides a detailed overview of the 3D FFT application as well as another data point for results comparison. The second is to implement the infrastructure needed for a 3D FFT calculation on a multinode FPGA cluster. A multi FPGA design shows the ability of a directly connected array of FPGA nodes for this type of problem as well as a data point for results comparison. These results also provide a direction for using FPGAs to implement a molecular dynamics simulation.

6. Methods and Assumptions

Since it is unrealistic and very difficult to develop all aspects of a real system in a relatively short period of time, this design started with a series of assumptions that would enable the design to get in the interesting parts of the problem. The first assumption is that all of the data points for the 3D FFT begin in the RAMs themselves and that there is no mechanism to extract the finished data from the device. Getting large amounts of data in and out of any real system is a considerable problem itself, but is considered outside the scope of this design. The second assumption is that these designs will only be simulated, synthesized, and mapped to an FPGA due to lack of real hardware. The simulation is used to check the numerical and architectural accuracy of the design; while the synthesis and mapping will check the spatial and timing requirements of a design. For architectural modeling on FPGAs, simulation, synthesis, and mapping provide an

acceptable level of credibility and quality for the design. Another factor that was constrained to limit the problem scope was the number of points in the design. In order to make good comparisons with other results, the sizes 32^3 and 64^3 in both fixed and floating point were selected for the computation results. These sizes tended to be good sizes for the design as they were large enough to get good parallelism, while not containing too much data to overwhelm the internal memories of the FPGA.

7. Tools

All designs used the Xilinx ISE design suite for simulation, synthesis, and mapping because it was decided early on to target Xilinx FPGAs because of the availability of the tools [22]. The ISE suite contains all of the Xilinx FPGA synthesis and targeting tools as well as the ISIM mixed language simulator and the LogiCORE IP core generator. The biggest problem with using the ISE design suite was the limitations of the ISIM simulator. The simulator did not handle simulating some of the generated IP that was used due to the IP simulation model residing in a structural format rather than a behavioral format. For example, simulations with 64^3 points and 16 IP blocks took approximately 45 minutes to simulate with partial wave dumping. Simulations with 64^3 points and 64 IP blocks would not even simulate due to machine resource limitations. These types of simulation issues led to building a fake, non-synthesizable IP block that modeled the timing of the real IP blocks in a behavioral way. This fake IP block was able to model the different widths and latencies of the real IP block, but was an order of magnitude faster to simulate so it was used in simulations for checking the general dataflow and overall timing. The prior example that used 64^3 points and 16 IP blocks ran

for about 45 minutes using the real IP took less than 5 minutes with full wave dumping when using the fake IP.

All designs targeted the Xilinx Virtex-7 xc7v2000t-lflg1925 device. This FPGA was targeted because it was a large, new device that was readily available for new designs. An FPGA with a large amount of fabric would prohibit any design from unnecessarily running into spatial restrictions due to the physical FPGA. The target FPGA also contained many high speed I/O interfaces, which would be q requirement in any multinode FPGA array system.

II. THE 3D FAST FOURIER TRANSFORM

1. The 1D FFT Overview

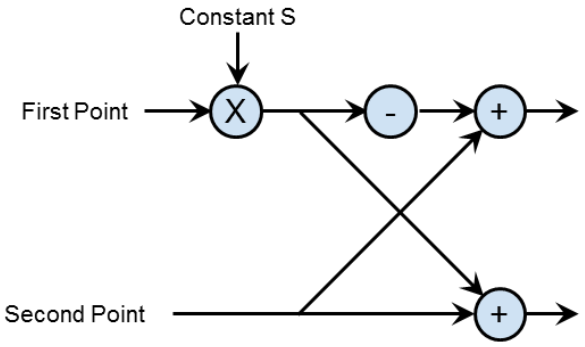

Figure 1: Diagram of FFT computation between two points.

In order to understand a 3D FFT, it is first necessary to have a rudimentary understanding of a single dimension FFT. Any 1D FFT is comprised of 2^N number of data points, and each data point is has a real and imaginary component. Each data point will undergo log N computation rounds with different points in the series. The

computation between two points, shown in Figure 1, is quite simple. For the first input point it involves multiplying the first point by a constant S and then subtracted from the second point; while the second point is just added to the first point. Each point pair will use a different value in the constant array S based on their order in the series. In between each round of computation that data is shuffled into odd/even points for the next computation, like the example shown in Figure 2. At the end of the computation rounds the data is in a disoriented pattern known as bit-reversal. So the data must be shuffled again at the end to put it back into natural order.

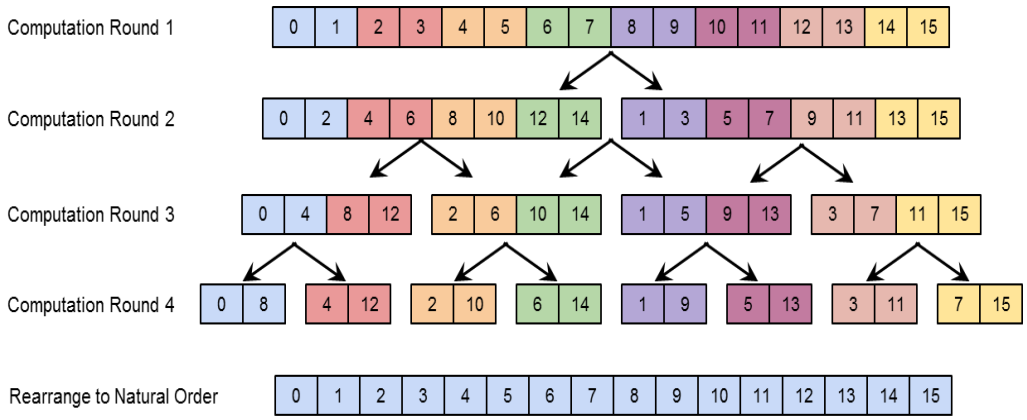

Figure 2: The butterfly pattern of a 16 point FFT calculation, showing the rearrangement of data points over each round.

2. Extending the 1D FFT into 3D

The 3D FFT data can be viewed as a cube of cubes, as seen in Figure 3. Each small cube in the larger cube is a point of data for an FFT calculation. The first step in calculating a 3D FFT is to calculate the 1D FFT on each column of data in the y dimension. Then the 1D FFT must be calculated on each row of data in the x dimensions. Finally the 1D FFT must be calculated on each deep row of data in the z

dimension. Each of these dimensions requires calculating N^2 single dimension FFTs. So in total, to fully calculate a 3D FFT on a block of NxN data there are $3N^2$ single dimension FFTs that require calculations [20].

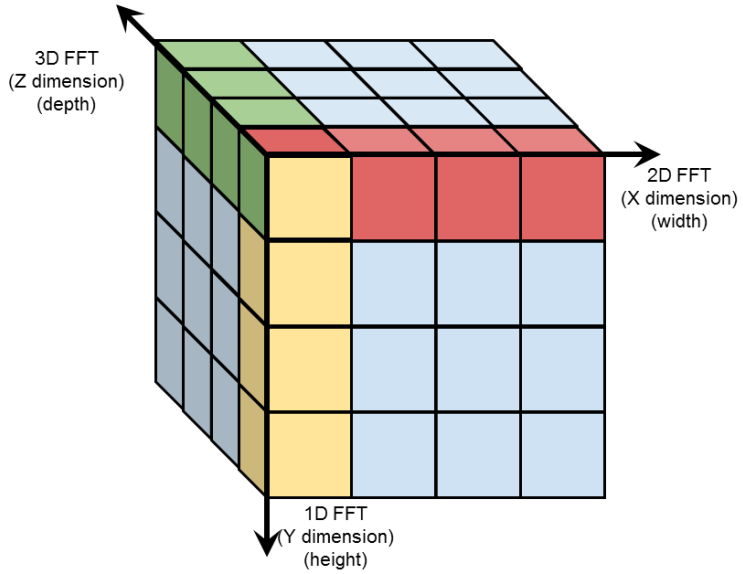

Figure 3: The cube of data points for a 3D FFT. The yellow, red, and green cubes show the points needed for 1D, 2D, and 3D FFT calculations respectively.

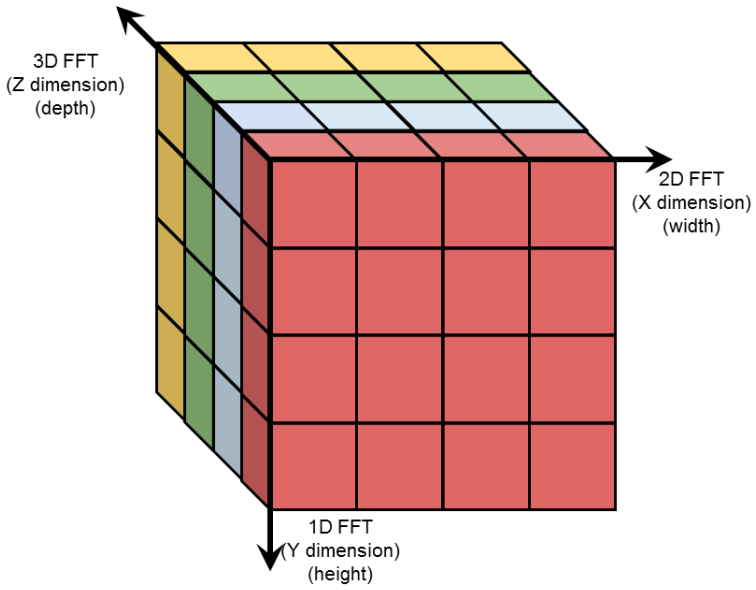

Figure 4: Example 1D/2D data division for the data points in the 3D FFT.

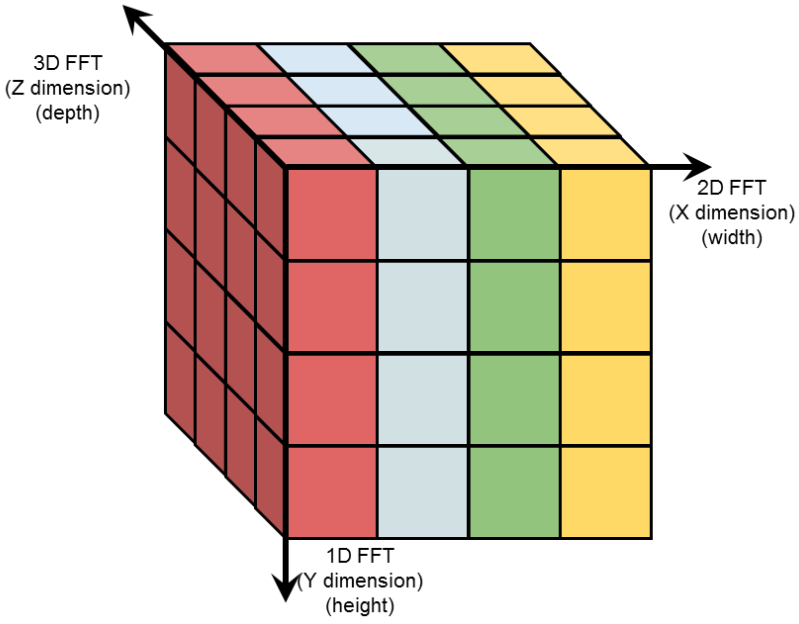

Figure 5: Example 1D/3D data division for the data points in the 3D FFT.

3. 3D Data Alignments

There are two ways to efficiently divide up the data points into planes, the 1D/2D and the 1D/3D. Figure 4 and Figure 5 show examples of 1D/2D and 1D/3D plane divisions respectively. The 1D/2D plane division gives each node one or more planes of data that it can calculate all of the 1D and 2D FFTs on. The calculation process for this data division would be that each node would calculate all of the 1D FFTs, then calculate all of the 2D FFTs, then transmit data for the 3D FFTs, then calculate all of the 3D FFTs, and finally transmit the data back to the original nodes. The problem with a 1D/2D plane division is that all of the data transmission is clumped at the end of the process, which makes the latencies due to data transmission difficult to hide behind the computation latencies. The 1D/3D plane division gives each node one or more planes of data that it can calculate all of the 1D and 3D FFTs on. The calculation process for this data division would be that each node would calculate all of the 1D FFTs, then transmit data for the 2D FFTs, then calculate all of the 2D FFTs, then transmit data back to the original nodes, and finally calculate all of the 3D FFTs. In contrast to the 1D/2D, the 1D/3D plane division evenly interlaces the computations with the transmissions, allowing for better success with hiding data transmission latencies.

It is worth noting that the 2D/3D data plane division is also another viable way of dividing up the data. This data alignment has the problem that the data has to either by realigned as the first operation of the system or it would have to be pre-aligned prior to the initialization of the system and loaded in a pre-aligned fashion. Realignment of data as the first operation of the system would be inefficient because data communication is

already the slowest part of the system and this would prohibit the parallelization of the first round of data communication with any computation. The pre-alignment of the data off-line would make the system itself faster at the expensive of the host driving the system. Given that the purpose of the system is to accelerate a 3D FFT computation, forcing the host to do part of the work of the accelerator is counterproductive and should be avoided.

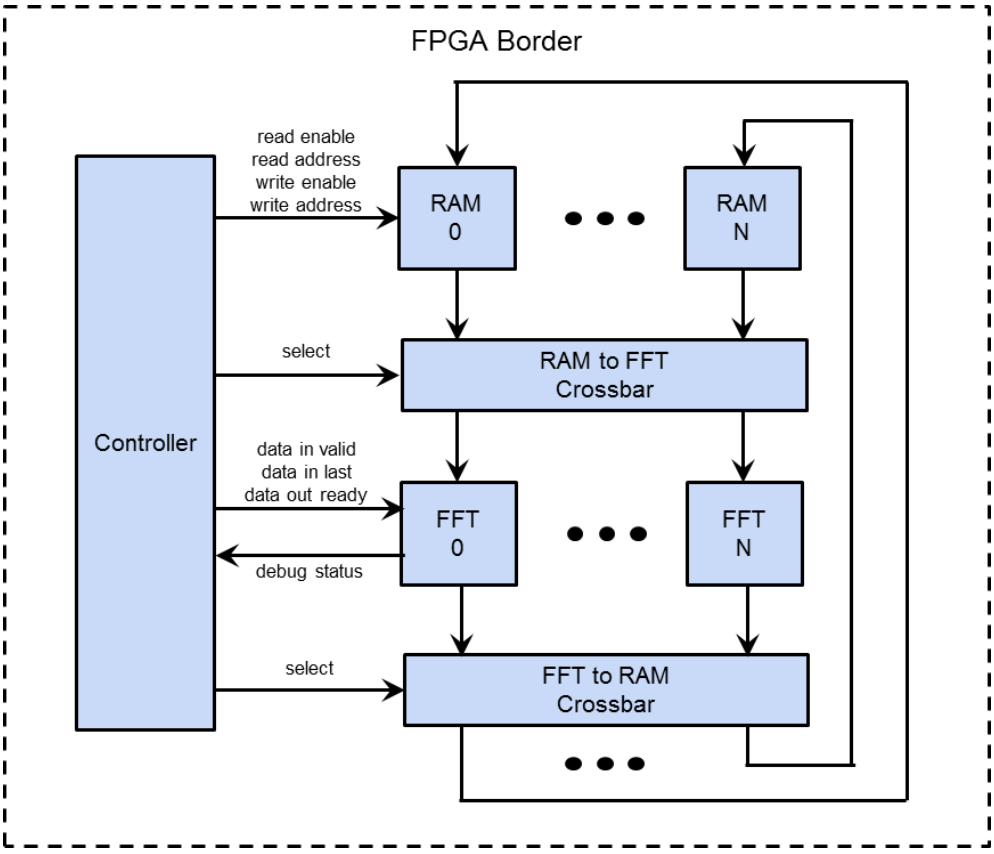

Figure 6: Top level block diagram for the single FPGA design.

III. SINGLE FPGA ARCHITECTURE

1. High Level Design

Figure 6 shows the block diagram for the 3D FFT on a single chip design. As an overview, the single FPGA design is composed of four main parts the RAMs, the Crossbars, the FFT Pipelines, and the Controller. The RAMs primary purpose is to simply store all of the data before, during, and after the computation. However they also play an important role in allowing the system to transpose data in flight. The input controls to each of the RAMs are read enable, write enable, read address, and write address. The Crossbars work in conjunction with the RAMs to select the flow of data. The purpose of the RAM to FFT Crossbar is to transpose data as it enters the FFT Pipelines, and the purpose of the FFT to RAM Crossbar is to un-transpose the data as it leaves the FFT Pipelines. The control for each Crossbar output is a mux selection to choose one of the inputs. The FFT Pipelines are the primary calculating centers for the design. Each FFT Pipeline is comprised of a single Xilinx LogiCORE IP FFT pipeline module. The FFT Pipeline has some control inputs and status outputs. The final part is Controller, which is essentially a large state machine that drives all of the inputs to the RAMs, Crossbars, and FFT Pipelines. The Controller selects the proper data at the proper time to make the massive data pipeline operate like a 3D FFT.

2. FFT Pipelines

The first step to better understanding the overall design begins with a better understanding of the FFT Pipeline. The FFT Pipelines used were generated using the

Xilinx LogiCORE IP Core generator [26]. The decision to use Xilinx generated cores over custom built cores came from the fact the implementing the FFT is a common problem for many applications, so Xilinx most likely has the optimal solution for targeting their own parts. The IP Core generator also has a very large range of parts available, so changing different parameters of the FFT itself becomes a simple regenerate process with the Core generator. Most custom built parts would require either a significant initial time investment to build a single flexible part or time later spent changing the options every time a new part is required.

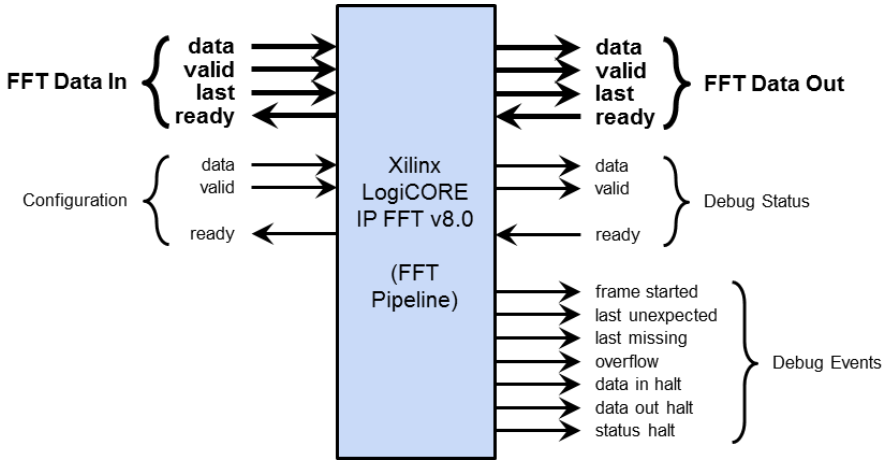

Figure 7: Interface for the Xilinx LogiCORE FFT v8.0 IP.

Having all of the flexibility that comes with using the Xilinx Core generator comes with the downside of having a massive amount of options for selecting what type of FFT to use. For this project the focus is on low latency processing, so all of the FFT cores used were Pipelined Streaming I/O with non-configurable transaction lengths and real-time throttling. The fully Pipelined Streaming I/O core versions allow for a nice even flow of data by inputting and outputting one word of data per clock. A full FFT is

calculated by clocking in all words of data, waiting a fixed number of cycles, and then clocking all words of data out. The Pipelined Streaming I/O versions allow for words from subsequent FFT frames to be input as it is calculating and outputting prior frames.

Table 1 shows the latencies for several different versions Xilinx Pipelined FFTs. All of the data widths in Table 1 define the width of a single real or imaginary part of the data; therefore the actual bit width of the pipeline is twice that of the data width listed in the table. Other than the FFT option constraints listed above, the data length, output order, and data type had the most significant impact on the FFT latencies. The output order is the order the data exits the pipeline; data is either in bit-reversed order or in the natural input order. The data length is the number of data points in a single FFT calculation. The data type consists of the data width and the data packing, fixed or floating point. The fixed point values use a constant precision phase factor of 32, scaled output, and truncated rounding mode. The floating point values use a constant precision phase factor of 24.

Table 1: Latencies for different versions of Xilinx LogiCORE FFT v8.0 IP. The highlighted versions were the versions used.

Data Length	Output Order	Latency (in cycles)			
		Fixed 16	Fixed 24	Fixed 32	Float 32
32	Reversed	131	135	135	188
	Natural	165	169	169	222
64	Reversed	201	205	205	290
	Natural	267	271	271	356
128	Reversed	336	342	342	490
	Natural	466	472	472	620
256	Reversed	598	604	604	880
	Natural	858	864	864	1140

Figure 7 shows the full details of the interface for the Xilinx LogiCORE IP FFT v8.0 [26]. Version 8.0 was selected because of the use of standard AXI Stream interfaces for the input data, output data, configuration settings, and debug status. The AXI Stream interface is comprised of four signals, data, last, valid, and ready. The master controls the data, last, and valid signals; while the slave controls the ready signal. The protocol from the master's perspective is that when ready is asserted, data will be saved by the slave when the valid signal is asserted. The last signal is used by the master to tell the slave that the current data word is the last in the frame. In the FFT Pipeline application, the last is used to indicate the final word in a single FFT data set. The data in and data out busses are driven by the Crossbar; while the valid, last, and ready controls signals are driven by Controller. The FFT Pipeline also drives several discrete event signals for the start of a new frame, unexpected last frame, missing last frame, data overflow, and several channel halt signals. These signals were used for bring-up debugging, but are not used in the final design due to the pre-computed nature of the design.

3. Mapping the Data to the RAMs

The next step is to understand the how the 3D data is mapped onto a single FPGA design. Since a single FPGA solution is essentially a one node system, all of the data obviously has to reside in that FPGA's internal RAM. This is where the RAMs from the block diagram in Figure 6 come into play. Each RAM holds an even part of the data and feeds data into the FFT Pipelines. Data is also fed into the RAMs from the output of the FFT Pipelines. In order for the flow rate of data to be equal throughout the entire design, the number of RAMs must match the number of FFT Pipelines. The data mapping

scheme for this design was chosen to be the 1D/2D because it was easier to conceptualize at the beginning of the project. Figure 8 shows an example of how the 1D/2D mapping scheme is implemented using a bank of RAMs. The actual number of address bits changes depending on the number of data points and the number of RAMs. Even though each RAM contains multiple planes of data, each 1D/2D plane of data resides entirely in the same RAM. The least significant bits of address describe the 1D data address in the x dimension. The middle bits of address then describe the 2D data address in the y dimension. If multiple planes exist in the same RAM, then the most significant bits of address describe the 3D data address in the z dimension. The exact address partitioning depends on the number of FFT data points to process and the number of FFT Pipelines in which to process them.

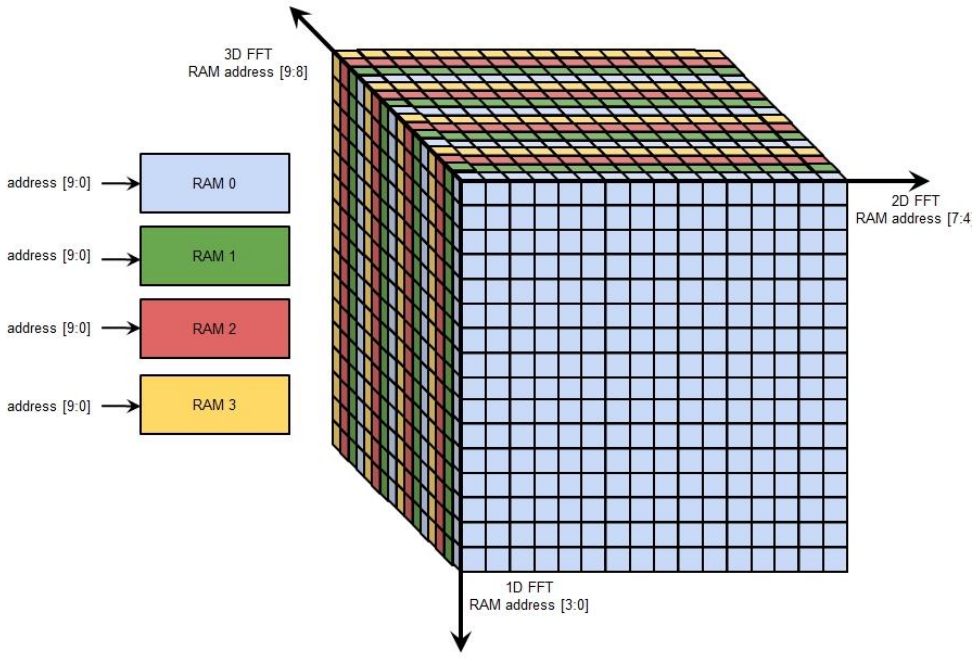

Figure 8: Mapping the 3D FFT data points to the RAMs in the single FPGA version. Each colored data point in the cube resides in the corresponding colored RAM.

Table 2 shows how the FFT sizes used and exactly how the address bits were partitioned for the RAMs. With this type of mapping, every data point now has a permanent home that can indexed with the RAM number and RAM address. The number of RAMs/FFT Pipelines and FFT size obviously dominate the latency of the entire calculation. Having more RAMs also decreases the number of data points/planes that reside in each RAM. In order to prevent stalls and bubbles from forming in the FFT Pipelines, the minimum number of data points in each RAM must be greater than the latency of the FFT Pipeline.

Table 2: A breakdown of data points and RAM address bits for various data sizes on the single FPGA design. The highlighted versions were the ones selected to implement.

FFT Size	# Data Points	# Data Points per Plane	# RAMs	# Planes per RAM	# Data Points per RAM	3D Address Bits	2D Address Bits	1D Address Bits
16	4096	256	4	4	1024	[9:8]	[7:4]	[3:0]
16	4096	256	8	2	512	[8]	[7:4]	[3:0]
16	4096	256	16	1	256	N/A	[7:4]	[3:0]
16	4096	256	32	0.5	128	N/A	[6:4]	[3:0]
16	4096	256	64	0.25	64	N/A	[5:4]	[3:0]
32	32768	1024	4	8	8192	[12:10]	[9:5]	[4:0]
32	32768	1024	8	4	4096	[11:10]	[9:5]	[4:0]
32	32768	1024	16	2	2048	[10]	[9:5]	[4:0]
32	32768	1024	32	1	1024	N/A	[9:5]	[4:0]
32	32768	1024	64	0.5	512	N/A	[9:5]	[4:0]
64	262144	4096	4	16	65536	[15:12]	[11:6]	[5:0]
64	262144	4096	8	8	32768	[14:12]	[11:6]	[5:0]
64	262144	4096	16	4	16384	[13:12]	[11:6]	[5:0]
64	262144	4096	32	2	8192	[12]	[11:6]	[5:0]
64	262144	4096	64	1	4096	N/A	[11:6]	[5:0]
128	2097152	16384	4	32	524288	[18:14]	[13:7]	[6:0]
128	2097152	16384	8	16	262144	[17:14]	[13:7]	[6:0]
128	2097152	16384	16	8	131072	[16:14]	[13:7]	[6:0]
128	2097152	16384	32	4	65536	[15:14]	[13:7]	[6:0]
128	2097152	16384	64	2	32768	[14]	[13:7]	[6:0]

4. The Crossbars

The next part to the design is the Crossbar. As seen in Figure 9, the Crossbars are simply a collection of muxes that allow any input data to be driven out of any output port. The general form of this also allows a single input to drive multiple outputs; however this feature of a Crossbar is not used in this design. The RAM to FFT Crossbar provides the capability to transpose data going into the FFT Pipeline; while the FFT to RAM Crossbar provides the capability to transpose the data going into the RAMs.

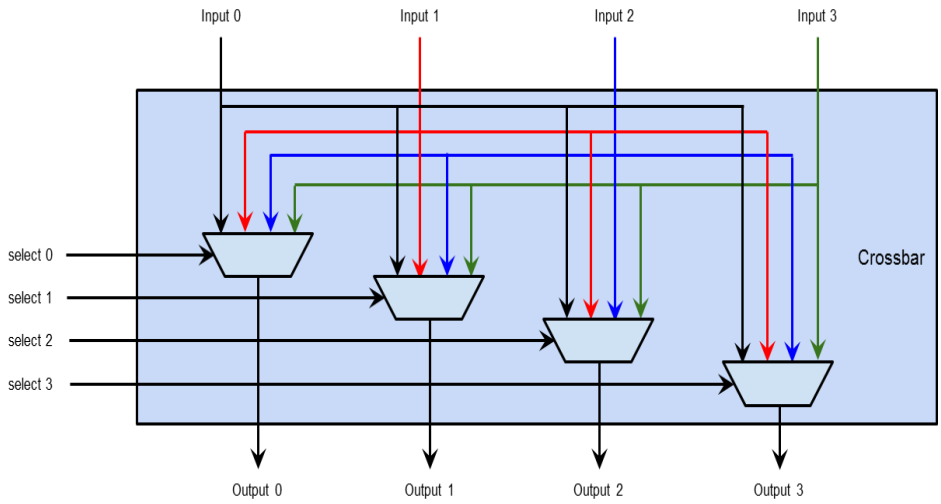

Figure 9: An example 4 input/output Crossbar.

5. The Controller

The final part to the design is the Controller. This device drives all of the control lines for the RAMs, FFT Pipelines, and Crossbars. This is where the concept of offline routing is implemented. It is easy to see that the Controller could be built as a large state machine with combinational logic controlling every step of the design, but this is painful to implement and even more painful to change later on. The solution here is to compute

the dataflow schedule for each cycle offline and just load the data into a ROM in the controller. Then when the design starts processing, the data in this Controller ROM is read out sequentially and the data from it drives the control lines for the entire system. Table 3 shows the sizes and what is encompassed in the Control ROM for this particular design. The primary factor that influences the width of the Control ROM data is the number of parallel RAMs/FFT Pipelines in the system. All of the controls, except the Crossbar selects and the done signal, scale linearly with the number of RAMs/FFT Pipelines. The Crossbar selection controls scale almost exponentially with respect to the number of RAMs/FFT Pipelines, so these controls can quickly become the dominating factor is the number of control bits in the Control ROM. The data depth of the Controller ROM is completely dependent on the number of clocks required for control, which would be included as part of the offline computations. It should also be noted that the controls shown in Table 3 show the most general form of the Controller, and given some additional constraints on the problem and design, the number of control signals could be significantly limited.

Table 3: The data that comprises a single entry in the Control ROM for different size FFTs and number of RAMs in the single FPGA design. The implemented versions are highlighted.

FFT Size	# RAMs/ FFTs	RAM Wr En	RAM Wr Adr	RAM Rd En	RAM Rd Adr	RAM to FFT Crossbar	FFT to RAM Crossbar	FFT Data Valid	FFT Data Last	Done	TOTAL Control Width
16	4	4	40	4	40	8	8	4	4	1	**113**
16	8	8	72	8	72	24	24	8	8	1	**225**
16	16	16	128	16	128	64	64	16	16	1	**449**
16	32	32	224	32	224	160	160	32	32	1	**897**
16	64	64	384	64	384	384	384	64	64	1	**1793**
32	4	4	52	4	52	8	8	4	4	1	**137**
32	**8**	**8**	**96**	**8**	**96**	**24**	**24**	**8**	**8**	**1**	**273**
32	**16**	**16**	**176**	**16**	**176**	**64**	**64**	**16**	**16**	**1**	**545**
32	32	32	320	32	320	160	160	32	32	1	**1089**
32	64	64	576	64	576	384	384	64	64	1	**2177**
64	4	4	64	4	64	8	8	4	4	1	**161**
64	8	8	120	8	120	24	24	8	8	1	**321**
64	**16**	**16**	**224**	**16**	**224**	**64**	**64**	**16**	**16**	**1**	**641**
64	**32**	**32**	**416**	**32**	**416**	**160**	**160**	**32**	**32**	**1**	**1281**
64	64	64	768	64	768	384	384	64	64	1	**2561**
128	4	4	76	4	76	8	8	4	4	1	**185**
128	8	8	144	8	144	24	24	8	8	1	**369**
128	16	16	272	16	272	64	64	16	16	1	**737**
128	32	32	512	32	512	160	160	32	32	1	**1473**
128	64	64	960	64	960	384	384	64	64	1	**2945**

6. Design Dataflow

The dataflow can be broken down even further into mirrored controls split at the FFT Pipelines. Given that a particular RAM index and RAM address is always the home of any given data point, then the controls to route data out of the FFT Pipelines are delayed mirrors of the controls to route data into the FFT Pipelines. The RAM read addresses and the RAM to FFT Crossbar selections provide the input controls, and the RAM write addresses and the FFT to RAM Crossbars selections provide the output

controls. This greatly simplifies the modeling of the dataflow to the point that only the input flow has to be modeled and the output flow will simply be the input flow delayed by the latency of the FFT Pipeline. The one caveat to this principle is that the input routing flow must ensure that the data points from the prior FFT dimension has been written back to RAM before it is read out for the current FFT dimension. This data dependency is what limits the number of FFT Pipelines in the design and hence the overall latency of 3D calculation as a whole.

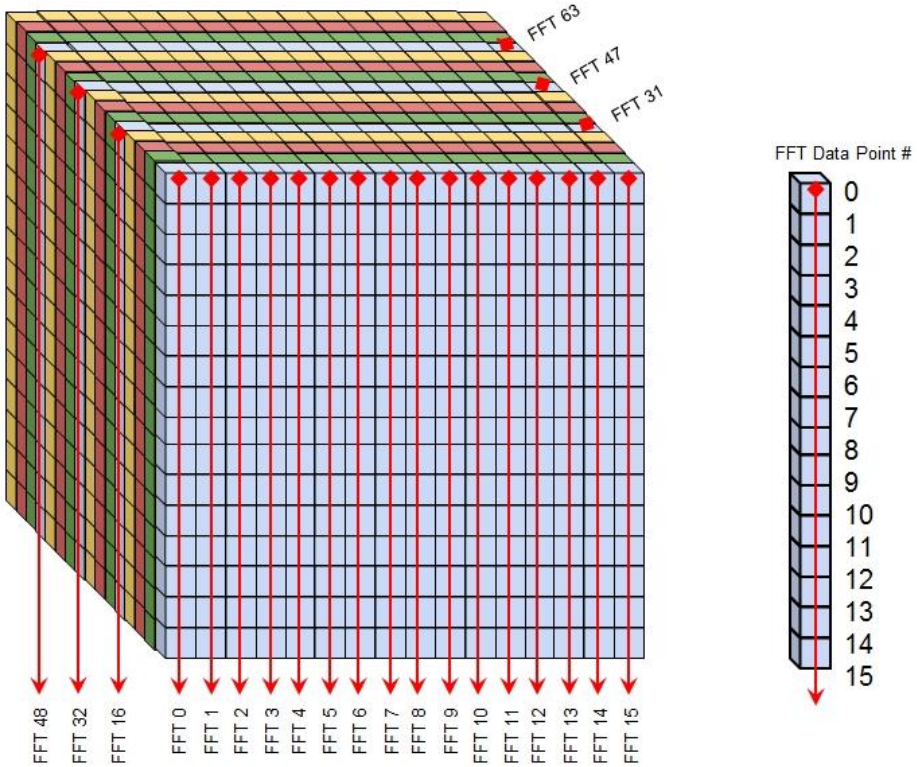

Figure 10: Traversing the 1D FFTs on the single FPGA design. This example shows a 16x16x16 cube of data points and 4 RAMs, so each FFT Pipeline would need to calculate 64 FFTs. The FFTs that will be calculated are listed as well as the order of how the data points are read into the FFT Pipeline. Accessing the RAM in this case is as simple as reading out incrementing values.

The dataflow of a single RAM address and Crossbar selector can then be broken down into three phases, one for each of the FFT dimensions. Since the data is aligned in 1D/2D planes the first and second phases both have a similar incrementing patterns for the RAM addresses and both Crossbars select the same RAM for the entire phase. The RAM pattern for the first phase is that the address will increment from the lowest bit to the highest bit. Figure 10 shows how this gives the effect of traversing a 1D column then traversing the next until all of the 1D columns are complete. The RAM pattern for the second phase is that the address will increment just like the first phase, except that the least significant 1D address bits and middle 2D address bits are swapped so that the 2D address bits get incremented sequentially and the rollover increments the 1D address bits. Figure 11 shows how this gives the effect of traversing a 2D row then traversing the next row until all of the 2D rows are complete.

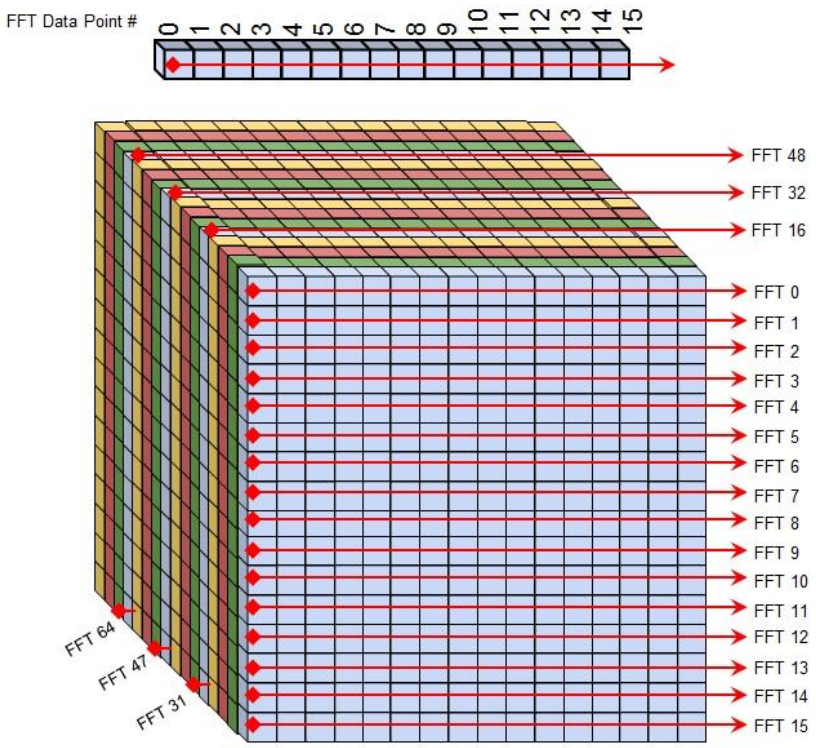

Figure 11: Traversing the 2D FFTs on the single FPGA version. This example shows a 16x16x16 cube of data points and 4 RAMs, so each FFT Pipeline would need to calculate 64 FFTs. The FFTs that will be calculated are listed as well as the order of how the data points are read into the FFT Pipeline. Accessing the RAM in this case is analogous to striding through the memory with a stride of 16.

The third phase is by far the most complicated because it requires obfuscated RAM address controls along with varying Crossbar selections. Figure 12 shows an overview of how the data needs to be traversed for 3D processing in the third phase. The third phase also imposes an additional timing requirement on the prior two phases. The reason for this is that the third phase is operating on data that spans multiple RAMs and each FFT requires data from the same RAM on the same clock cycle.

The solution to this is to skew the data driven to each FFT Pipeline so that only a single point of data is required from any particular RAM in any given cycle. When the skewing is propagated to the prior phases, it does not change the data flow control but

merely skews it by the same amount as what it is in the third phase. As far as the penalty for skewing the data, it really only adds cycles for the data to fill up and drain out; which is negligible over the course of the entire calculation.

For the third phase, the Crossbar selection for each FFT input/output will walk across all of the RAMs starting at RAM 0 and will repeat for the entire phase. The RAM address incrementing will now look obfuscated because a single RAM is no longer tied to one specific FFT Pipeline. The address pattern is that the 2D dimension will increment and whenever the value is modulo the number of FFT Pipelines the 3D dimension will increment and eventually rollover to increment the 1D dimension. The entire skewed dataflow can ensure that all of the FFT Pipelines will stay completely saturate for the entire calculation time, with the exception of the fill and drain times due to the skew.

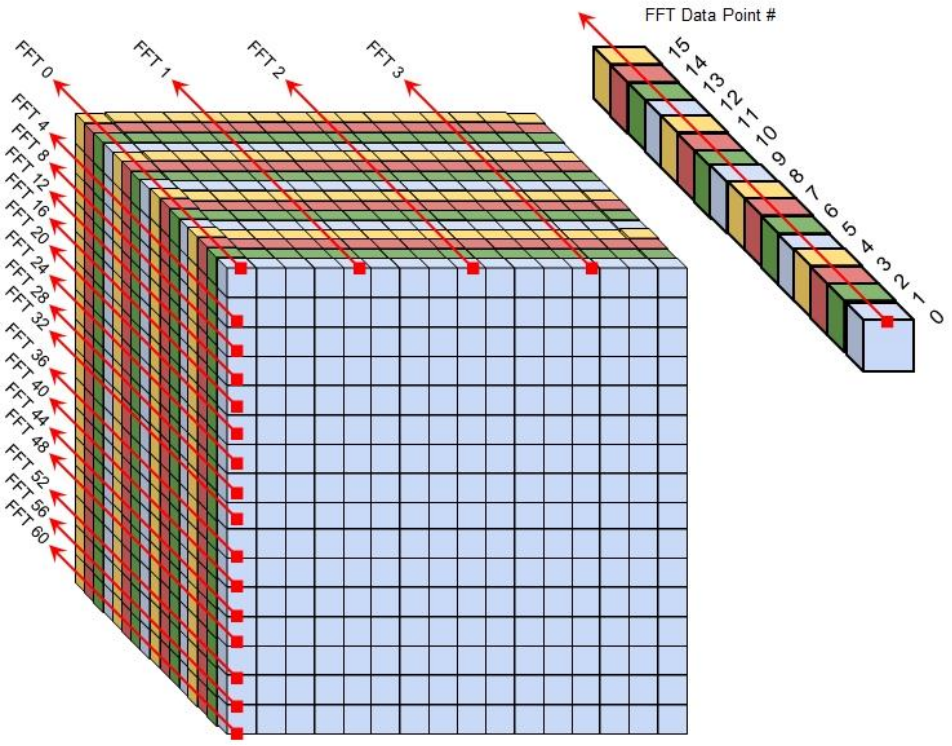

Figure 12: Traversing the 3D FFTs on the single FPGA version. This example shows a 16x16x16 cube of data points and 4 RAMs, so each FFT Pipeline would need to calculate 64 FFTs. The FFTs that will be calculated are listed as well as the order of how the data points are read into the FFT Pipeline. Accessing RAMs in this case is complicated because it crosses all of the RAM boundaries.

7. Results

There are two types of results that are extracted from the single FPGA design implementation of the 3D FFT. The first result is the cycle counts and latencies of the calculations which are shown in Table 4. Now given that the design cannot stall the pipelines and all of the routing is pre-computed before runtime, the number of clock cycles for each version of the design can be fully predicted. The versions that were actually simulated using ISE are 8 and 16 RAM counts for the 32^3 FFT size and 16 and 32 RAM counts for the 64^3 FFT size. These were selected because they were middle of the road number of RAMs in comparison to the FFT size. If too many RAMs were used

there was the potential for data to not be available, which the design could not handle. Too few RAMs would unnecessarily slow down the parallelization of the design because the number of FFT Pipelines must equal the number of RAMs.

Table 4: Cycle Counts and Latencies for the single FPGA. Highlighted versions were actually simulated.

FFT Size	# RAMs/ FFTs	Fixed 24/32 Cycles	Latency @ 50 MHz (in ns)	Latency @ 100 MHz (in ns)	Latency @ 200 MHz (in ns)	Float 32 Cycles	Latency @ 50 MHz (in ns)	Latency @ 100 MHz (in ns)	Latency @ 200 MHz (in ns)
32	4	24809	**496180**	248090	124045	24862	**497240**	248620	124310
32	8	12521	**250420**	125210	62605	12574	**251480**	125740	62870
32	16	6377	**127540**	63770	31885	6430	**128600**	64300	32150
32	32	3305	**66100**	33050	16525	3358	**67160**	33580	16790
32	64	1769	**35380**	17690	8845	1822	**36440**	18220	9110
64	4	197007	**3940140**	1970070	985035	197092	**3941840**	1970920	985460
64	8	98703	**1974060**	987030	493515	98788	**1975760**	987880	493940
64	16	49551	**991020**	495510	247755	49636	**992720**	496360	248180
64	32	24975	**499500**	249750	124875	25060	**501200**	250600	125300
64	64	12687	**253740**	126870	63435	12772	**255440**	127720	63860

The second set of results from the single FPGA design is the utilization information. Table 5 shows the design's synthesis and place and route utilization statistics for several different versions of the design. It also shows the FPGA that was the target for the synthesis and place and route. This design is intended to be the starting point for a general architecture, but a specific FPGA target was required for synthesis and place and route. So a large, full-featured FPGA was selected to prevent any unnecessary technology restrictions from limiting the design.

The primary differences in the different design versions are the FFT size, data width, data type, and number of RAMs used in the design. The slice logic utilization

columns show how the slices break down as registers and LUTs. This shows that the number of logic and register slices used only slightly increase with the number of FFT Pipelines. This means the even with the largest and fastest Xilinx FFT IP cores, they take up relatively little space. This allows future designs to be able to generously add FFT Pipelines with little concern to the spatial aspect of the FFT IP. The slice logic distribution shows the high fanout of the design by the high number of slices with an unused flip-flop and LUT. This is further reinforced by the large number of slices used in place and route for just routing. This type of sparse usage is to be expected from this design because of the low compute power and high routing effort required for the FFT. The block RAMs column of data shows that a 64^3 FFT consumes half of the RAM in the FPGA, which shows that the 64^3 FFT size is the largest that can implemented in the current single FPGA design.

The final column of the table is the timing information. The most important result from it is that something in the place and route stage of design is drastically slowing down the frequency of the design. It turns out that the Crossbar implementation causes massive fanout in the design. The multiple wide data buses running from the RAMs to the muxes and then back out to the RAMs is causing some very long timing paths due to routing limitations inside the FPGA itself. That is why this fanout effect is not seen until the place and route stage. Everything learned from the utilization information from this design can be directly propagated forward to the multi FPGA design of the 3D FFT.

Table 5: Synthesis and Place and Route results from the single FPGA version.

Design Version					Slice Logic Utilization						Slice Logic Distribution				Block RAMs		Timing	
Design Stage	Data Type	Data Width	FFT Size	# RAMs/ FFTs	% as Register	# as Register	% as LUTs	# as LUTs	# as LUTs used as Logic	# as LUTs used as Mem	# w/ unused FF	# w/ unused LUT	# w/ unused FF and LUT	# w/ only routing	%	#	Clock Min Period (ns)	Clock Max Frequency (MHz)
Syn	Float	32	32	8	1	34885	2	34897	23457	11440	7997	7985	26900	N/A	15	197	3.27	305.843
Syn	Float	32	32	16	2	69748	6	73864	50984	22880	20085	15969	53779	N/A	19	257	3.48	287.356
Syn	Fixed	32	32	8	1	26197	2	25393	16569	8824	4205	5009	21188	N/A	15	197	3.27	305.843
Syn	Fixed	32	32	16	2	52372	4	54856	37208	17648	12501	10017	42355	N/A	19	257	3.48	287.356
Syn	Float	32	64	16	3	79827	6	84679	57431	27248	21189	16337	63490	N/A	45	584	3.53	283.286
Syn	Float	32	64	32	6	159635	15	194039	139543	54496	67077	32673	126962	N/A	50	655	3.77	265.252
Syn	Fixed	32	64	16	2	62451	5	66759	44759	22000	13477	9169	53282	N/A	45	584	3.53	283.286
Syn	Fixed	32	64	32	5	124883	12	158199	114199	44000	51653	18337	106546	N/A	50	655	3.89	257.069
P&R	Float	32	32	8	1	34477	2	26143	18189	6912	3610	2569	22533	1042	15	198	10.057	99.433
P&R	Float	32	32	16	2	68924	3	58044	42432	13847	13068	4930	45031	2031	19	262	12.71	78.678
P&R	Fixed	32	32	8	1	25389	1	18394	12238	5376	2516	2059	15878	780	15	198	9.016	110.913
P&R	Fixed	32	32	16	2	50756	3	40839	28549	10770	9108	3951	31731	1520	19	262	11.394	87.765
P&R	Float	32	64	16	3	78893	6	63437	44533	16463	9565	5256	53183	1876	45	586	10.857	92.106
P&R	Float	32	64	32	6	157749	8	152480	116136	32981	43643	10087	106310	3656	50	653	13.769	72.627
P&R	Fixed	32	64	16	2	60525	3	48358	33059	13403	8064	3769	39929	1404	45	586	9.733	102.743
P&R	Fixed	32	64	32	5	121030	9	117775	87623	26852	37633	7233	79821	2736	50	653	12.736	78.518

IV. MULTI FPGA ARCHITECTURE

1. Mapping the Data to Multiple Nodes

The goal of this part of the project is to extend the single FPGA version of the 3D FFT to a multi FPGA design. The multi FPGA design will use the same assumptions as in the single FPGA version, such as pre-populated data RAMs, no input/output ports, and FFT data sizes. Assumptions about the quantity and topology of the nodes also need to be made to keep the problem constrained to a reasonable level. The number of nodes for the design will be constrained to number 64 and the topology will be arranged as a 3D torus. Given that the data sets and physical topology are in three dimensions, it is easier to think about the number of nodes in three dimensions as 4^3 instead of 64. This particular node count and topology was selected as an example of how this architecture can be used across multiple FPGAs.

The actual design architecture is not tied to any particular number of nodes or connection structure. With a multi FPGA approach the problem of the organization of the data transfers becomes much more significant than in the single FPGA version due to the limitation of the physical connections between chips. As such, it is assumed that for the 3D FFT data set and the selected topology an optimal routing pattern exists for the data flow [4, 5, 10, 11, 15, 17, 18, 19]. The actual data routing pattern is a widely published topic and is therefore not discussed here. This architecture design describes the hardware that is required to support any possible data routing pattern chosen.

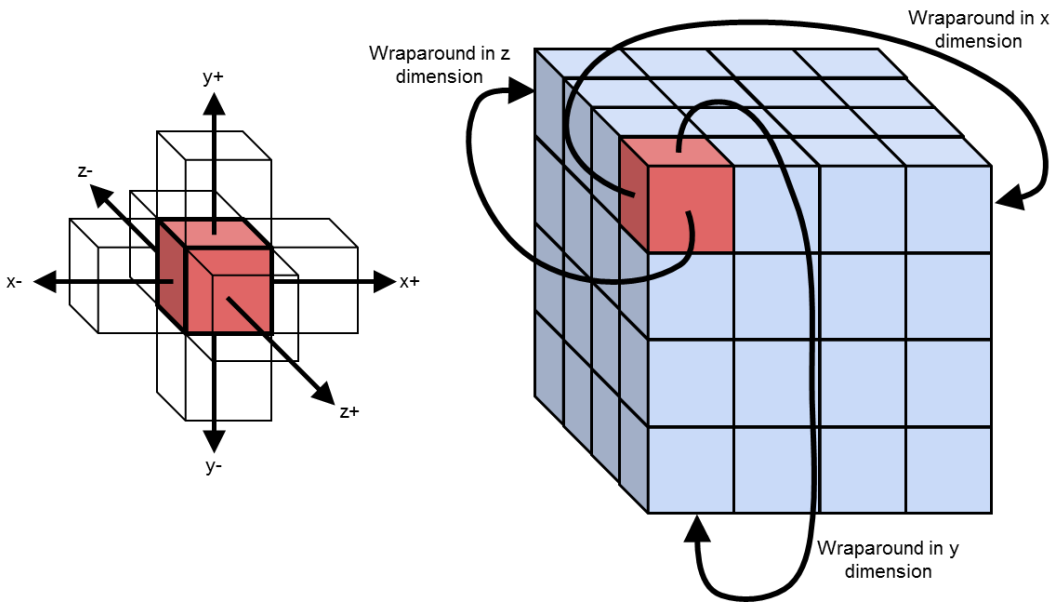

Figure 13: 3D torus connection topology. Each node can communicate with each of its 6 neighbors, which can include wrapping around the edge of the cube.

2. Multi FPGA Topology

Before getting into the actual design details it is important to consider the physical connections of the FPGAs. The ideal connection structure for any multi-node system would be for every node to have a direct connection to every other node. This however is impractical for many reasons. The most significant reason that the FPGA devices and the boards they are attached to have only a finite number of pins and even fewer high speed pins [27, 28]. For this design the 3D torus connection pattern as shown in Figure 13 has been selected in an attempt to take advantage of the spatial similarities of the 3D FFT data. This means that every node will be able to transmit and receive data from 6 other nodes. This gives each node the concept of transmitting/receiving data to/from the x +/- dimension, y +/- dimension, and z +/- dimension. It also gives the edges of the cube of nodes the ability to communicate to nodes around the edge of the cube [19, 20].

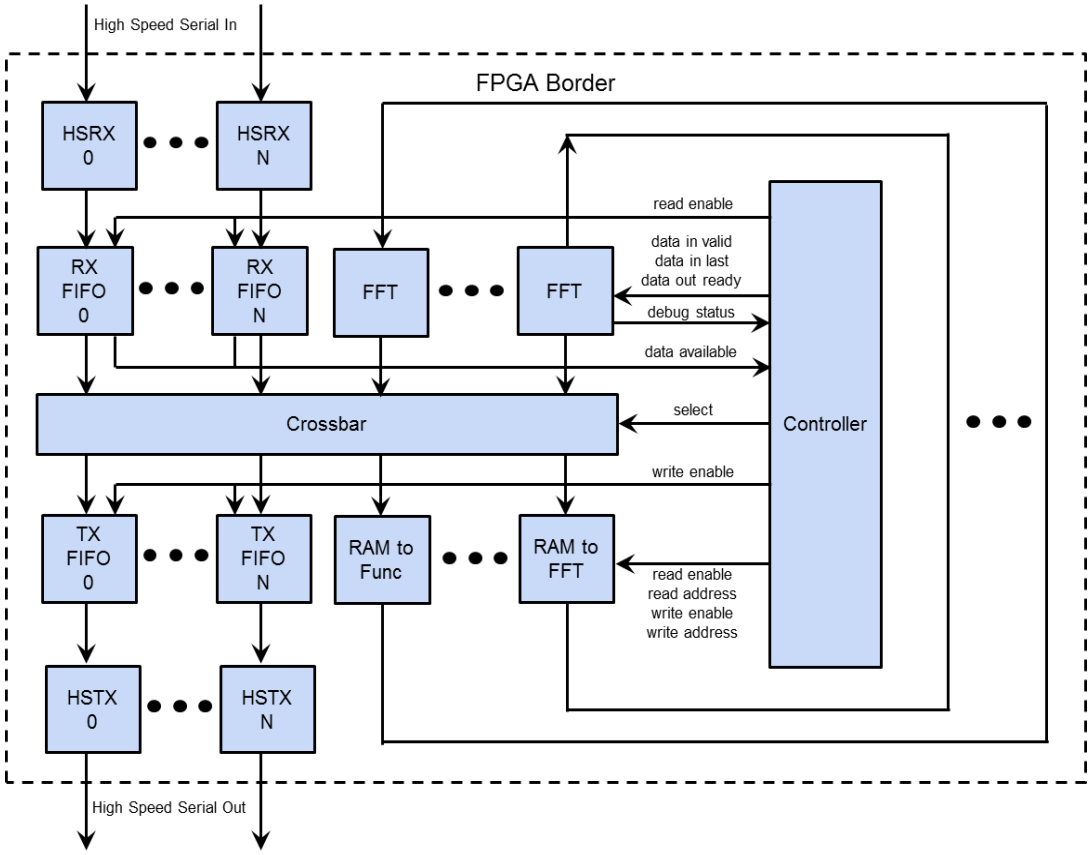

Figure 14: Top level block diagram for the multi FPGA version.

3. High Level Design

Figure 14 shows the block diagram for a single node of the multi FPGA architecture. It contains several of the same parts as the single FPGA design, such as RAMs, FFT Pipelines, and a Crossbar. The new parts to the design are the High Speed SERDES blocks and the FIFOs. By definition this design is intended to cross chip boundaries, and the High Speed SERDES blocks are the mechanism for passing data from FPGA to FPGA with a reasonable bandwidth. Each SERDES is comprised of independent High Speed Transmit (HSTX) and Receive (HSRX) blocks, as shown in Figure 14. The inputs to a High Speed Transmitter is a full word of data, and the output

is a pair of serial lines that are connected to the FPGA output ports which drive physical high speed lines outside the FPGA chip. The High Speed Receiver is the inverse, which has inputs coming from the physical serial lines and the outputs being a full word of data. The FIFOs after the HSRX blocks and the FIFOs before the HSTX blocks serve two purposes. First they give a reliable way for data to cross clock domains, and secondly they help stabilize the ebb and flow of the High Speed links.

4. Registered Crossbar

Initially the Crossbar from the single FPGA version was thought to be good enough for this design. However it was discovered that the Crossbar in the single FPGA version was found to be the primary reason for low maximum clock frequencies after place and route. The first and most trivial approach to fix the Crossbar fanout was to add delays to the data path as seen in Figure 15. This version of the Crossbar will register all of the input data twice and the output data once for each mux. This gives the FPGA place and route extra clock cycles to physically move the data around to the different muxes. Registering the data twice before driving it into the muxes seemed to be the optimal number. Two registers gave significantly better performance than one, while three registers provided very little extra performance.

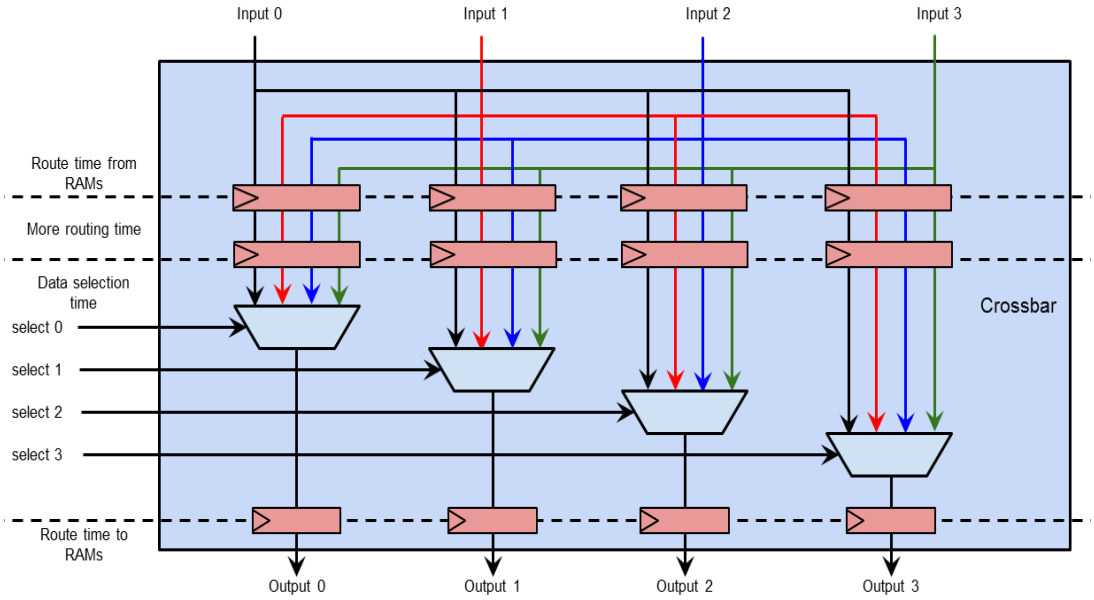

Figure 15: An example 4 input/output registered Crossbar.

5. The Racetrack

Another routing mechanism that was considered in place of the Crossbar is a Racetrack. The Racetrack, shown in Figure 16, is a circular routing ring that data will travel around until it exits at its destination. This type of mechanism requires that all data entering the Racetrack must have a routing ticket, or tag, to let the data know where to exit the ring. In keeping the offline routing approach, each input port to the Racetrack would have a queue of predetermined tickets that it would give to each data that passes through the port. The tickets would then be stripped off when the data exits the Racetrack.

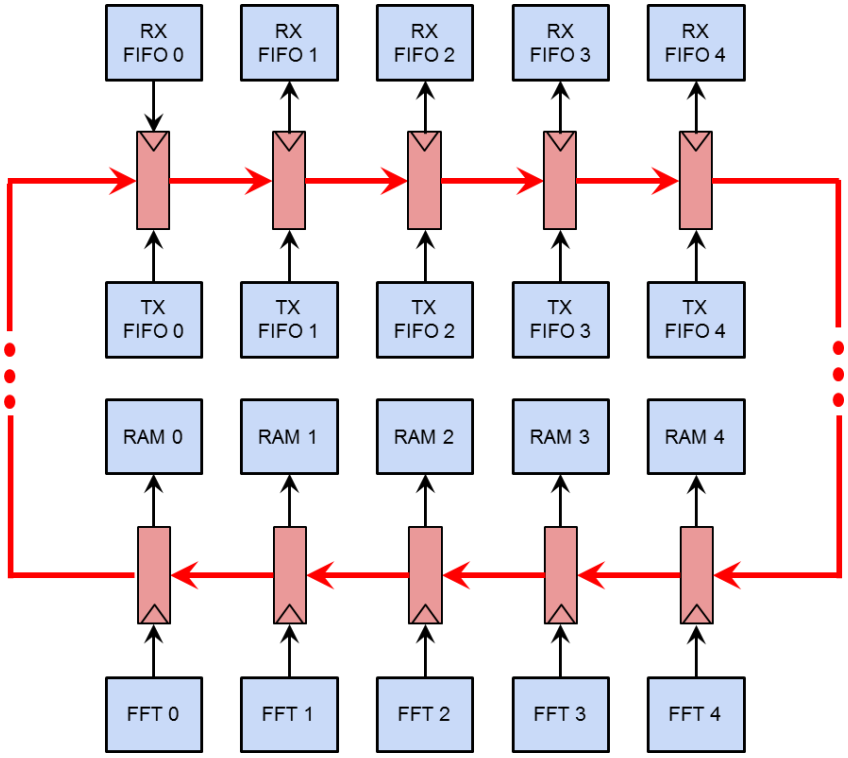

Figure 16: A functional block diagram of how a Racetrack would fit into the multi FPGA design. The actual implementation would also have logic for data tagging/untagging and muxing the racetrack register.

There are a couple of significant problems with using a Racetrack in this design. The first is that for the offline routing system to work, the order that data exits the Racetrack must be maintained. If strict order is not maintained, then downstream mechanisms would use the wrong data in the computations. The problem with the Racetrack is that there are race conditions where new data bound for the same output port as older data can exit the Racetrack before the older data. These race conditions can be solved by waiting for all of the data in one entrance cycle to fully exit the Racetrack before putting the next set of data on. This leads to the second problem of the Racetrack, which is that a strict ordering Racetrack cannot be fully pipelined. This is a problem for

the design because extra latency for each entrance cycle of data will slow the entire system to a crawl because every route on the Racetrack would take multiple clock cycles.

A solution for ensuring strict order while allowing data to be pipelined would be to use a priority queue on the output port. Since the data queuing on the output is small, a shift register priority queue would work well for this application because it sorts the data immediately as the input data enters the queue [14]. The problem with using a priority queue in this design is that when data is released into the Racetrack there is nothing to stall the output data before it gets written into the RAMs. This presents a problem for the priority queue because the queue will require several clock cycles sort out any possible race conditions in the Racetrack.

Another issue with the Racetrack dataflow that would be problematic with this design is each entry into the Racetrack can be arbitrarily stalled by the data on the track itself. This is not a problem with the HSIF FIFOs but is a problem with the FFT Pipelines that also feed the Racetrack. The FFT Pipelines can be stalled from the Control ROM but arbitrarily stalling one FFT Pipeline while letting the others continue would cause different pipelines and RAMs to be on different steps of execution. With all these problems a Racetrack does not appear to be a good solution for the routing in this design.

6. Control with High Speed Interfaces

The Controller for the multi-node design is significantly different from the one in the single FPGA design. This is due to the difficulties of working with High Speed SERDES blocks. The problem is that the SERDES are always on their own clock domain from the rest of the system because their external clocks have to run at gigahertz

speeds. SERDES blocks get their high speed clocks from internal PLLs on the FPGAs themselves and typically use a clock that is different from the system clock because the frequency needs to be easily modified to get the correct PLL speeds for the high speed link [29]. This clock domain crossing invalidates the assumption from the single FPGA version that movement of data can simply be pre-determined at every clock cycle.

It is still possible to get the same pre-determined data flow effect; it just requires the ability to stall all of the pipelines. Essentially the Controller now has to wait for data to be ready in all of the inbound HSRX FIFOs before it can proceed to the next step of operation. If the Controller sees that data is not available at a FIFO it has the ability to stall all of the FFT Pipelines, RAMs, and FIFOs while it waits for data. The Controller also has the option of not waiting for data to be available in any particular RX FIFO by letting the RX FIFO enable bit in the Control ROM to be zero. The second half of letting a NOP bubble into the system is that the TX FIFOs and RAMs must have the knowledge to know that it is a NOP and not latch the data. This action is performed using the TX FIFO enable and RAM Write enable bits in the Control ROM.

Table 6 shows the details of the data stored in the Control ROM. It works in the same fashion as the single FPGA version, in that every cycle a control word is read out and that is used to drive all of the RAM controls, Crossbar controls, FFT Pipeline controls, and now what input FIFO to require data from. This simple stalling system allows for the entire FPGA to stay in sync to the current control cycle while handling the potential ebb and flow from the HSRX blocks.

Table 6: The data that comprises a single entry in the Control ROM for different size FFTs and number of RAMs in the multi FPGA design. The implemented versions are highlighted.

# Nodes	FFT Size	# SERDES	# RAMs/ FFTs	RAM Wr En	RAM Wr Adr	RAM Rd En	RAM Rd Adr	Cross bar	RX FIFO En	TX FIFO En	FFT Data In Valid	FFT Data In Last	FFT Data Out Halt	Done	TOTAL Control Width
64	16	6	2	2	10	2	10	24	6	6	2	2	2	1	**67**
64	16	6	10	10	30	10	30	64	6	6	10	10	10	1	**187**
64	16	6	26	26	52	26	52	160	6	6	26	26	26	1	**407**
64	16	6	58	58	58	58	58	384	6	6	58	58	58	1	**803**
64	32	6	2	2	16	2	16	24	6	6	2	2	2	1	**79**
64	32	6	10	10	60	10	60	64	6	6	10	10	10	1	**247**
64	32	6	26	26	130	26	130	160	6	6	26	26	26	1	**563**
64	32	6	58	58	232	58	232	384	6	6	58	58	58	1	**1151**
64	64	6	2	2	22	2	22	24	6	6	2	2	2	1	**91**
64	64	6	10	10	90	10	90	64	6	6	10	10	10	1	**307**
64	64	6	26	26	208	26	208	160	6	6	26	26	26	1	**719**
64	64	6	58	58	406	58	406	384	6	6	58	58	58	1	**1499**
64	128	6	2	2	28	2	28	24	6	6	2	2	2	1	**103**
64	128	6	10	10	120	10	120	64	6	6	10	10	10	1	**367**
64	128	6	26	26	286	26	286	160	6	6	26	26	26	1	**875**
64	128	6	58	58	580	58	580	384	6	6	58	58	58	1	**1847**

7. The High Speed Interfaces

In the same manner that the single FPGA design used the Xilinx LogiCORE IP core generator to generate the FFT Pipeline IP, the multi FPGA version uses the Xilinx core generator to generate IP for the FFT Pipeline and for the High Speed SERDES. The LogiCORE IP Aurora 64B/66B v6.1 is the core that was selected for the SERDES blocks in this design. The reason is because the Aurora offers options for a stripped down, low overhead communication protocol that is available in a streaming, frameless mode. It also provides a natural data width of 64 bits, which is the same width as the largest FFT Pipeline data width (32 bits for real and 32 bits for imaginary).

For this design the core parameters for the Aurora IP are a single lane with unidirectional communication, no flow control, no user control blocks, and a streaming interface [23]. This setup provides the architecture with the lowest overhead and therefore the lowest latency for data transmission. Increasing the number of lanes for each transceiver would increase the throughput of the entire system, but for this design that is limited to the number of physical pins on the FPGA and board. The unidirectional and streaming interface parameters allows for constant streams of data without any headers or footers being transmitted, which is ideal for this design. The Aurora 64B/66B in streaming mode uses an AXI Stream user interface, similar to the data interfaces used by the FFT Pipeline.

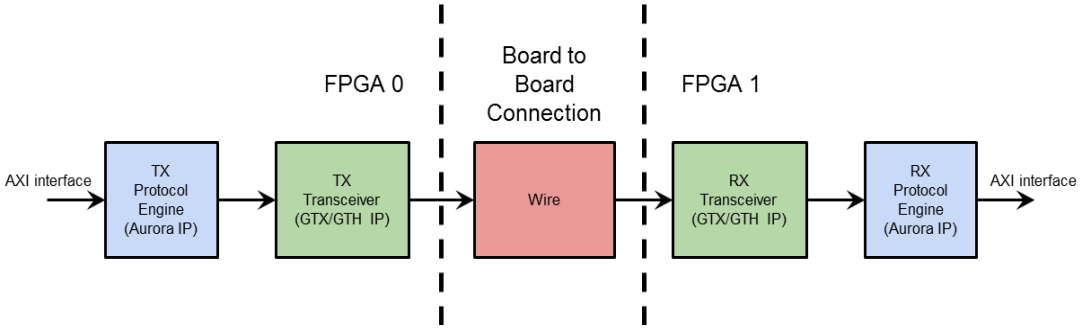

Figure 17: The latency path for a single high speed connection. The Aurora IP is the SERDES block, and the GTX/GTH IP is the transceiver block.

The throughput for an Aurora connection ranges from 600 Mbps to 194 Gbps depending on the type of transceiver (Xilinx GTX or GTH) and the speed of the clock that the transceiver is tied to [23, 29]. However, this design focuses on latency over throughput, so trying to understand what the latencies are is important. Figure 17 shows the latency path for a single high speed connection. The Protocol Engines comprise all of the coding, serialization, and link layer tasks found in the Aurora modules. The GTX/GTH transceivers comprise all the high speed data transmission and physical layer tasks such as syncing. The wire comprises all of the time delay due to the length and physical composition of the physical connection between the transmit and receive transceivers. The latency of the Protocol Engines tend to be straight forward due to their mechanical nature, however the transceiver, physical wire, and physical connection protocols tend to be difficult to accurately predict the latencies of a single transfer. Xilinx claims that the maximum latencies for designs using GTX and GTH transceivers is approximately 37 and 45 user clock cycles respectively [25]. This inability to exactly predict the transfer latency is why elasticity was added to the design in the form of the RX/TX FIFOs and the pipeline stall mechanism.

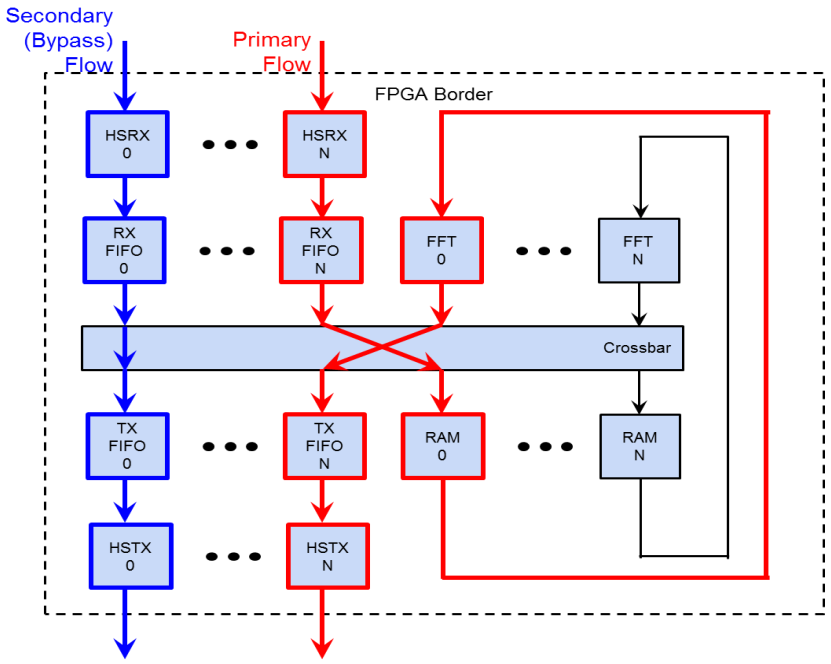

Figure 18: Primary and secondary data flows through the multi FPGA design.

8. Dataflow through the Design

The data flow and controls of the multi FPGA node is very similar to that of the single FPGA version, except that there is now a bypass data flow as well. The primary flow for a single node is shown is Figure 18. Data will enter through the HSRX blocks, get routed to a RAM, run through an FFT Pipeline, and then get routed either to an HSTX block or to a RAM for the next round of calculation. The secondary (bypass) flow, also shown in Figure 18, is when data enters from an HSRX block and is routed directly to a HSTX block. The bypass flow is due to the physical 3D torus topology. If a node needs to send data to a node that it is not directly connected to, then the data must flow through one or more FPGAs using the bypass flow.

The number of High Speed SERDES and this extra bypass flow lever some restrictions on the number of usable FFT Pipelines in the design. This is due to the contention for outbound data resources. The more pipelines there are that require data to leave immediately, the less room there is for bypass data to use those outbound slots. For the 3D FFT case, the bypass data tends to be very bad because the transpose of a single block of FFT requires it to be distributed evenly across all of the nodes in that dimension. This means that of all the data in one FFT block, the same number of data points will be sent 1 hop, 2 hops, etc. up to the maximum number of hops. The other part that makes this transmission worse is that when one node begins transmitting, all of the nodes will begin transmitting, which will require a significant amount of the bypass bandwidth in each node.

Table 7: 3D FFT data point distribution over multiple nodes and multiple RAMs in each node. The highlighted versions were implemented.

# Nodes	FFT Size	# Data Points Total	# Data Points per Plane	# Data Points Total per Node	# Planes per Node	# FFTs per Node	# RAMs	# Data Points per RAM	# Data Points per RAM (Rnd Up)	RAM Address Width
64	16	4096	256	64	0.25	4	2	32	32	5
64	16	4096	256	64	0.25	4	10	6.4	8	3
64	16	4096	256	64	0.25	4	26	2.461	4	2
64	16	4096	256	64	0.25	4	58	1.103	2	1
64	32	32768	1024	512	0.5	16	2	256	256	8
64	32	32768	1024	512	0.5	16	10	51.2	64	6
64	32	32768	1024	512	0.5	16	26	19.69	32	5
64	32	32768	1024	512	0.5	16	58	8.82	16	4
64	64	262144	4096	4096	1	64	2	2048	2048	11
64	64	262144	4096	4096	1	64	10	409.6	512	9
64	64	262144	4096	4096	1	64	26	157.53	256	8
64	64	262144	4096	4096	1	64	58	70.62	128	7
64	128	2097152	16384	32768	2	256	2	16384	16384	14
64	128	2097152	16384	32768	2	256	10	3276.8	4096	12
64	128	2097152	16384	32768	2	256	26	1260.30	2048	11
64	128	2097152	16384	32768	2	256	58	564.96	1024	10

9. Mapping the Data to the RAMs

Even though the exact routing pattern is not defined here, the mapping of the 3D FFT data on a 3D torus of nodes yields some interesting observations. Table 7 shows how data is distributed among the nodes and the RAMs in the nodes for different FFT sizes. The first observation from this table is that as the number of nodes and RAMs per node increases and the FFT size decreases, the data to fill the FFT Pipelines will become sparser until the pipelines are no longer continuously full. This is against the goal of the

design which is to completely hide the communication latency behind computation latency.

The trivial solution for this is to ensure a larger FFT size with fewer RAMs/FFT Pipelines per node. A more useful solution would be to decouple the number of RAMs from the number of FFT Pipelines. The orthogonality of the design would allow for these RAMs to become short term or long term storage that could be used to relieve possible difficult routing conditions. Taking this non-orthogonal design a step further would be to replace the RAM/FFT Pipeline with a data entry/exit point. These entry/exit points could provide access to other FPGAs, external RAMs, or debug interfaces. The RAM/FFT Pipelines could also be replaced with anything, including another function unit other than the FFT, without ever affecting the rest of the system other than the Control ROM data. A modified non-orthogonal block diagram of the design is shown in Figure 19.

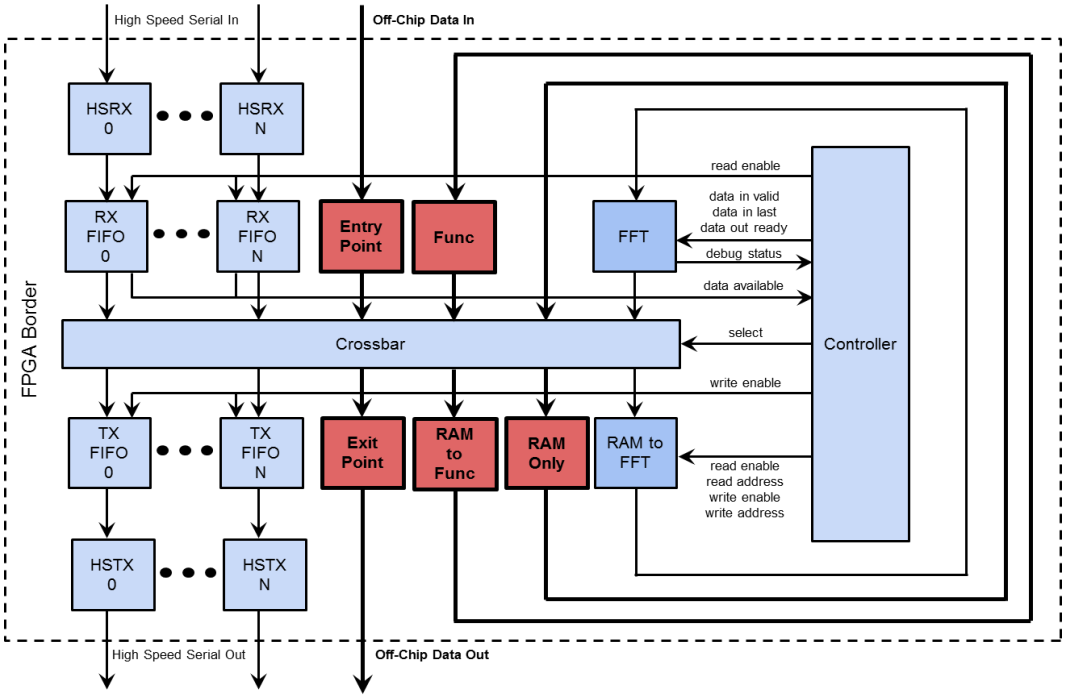

Figure 19: A example of how the original orthogonal multi FPGA design can be modified to add different components.

A final observation is the selection of an odd number of RAMs per node which propagates to an odd number of data points per RAM. The reason for the odd number of RAMs per node is due to the fact that in this design there are 6 High Speed FIFOs in and out of the Crossbar. To make the Crossbar selections efficient it was best to select a number of RAMs to make the total number of inputs/outputs to/from the Crossbar a power of two. This non-power-of-two division then creates an awkward number of data points per RAM, which is rounded up for the actual RAM widths.

10. Results

There are two types of results that are extracted from the multi FPGA design. The first, shown in Table 8, gives the theoretical cycle counts and latencies for various

versions of the design. These are theoretical speeds because the cycle counts of the calculations depend on how many steps are executed in the Controller ROM, and since the actual routing algorithm is not specifically considered in this design it is impossible to give actual simulated cycle counts as shown in the single FPGA version. The cycle times are calculated by figuring out what the minimum number of cycles to complete all of the FFTs over multiple nodes assuming the data is always available. These numbers show the limitations on how many Controller ROM control commands can be used to attain particular levels of efficiency in the design. Actual latency values for various clock frequencies are also shown for the theoretical clock cycle numbers.

Table 8: Theoretical Cycle Counts and Latencies for the multi FPGA. Cycles were calculated using the formula: $Cycles_{Fill} + Cycles_{Drain} + (\#\ Dimensions * \#\ Points/RAM * Cycles_{Pipeline\ Delay})$

# Nodes	FFT Size	# RAMs/ FFTs	Fixed 24/32 Cycles	Latency @ 50 MHz (in ns)	Latency @ 100 MHz (in ns)	Latency @ 200 MHz (in ns)	Float 32 Cycles	Latency @ 50 MHz (in ns)	Latency @ 100 MHz (in ns)	Latency @ 200 MHz (in ns)
64	32	2	1005	20100	10050	5025	1058	21160	10580	5290
64	32	10	429	8580	4290	2145	482	9640	4820	2410
64	32	12	498	9960	4980	2490	551	11020	5510	2755
64	32	18	408	8160	4080	2040	461	9220	4610	2305
64	64	2	6547	130940	65470	32735	6612	132240	66120	33060
64	64	10	1939	38780	19390	9695	2004	40080	20040	10020
64	64	12	1483	29660	14830	7415	1548	30960	15480	7740
64	64	18	1093	21860	10930	5465	1158	23160	11580	5790

The second set of results from the multi FPGA design is the utilization information. Table 9 shows the synthesis and place and route utilization and timing data for a single node instantiation of the multi FPGA design. The overall utilization for the multi FPGA design is significantly smaller than the single FPGA version, which is to be

expected because there are fewer FFT Pipelines and RAMs in the multi FPGA versions. The biggest result is the major positive affect the registered Crossbars had on the place and route timing numbers. For example, a single FPGA design with 16 input non-registered Crossbars had a maximum clock period of 10.857; whereas a multi FPGA design of the same input width with registered Crossbars has a maximum clock period of 8.259. This is a 24% reduction of the clock period, and more importantly it makes the entire design more realistic by pushing it over the 100 MHz barrier, which is the starting point of reasonable FPGA clock speed. Even though strides were made in the multi FPGA design over the single FPGA design, the table of utilization and timing still clearly shows that this design is not scalable. As the number of input ports to the Crossbar increases, so does the latency of the design. The place and route issues resulting from the Crossbar are severely limiting even simple versions of this design.

Table 9: Synthesis and Place and Route results from the multi FPGA version.

Design Version					Slice Logic Utilization						Slice Logic Distribution				Block RAMs		Timing	
Design Stage	Data Type	Data Width	FFT Size	# RAMs/ FFTs	% as Register	# as Registers	% as LUTs	# as LUTs	# as LUTs used as Logic	# as LUTs used as Memory	# w/ unused FF	# w/ unused LUT	# w/ unused FF and LUT	# w/ only routing	%	#	Clock Min Period (ns)	Clock Max Frequency (MHz)
Syn	Float	32	32	2	<1	11243	<1	10295	7435	2860	2430	3378	7865	N/A	2	33	3.106	321.958
Syn	Float	32	32	10	1	47643	4	50288	35988	14300	16003	13358	34285	N/A	4	58	3.106	321.958
Syn	Fixed	32	32	2	<1	9071	<1	7919	5713	2206	1482	2634	6437	N/A	2	33	3.008	332.447
Syn	Fixed	32	32	10	1	36783	3	38408	27378	11030	11263	9638	27145	N/A	4	58	3.068	325.945
Syn	Float	32	64	2	<1	12506	<1	11656	8250	3406	2568	3418	9088	N/A	6	78	3.13	319.489
Syn	Float	32	64	10	2	53946	4	53040	36010	17030	12652	13558	40388	N/A	15	198	3.13	319.489
Syn	Fixed	32	64	2	<1	10334	<1	9416	6666	2750	1604	2522	7812	N/A	6	78	3.012	332.005
Syn	Fixed	32	64	10	1	43086	3	41840	28090	13750	7832	9078	34008	N/A	15	198	3.065	326.264
P&R	Float	32	32	2	1	11143	1	8313	5994	1728	1345	1379	6968	591	2	33	6.531	153.115
P&R	Float	32	32	10	1	47143	3	37836	27197	8640	7329	4064	30507	1999	4	58	7.441	134.39
P&R	Fixed	32	32	2	1	8871	1	6404	4506	1344	1098	1307	5306	554	2	33	6.896	145.011
P&R	Fixed	32	32	10	1	37531	3	29148	20446	6720	5984	3852	23231	1874	4	58	7.857	127.275
P&R	Float	32	64	2	1	12356	1	9513	6581	2336	1405	1197	8108	596	6	78	7.249	137.95
P&R	Float	32	64	10	2	53196	3	41912	28117	11696	6749	4198	35163	2099	14	194	8.259	121.08
P&R	Fixed	32	64	2	1	10084	1	7573	5241	1832	1234	1278	6339	500	6	78	7.157	139.723
P&R	Fixed	32	64	10	1	41836	2	32462	21418	9160	5421	3695	27041	1884	14	194	8.154	122.639

V. MULTI FPGA STAGED ARCHITECTURE

1. High Level Design

After analyzing the results from the prior designs, it is obvious to see that the primary limiter for building a scalable high frequency design was to drastically change the design and put the data routing at the center of the design strategy. As such, a more modular and scalar design, shown in Figure 20, was constructed. The primary focus in this staged, cellular design was the unobstructed wide data paths. It aims to keep a clean primary data path flow through all of the computation cells in the design. Some parts of the design were directly leveraged from the first multi FPGA design, such as the HSIFs and the TX/RX FIFOs.

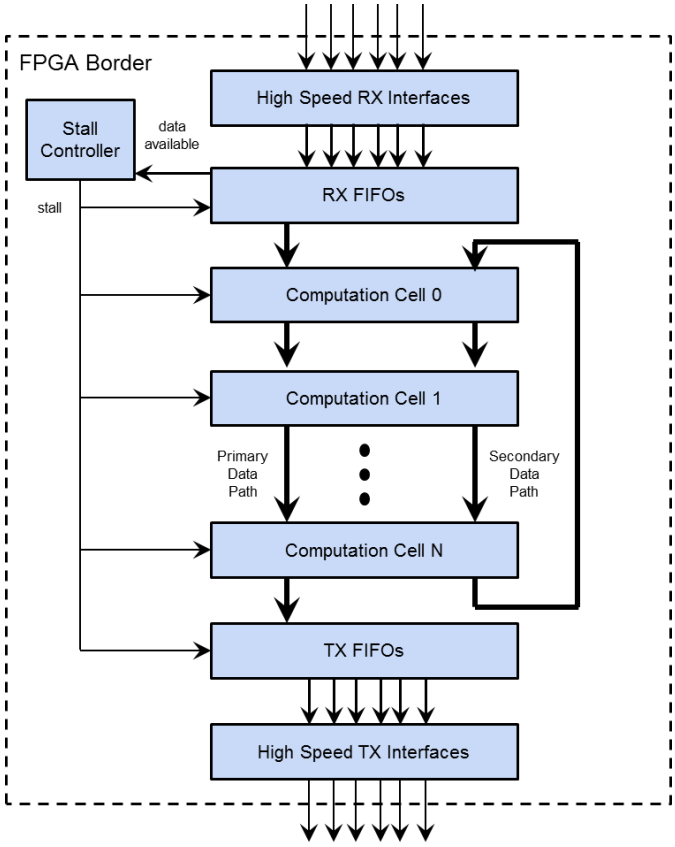

Figure 20: Top level block diagram for the Staged architecture design.

2. Rules and Control

The deviation from the prior designs is the unidirectional data paths that are driven in and out of the computation cells. A computation cell is any function that needs to modify data from the primary data path trunk and do some mathematical function on that data. The computation cells can do any actions as long as they pass along the primary and secondary data streams and abide by the stall signal. Since the primary data flow is always flowing towards the outward edges of the chip, the computation cells must have a mechanism for recycling data backwards in the flow for upstream computation

cells to use. This is where the secondary data path comes in the play. It is essentially a full width data path that is cycled through all of the computation cells.

The final high level part of this high level design is the Stall Controller, which determines when to tell all of the computation cells and FIFOs to stall based on when input data has arrived from the HSRX ports. The Stall Controller contains a ROM, which contains the information of when to wait for data from FIFOs and when to allow NOPs into the pipeline.

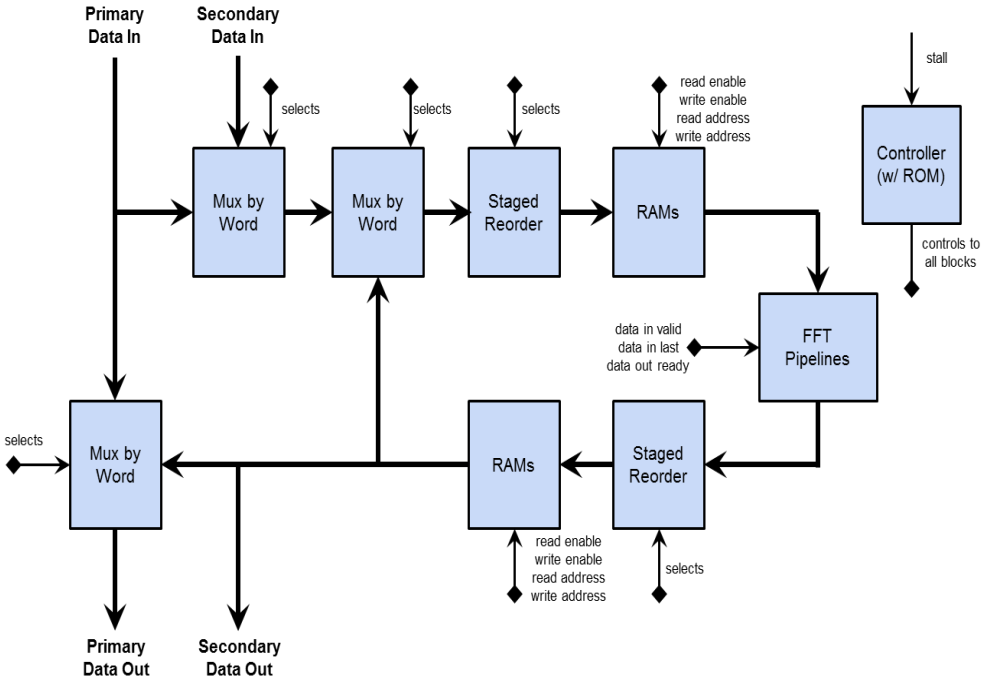

Figure 21: Computation Cell block diagram for the 3D FFT computation cell.

3. FFT Computation Cell Overview

Figure 20 shows the general form of the design which contains N computation cells. For the 3D FFT application these general computation cells are replaced with the FFT Cells shown in Figure 21. Just like in the general cells, the I/O for the FFT Cell is

primary and secondary data in and out as well as the stall control signal. As the prior designs have shown, the routing of the 3D FFT requires many degrees of freedom in routing data in, routing data out, and moving data to/from the RAMs. Instead of trying to pack all of the necessary degrees of freedom into a single location, such as a Crossbar, the data movement freedom is spread over multiple blocks and clock cycles. In keeping with the constraints of the prior multi FPGA design, the cell design is also limited to 6 TX and RX high speed links that will connection to other FPGAs. As such, the primary and secondary data paths have a width of 6 words, which for 64-bit words gives a total of 384-bits per path. This notion of words continues all the way through the FFT Cell. Each block is given inputs of 6 64-bit words that the block deals with. In the case of the RAMs and FFT Pipelines, they also contain 6 of each without any special muxing or switching built in.

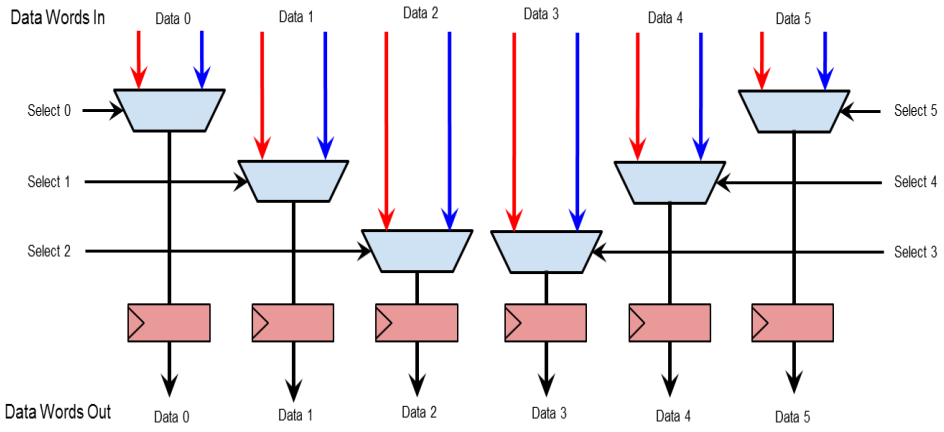

Figure 22: Mux by Word block diagram for the 3D FFT computation cell.

4. The Mux by Word Block

The first major block that is unique to the FFT Cell is the Mux by Word block, which can be seen in Figure 22. All this block does is take two sets of 6 64-bit words and independently selects between them on a word basis to output. This allows the main loop of the FFT Cell to input data from either the primary or secondary streams. It also allows for selective outputting of data in the primary stream. This selective replacement means that the primary stream can easily have bypass data running alongside newly calculated data that is getting shuttled to the next FFT for further calculations. The main reason for such a light selection mechanism for pushing data onto the primary path is to keep that path free of as much logic as possible to ensure easy routing through the FPGA.

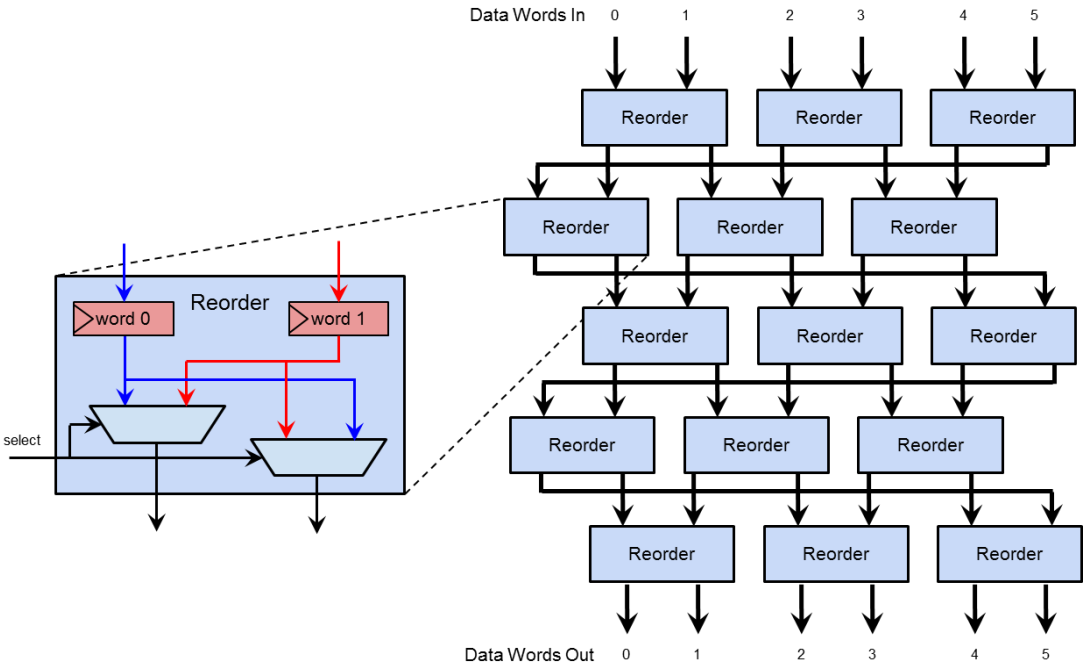

Figure 23: Staged Reorder block diagram for the 3D FFT computation cell.

5. The Staged Reorder Block

The next specific block of the FFT Cell is the Staged Reorder block, which can be seen in Figure 23. The Staged Reorder block is where all of the inter-transmit-and-receive degrees of freedom are created. This block gives the design spatial freedom, meaning any shuffles of data that need to occur in the same time step will take place in this block. The Staged Reorder takes in 6 64-bit words and has the ability to independent swap pairs of them each clock cycle. As show in the example figure, there are 6 words of data which would require a maximum of 3 swaps in an "around the corner" configuration for any particular data point to reach any output location. However, 5 stages are used to ensure that any permutation of input to output data is possible.

6. The Controller Block

All of the other blocks in the Cell FFT design, such as the RAMs, FFT Pipelines, and the Controller, are the same as in prior designs. As in the prior designs, the RAMs provide temporal freedom to reorganize data from different time steps. The one exception to the reuse of blocks is the in the ROM data of the Controller. Since the Cell FFT module has completely different blocks to control, the width, depth, and use of the bits in the ROM are completely different. Table 10 shows the breakdown of the control bits in the Control ROM. Since this cell design allows for multiple FFT Cells, it is important to note that the controls in the table refer to only a single FFT Cell. Each cell could have completely different ROM controls. Given that a simulation model outside of this design is pre-computing all of these ROM tables, splitting it into multiple ROMs should not be a problem.

Table 10: The data that comprises a single entry and total widths of the Control ROM for the Staged design. The implemented versions are highlighted.

# Nodes	FFT Size	# SERDES	# Cells	# RAMs/ FFTs	RAM Wr En (both)	RAM Wr Adr (both)	RAM Rd En (both)	RAM Rd Adr (both)	Word Muxes (all)	Reorder (both)	FFT Data In Valid	FFT Data In Last	FFT Data Out Halt	Done	TOTAL Control Width per Cell	Stall RX FIFO Enables	TOTAL Control Width
64	32	6	1	6	12	96	12	96	18	30	6	6	6	1	**283**	6	**289**
64	32	6	3	12	12	84	12	84	18	30	6	6	6	1	**259**	6	**783**
64	32	6	6	36	12	72	12	72	18	30	6	6	6	1	**235**	6	**1416**
64	64	6	1	6	12	108	12	108	18	30	6	6	6	1	**307**	6	**313**
64	64	6	3	12	12	96	12	96	18	30	6	6	6	1	**283**	6	**855**
64	64	6	6	36	12	84	12	84	18	30	6	6	6	1	**259**	6	**1560**
64	128	6	1	6	12	120	12	120	18	30	6	6	6	1	**331**	6	**337**
64	128	6	3	12	12	108	12	108	18	30	6	6	6	1	**307**	6	**927**
64	128	6	6	36	12	96	12	96	18	30	6	6	6	1	**283**	6	**1704**

7. Results

Table 11 gives the synthesis and place and route results for several versions of the cell design architecture. The most prominent and most promising result is the timing results. Both the synthesis and place and route periods were very low compared to the prior designs, but the more important part is that adding cells in series with each other does not adversely affect the timing. By primarily focusing on the routing during the design, a cell architecture has now allow for much more of the FPGA fabric to be usefully utilized by real functionality instead of being used just to do data routing as in prior designs.

It is clear to see that even though adding more cells is not increasing the maximum clock period they are increasing the utilization of the FPGA itself. This allows for many cells to be added and only reduce the latency of the output data by one clock cycle. Addition of multiple cells, up to the limit of the FPGA fabric, can be implemented without putting any adverse strain on the clock period. However the fabric utilization numbers are still quite small for this FPGA, but design will be limited by the number of block RAMs in the design long before it runs out of slice fabric to use.

The results for clock counts and latencies can be seen in Table 8. The reason for this is that since this design is a multi FPGA architecture, the actual number of cycles is dependent on the routing algorithm and the latencies between the high speed connections. The theoretical estimations of the clock counts from the multi FPGA section should still hold true with this design, simply by changing the number of RAMs/FFT Pipelines.

Table 11: Synthesis and Place and Route results from the Staged architecture version.

Design Version					Slice Logic Utilization						Slice Logic Distribution				Block RAMs		Timing	
Design Stage	Data Type	Data Width	FFT Size	# FFT Cells	% as Register	# as Registers	% as LUTs	# as LUTs	# as LUTs used as Logic	# as LUTs used as Memory	# w/ unused FF	# w/ unused LUT	# w/ unused FF and LUT	# w/ only routing	%	#	Clock Min Period (ns)	Clock Max Frequency (MHz)
Syn	Fixed	32	64	1	1	28868	2	27217	18967	8250	2240	3891	24977	N/A	4	56	2.818	354.862
Syn	Fixed	32	64	3	3	84931	6	80499	55749	24750	6418	10850	74081	N/A	9	120	2.818	354.862
Syn	Fixed	32	64	6	6	169057	13	160478	110978	49500	12837	21416	147641	N/A	16	216	2.818	354.862
P&R	Fixed	32	64	1	2	29275	3	34443	24291	12012	4199	12567	28688	2297	4	56	7.472	133.833
P&R	Fixed	32	64	3	4	87037	7	100090	70907	37152	8342	21411	91295	3592	9	120	7.472	133.833
P&R	Fixed	32	64	6	7	174108	14	206839	145549	74304	18546	52616	185680	4210	16	216	7.472	133.833

VI. CONCLUSIONS AND SUMMARY

1. Comparison of Results

Table 12: Results for 3D FFT calculations of different sizes on the current designs and other HPC systems.

System	32x32x32		64x64x64	
	micro sec	# nodes	micro sec	# nodes
FFTW on 3.0 GHz Core 2 Duo [6]	230	1	1300	1
SPIRAL on two 3.0 GHz Core 2 Extreme chips [16]	107	1	2600	1
Bandwidth-oriented on 8800GTX GPU [1]	N/A	N/A	230	1
Single FPGA design (w/ 16 & 32 FFTs respectively) @ 50MHz	128	1	500	1
Single FPGA design (w/ 16 & 32 FFTs respectively) @ 100MHz	64	1	250	1
SPIRAL on 2.4 GHz Opteron/Infiniband cluster [16]	1700	8	15000	8
Desmond on 2.66 GHz Xeon/Infiniband cluster [20]	43	64	141	64
QCDOC [3]	700	256	1050	1024
Blue Gene/L [13]	100	512	200	4096
Anton [20]	4	512	13	512
Multi FPGA design w/ 10 FFTs @ 100 MHz	4.3	64	19.4	64
Cell design w/ 3 cells (18 FFTs) @ 100 MHz	4.1	64	10.9	64
Cell design w/ 3 cells (18 FFTs) @ 200 MHz	2.4	64	5.5	64

Table 12 shows the final 3D FFT results for all of the designs done here as well as some comparison points from other single and multiple node HPC systems. The single node category is dominated by the 8800GTX Nvidia GPU at 230ms for 64^3 data points. The single FPGA design run at 100MHz is almost compatible with the 8800GTX. This is a very interesting comparison which shows the FPGAs abilities to perform well in a highly parallel, strongly scaled problem space.

In the multinode category, the Anton processor cluster is the fastest by far of all the current systems at 13ms for 64^3 data points. The architecture for a multi FPGA

cluster described here shows comparable performance to a costly custom Anton processor. Given that this design was not actually implemented and validated in hardware these results are not firm. However it does show that COTS FPGA clusters have the potential to give excellent performance for HPC applications that generally require custom calculations.

2. Future Work

The next work to be done on these designs would be to optimize them for timing by adding more registers and working with the Xilinx tools. It is possible for the single FPGA design and Staged Architecture designs to attain a frequency of around 200 MHz based on implementations of similar designs. This would require more iterations of the designs and a bit of re-architecting to attain such performance, but it is very possible. Running at a faster clock frequency would also push the designs to being faster than the single node GPU and multinode Anton clusters for a 3D FFT. These designs would also need to be validated by running them on actual hardware to test for real-life speed and data accuracy before any solid speed claims can be made.

Another piece of work to be done would be to get rid of the Xilinx FFT Pipeline and replace it with a custom, fully parallel FFT Pipeline. A very wide FFT Pipeline would benefit a single FPGA design significantly by allowing more parallelization to take place and reducing the latency of each dimension of calculations. Given the current theoretical single FPGA speed and usage results, it seems possible for a single FPGA implementation of a 3D FFT to be parallelized more and with greater speed.

REFERENCES

[1] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. "Bandwidth-intensive 3-D FFT kernel for GPUs using CUDA," In Proceedings of the ACM/IEEE Conference on Supercomputing, 2008.

[2] F. Annexstein and M.A. Baumslag. Unified approach to off-line permutation routing on parallel networks the interaction of architecture and operating system design. In Proc. of the 2nd ACM Symposium on Parallel Algorithms and Architectures (1990), pp. 398-406.

[3] B. Fang, Y. Deng, and G. Martyna. Performance of the 3D FFT on the 6D network torus QCDOC parallel supercomputer. Computer. Physics Communication 176 (2007), 531–538.

[4] D. Bertsekas, C. Ozveren, G.D. Stamoulis, and P. Tseng. Optimal communication algorithms for hypercubes. Journal of Parallel and Distributed Computing 11 (1991), 263-275.

[5] A. Edelman. Optimal matrix transposition and bit reversal on hypercubes: All-to-all personalized communication. Journal of Parallel and Distributed Computing 11, 3 (1991), 328-331.

[6] FFTW benchmark page. http://www.fftw.org/speed/.

[7] M.C. Herbordt and P. Swarztbauber. "Towards Scalable Multicomputer Communication Through Offline Routing," National Science Foundation.

[8] M.C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. Achieving high performance with fpga-based computing. Computer, 40(3):50 –57, march 2007.

[9] Infiniband Trade Association. InfiniBand Architecture Specification, 2002.

[10] S. Johnsson and C.-T. Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. IEEE Transactions on Computers 38 (1989), 1249-1268.

[11] S. Johnsson, and C.-T Ho. Optimal communication channel utilization for matrix transpose and related permutations on binary cubes. Discrete Applied Mathematics 53 (1994), 251-274.

[12] A. Kojima, N. Sakurai, and J.I. Kishigami. "Motion detection using 3D-FFT spectrum," IEEE International Conference on Acoustics, Speed, and Signal Processing. ICASSP-93. 5, 213, 216.

[13] M. Eleftheriou, B. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. Germain. Performance measurements of the 3D FFT on the Blue Gene/L supercomputer. In Proceedings of Euro-Par…Parallel Processing, J. C. Cunha and P. D. Medeiros, Eds. Springer-Verlag, 2005, New York, NY, 795–803.

[14] S.-W. Moon, K.G. Shin, and J. Rexford. "Scalable hardware priority queue architectures for high-speed packet switches," Proceedings/Real-Time Technology and Applications Symposium, 1997, 203, 212.

[15] D. Shoemaker, F. Honore, C. Metcalf, and S. Ward. Numesh: An architecture optimized for scheduled communication. Journal of Supercomputing 10, 3 (1996), 285-302.

[16] SPIRAL results page. http://www.spiral.net/bench.html.

[17] Q. Stout and B. Wagar. Intensive hypercube communication: Prearranged communication in link-bound hypercubes. Journal of Parallel and Distributed Computing 4 (1987), 95-115.

[18] P. Swarztrauber. Transposing arrays on multicomputers using de Bruijn sequences. Journal of Parallel and Distributed Computing 53 (1998), 63-77.

[19] P. Swarztrauber, and S. Hammond. A comparison of optimal FFTs on torus and hypercube multicomputers. Parallel Computing. 27, 6, 847-859, 2001.

[20] C. Young, J.A. Bank, R.O. Dror, J.P. Grossman, J.K. Salmon, and D.E. Shaw. "A 32x32x32, spatially distributed 3D FFT in four microseconds on Anton," SC'09: Proceedings of the 2009 ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis, 1, 11.

[21] R.O. Dror, J.P. Grossman, K.M. Mackenzie, B. Towles, E. Chow, J.K. Salmon, C. Young, J.A. Bank, B. Batson, M.M. Deneroff, J.S. Kuskin, R.H. Larson, M.A. Moraes, and D.E. Shaw. "Exploiting 162-Nanosecond End-to-End Communication Latency on Anton," High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 1,12, 2010.

[22] Xilinx, Inc. 2011. ISIM User Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/plugin_ism.pdf

[23] Xilinx, Inc. 2011. LogiCORE IP Aurora 64B/66B v6.1. http://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b/v6_1/ug775_aurora_64b66b.pdf

[24] Xilinx, Inc. 2012. 7 Series FPGAs Overview. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[25] Xilinx, Inc. 2012. LogiCORE IP Aurora 64B/66B v7.1. http://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b/v7_1/ug775_aurora_64b66b.pdf

[26] Xilinx, Inc. 2012. LogiCORE IP Fast Fourier Transform v8.0. http://www.xilinx.com/support/documentation/ip_documentation/ds808_xfft.pdf

[27] Nallatech. http://www.nallatech.com/, 2013.

[28] GiDEL. http://www.gidel.com/, 2013.

[29] Xilinx, Inc. 2012. 7 Series FPGAs GTX/GTH Transceivers User Guide. http://www.xilinx.com/support/documentation/user_guides/ug476_7Series_Transceivers.pdf

[30] M. Chiu and M.C. Herbordt. “Molecular dynamics simulations on high performance reconfigurable computing systems,” ACM Transactions on Reconfigurable Technology and Systems. 3, 4, Article 23 (November 2010), 37 pages.

VITA

[redacted]