Transmathematica DOI Prefix 10.36285 ISSN 2632-9212

Slipstream Transcomputation Of The Fast Fourier Transform

Stephen F. K. Leach
sfkleach@gmail.com

James A.D.W. Anderson
j.anderson@reading.ac.uk
Department of Computer Science
The University of Reading
England
RG6 6AY

Submitted: 3 July 2019
Revised: 24 December 2019

Abstract

Computation based on a total arithmetic eliminates the need to handle arithmetical exceptions, which is one of the major obstacles to the exploitation of fine-grain, massive, multi-processor architectures. We have been designing and investigating a family of fine-grained architectures that exploit exception-free arithmetic to implement a statically assigned, systolic, dataflow technique that we call "slipstreaming." In this paper, we report on the simulation of a slipstreamed implementation of the Fast Fourier Transform (FFT), which is an important numerical algorithm in engineering. We consider its properties and compilation challenges. The compilation challenges additionally include using the output of the FFT to dynamically compute a match against a fixed target. We find, empirically, that the latency is a good aproximation to linear in the number of sample points transformed by the FFT.

1 Introduction

Almost everything that computers do involves arithmetic and may fail on arithmetical exceptions, such as division by zero. Computers usually have large and complicated exception handling circuitry to allow the programmer to specify how the computer should attempt to recover from an exception. Such recovery usually involves long delays in fetching and executing the exception handling code. Such delays can often be tolerated in serial computers but the position is more complicated in parallel

computers. We need to consider two kinds of parallelism. Course-grain parallelism executes large bocks of code in parallel. Within each block, instructions are executed sequentially. This requires large, von Neumann processors, that can execute arbitrary code. By contrast, fine-grain parallelism executes small blocs in parallel. The blocks may contain just one or a small number of sequential instructions. The smaller the block, the more the parallelism. Fine-grain parallelism can use much smaller and faster, non-von-Neumann processors. As the processors are smaller, a computer chip can have more of them and this opens up the possibility that it can do more computation per unit time. We take this to an extreme by executing just one instruction, in each processor, in a very fine-grain architecture.

Computer scientists have explored many different kinds of parallelism. The most parallel is a systolic array. These are arrays or lattices of processing cores laid out on a computer chip. On every clock tick, all of the cores read data from their immediate neighbours, process the data and get the result ready to be read by their immediate neighbours on the next clock tick. Data flows through the array systolically, that is, in a heart beat, regulated by the computer's clock. Systolic arrays have a number of advantages and disadvantages. Our approach is to go as far as we can to retain advantages and remove disadvantages.

Systolic arrays were proposed as an easy to manufacture way of getting many processors on a chip; but they make some assumptions that are impossible to deliver with current technology. A systolic array is a lattice of processors, arranged such that every processor, on the edge of the chip, has an input or output channel. It is impossible to fabricate a large number of Input and Output (I/O) channels, on a chip, so it is impossible to read and write data at the theoretical speeds required by systolic arrays. The literature on systolic arrays usually neglects the issue of I/O, which often leads to unrealistic expectations of processing speed. In our architectures, we require only one input and output channel per edge of the chip. This small number of channels can be manufactured. A greater number of channels would improve performance, up to and beyond that of a systolic array, but we accept the limitations of current fabrication methods.

Systolic arrays allow arbitrarily complex functions to be placed in each core. This can be achieved by using von Neumann cores to compute the functions but this complicates timing on the chip. A von Neumann core uses a clock to control its execution of a sequential program, executing at most one instruction, from a program, per clock tick. It follows that a von Neumann core takes many clock ticks to compute a complex function. The chip must have a processor clock whose timing signals are passed to all processors but the systolic requirement of passing data on each clock tick means there must be a separate data clock that instructs each processor to read data from its immediate neighbours. The data clock must be slowed down to allow the slowest von Neumann processor to do its work but this means that all von Neumann processors are slowed down to this slowest speed. This gives poor performance and requires additional circuitry to provide two clocks and the means for slowing one of them down.

We take a different approach. We use one or a small number of instructions, in each core, and use many cores to emulate complex functions.

This means that the array can operate at full speed all of the time. However, it requires that data, carried by tokens, passes systolically over the many adjacent cores that are emulating the complex function. This is done using a data pipeline that allows tokens to pass over an arbitrarily complex, acyclic, path using just two bits in a token's header. Further more, we arrange that each processor core has two I/O ports per edge. This supports efficient binary fan-out from a serial thread to many parallel threads and fan-in from many parallel threads to one serial thread. Fan out is typically used to distribute work and fan-in to collect results.

Thus we arrive at a very fine-grain architecture that has an array of many non-von-Neumann cores, arranged with simpler I/O, but more complicated data paths, than systolic arrays. We conservatively estimate that we will be able to manufacture of order ten thousand cores per chip, with 22 nm fabrication technologies, giving, at least, order one million cores per board. We further estimate that the chips will execute at 1 GHz, giving a theoretical maximum, of order, 1 PHz. As the cores execute two FLoating-point OPerations per clock tick, both a multiplication and an addition, this is a theoretical maximum of order 2 PFLOP per board. This is a very high performance and is more cores than early computers had memory locations. Indeed we think of our cores as smart memory locations that perform computations on their data. Just as today's computer users tolerate low utilisation of memory, so we tolerate similarly low utilisation of cores.

Computer scientists have explored many ways of compiling programs for arrays of processors; many of these compilation strategies are very complex. The second named author, of the present paper, specified the "slipstream" paradigm of language programming. In this paradigm, programs are laid out in a two dimensional pipeline, respecting our limitation of zero dimensional I/O, where systolic arrays specify one dimensional I/O. He implemented the first slipstream programs by hand. These supplied mathematical libraries that can be combined into mathematical programs that inherit the property of executing in a pipeline. Such pipelines read successive data, on each clock tick, and can execute many mathematical functions or, indeed, many mathematical programs, per clock tick. He implemented a molecular dynamics program with the property that many hundreds of instantiations of the program can be executed in a simulation of one of our boards. That is, it completes execution of many hundreds of molecular dynamics programs per clock tick, thereby allowing many hundreds of pairwise molecular interactions to be computed per clock tick.

The first-named author, of the present paper, generalised this approach and developed compiler techniques to support slipstream programming. He applied these techniques to the Fast Fourier Transform (FFT), which is an important numerical algorithm in engineering. The Fourier transform is unitary, in the complex domain, so it can be implemented, as a matrix multiplication, in time order $O(n^3)$, where n is the number of complex sample points that are transformed. This is too slow to be of practical application in engineering. By contrast, the FFT operates in time order $O(nlog_2n)$. This is so fast that the FFT is, today, the basis of a very large part of signal processing, in all manner of devices from mobile telephones to radio telescopes. Empirical measurements on our slipstream implemen-

tation of the FFT show that it operates with a latency of order $O(7n)$. This means that the first signal in a slipstream takes order $O(7n)$ clock ticks to transform and is then followed by successive transformed signals every clock tick. In further work, not reported here, we examined trade offs in the number of clock ticks in which a signal is input and the latency of processing. We found that the slower a signal is input, the shorter the latency and the smaller the number of cores used to compute the FFT.

A further advantage of the slipstream implementation of the FFT is that it can be followed by further processing, also in a slipstream. Here we investigate adding a matched filter to the FFT stream. Similar computations can be used to sharpen signals and to detect and track multiple targets, for example ground vehicles or aircraft in a radar signal.

In the next sections, we give further information about our architecture, the compilation strategies used and the performance of the slipstream FFT. We then discuss wider implications of our architecture and slipstream programming paradigm and conclude with a statement of the original contributions of the paper.

2 Architecture

Our target architecture for this case-study is a single chip that contains a dense, rectangular array of cores, each capable of a fused multiply-add, connected orthogonally by short pipelined queues. This is illustrated in Figure 1.

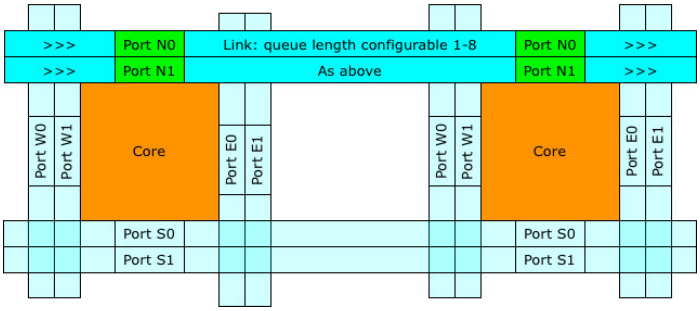

Figure 1: Target Architecture

The cores are not programmable in the conventional sense of being able to load and step through a sequence of instructions but are merely configurable: on a clock tick they accept fixed-size input tokens (or packets) on their input ports and generate fixed-size output tokens that are loaded onto their output ports. Depending on the configuration in the core and the routing bits set within the token, tokens may be routed onto an adjacent core without further processing and may additionally be routed into the arithmetic unit. Every core maintains a very small set of

registers that can be treated as additional inputs and can be targeted as additional outputs.

It should be noted that this design is a simplified version of our production-level design. This enabled the design team to decouple the current investigation from the ongoing production design. However, our simplified core has the all the features required for the problem at hand and presents near-identical challenges for compilation. Figure 2 shows a block-diagram for the simplified core.

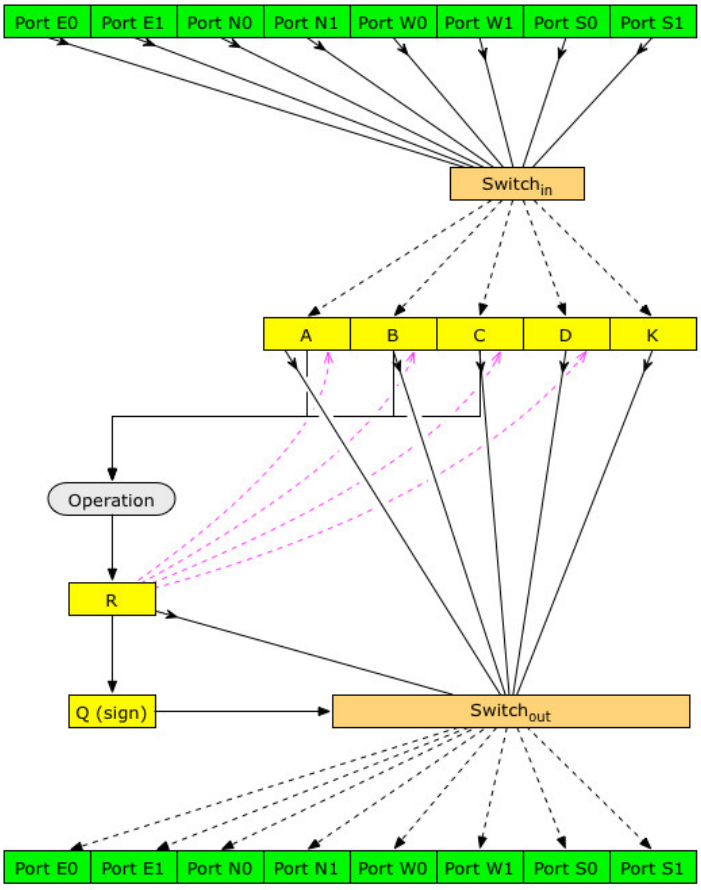

Figure 2: Simplified Core

There is one area in which the simplified core significantly differs from the production design, which is that it lacks support for the initial configuration phase. Configuration is effected by flooding the chip with a compiled stream of tokens that are tagged as configuration tokens. The configuration bit has the effect of diverting the token from the Arithmetical Logic Unit (ALU) into the configuration registers of the core. The task of converting a target configuration into a stream has been previously investigated; the straightforward implementation had a stream whose size and generation time were linear in the number of configuration registers to be set.

It should be apparent that, relative to a standard Central Processing Unit (CPU), these cores are radically stripped-down. The benefit is that they can be packed far more densely than would normally be possible. Our conservative estimate is that a single chip will support more than 10,000 cores, using standard 22 nm fabrication technology. The disadvantage is that the cores are so simplified that they cannot operate independently, which limits them to data-flow problems.

The central idea behind slip-streaming is to allow 'waves' of computation to sweep through the machine, each one following immediately behind the other. One can think of each wave as a state vector, propagating from one set of processors to the next, although the size and meaning of the vector typically changes at each step. It is vital to appreciate that slip-streaming is systolic, in so far as all processors activate on every clock tick, even if all inputs are blank tokens representing no-operation (NOOP). Processors do not store their inputs because they may be obliterated on the next clock tick by the next wave of state data.

In this style of computation, values that need to be processed together, by a multiply-add (or other instruction), must arrive at a processor on the same clock tick. It is the responsibility of the compiler to adjust travel time and path geometry so that data arrives simultaneously.

3 Problem

We implemented a slipstreamed FFT, followed by a complex, match-filter and calculate the energy of the match. We describe the FFT separately as it is entirely modular and is the most complex part of the design.

3.1 FFT

The Fast Fourier Transform, given below as F, is a discrete version of the Fourier transform and is completely described as the product of a square matrix, M, and a vector, Z, of size, n, in the complex domain:

$$F = \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \dots \\ 1 & \omega^{-1} & \omega^{-2} & \dots \\ 1 & \omega^{-2} & \omega^{-3} & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \dots \end{bmatrix} \tag{1}$$

Where ω is the n'th root of unity: $\omega = \exp 2\pi i/n$.

3.2 Match-Filter

The match-filter is defined as follows. Let F be the complex vector that is the FFT of the input vector, as shown, above, in Equation 1. Let T be the complex vector that is the target vector. Then the match vector, V, is obtained by multiplying corresponding complex elements (x, iy) of the vectors F and T to give the Hadamard product $V_k = F_k T_k$. The energy of the match is $\sum_k |x_k i y_k|^2$, where $V_k = (x_k, iy_k)$.

In our implementation we were interested in the challenges presented by all three parts of this computation. However, if only the energy of the match is needed and not the values of the match vector then we need not permute F in the output stage of the FFT. This saves a very significant amount, one third, of the latency of the FFT.

3.3 General Approach to Compilation

At the time of writing, we have implemented a variety of compiler components that we typically orchestrate using custom code for particular challenges, reflecting our historical focus on providing optimised low-level mathematical functions rather than the provision of a general purpose compiler. This has been a fruitful approach that has given us the opportunity to understand the nature of optimising for this novel architecture and the variants which we have studied. However, all the components fit within an overall general approach, which we describe here using the example of compiling an arithmetical expression.

The compilation strategy for arithmetical expressions is to transform them into a net representation, which is done by transforming into static, single-assignment form and then into a variant of Three-Address Code (TAC). For example, the single assignment

$$r = ax^2 + bx + c$$

transforms into TAC as

```
t0 = a * x
t1 = t0 * x
t2 = b * x
t3 = t1 + t2
r  = t3 + c
```

which is equivalent to the net shown in Figure 3.

In order to enforce simultaneous arrival, which is the systolic constraint, the compiler segments the net by required arrival time. In Figure 4, the segments are shown to the left of the red lines.

The compiler inserts explicit no-ops, representing pass-through, which specifies the appropriate state-vector for each time-step. See Figure 5.

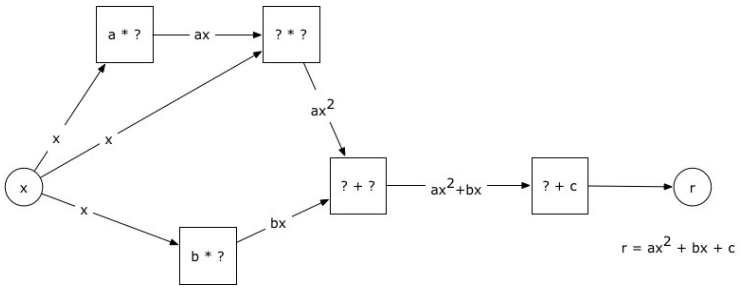

Figure 3: TAC Net

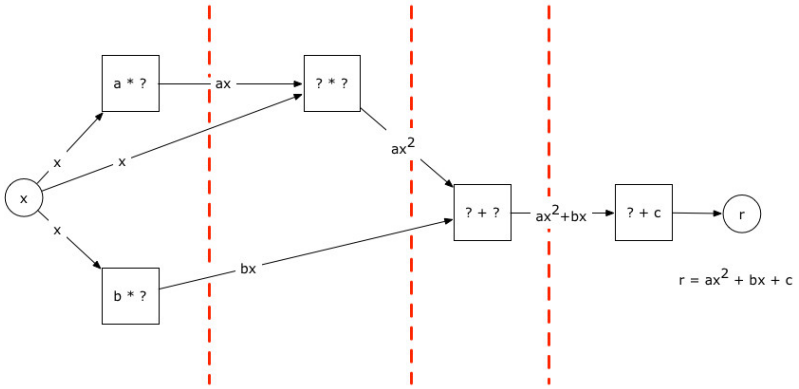

Figure 4: Segmented Net

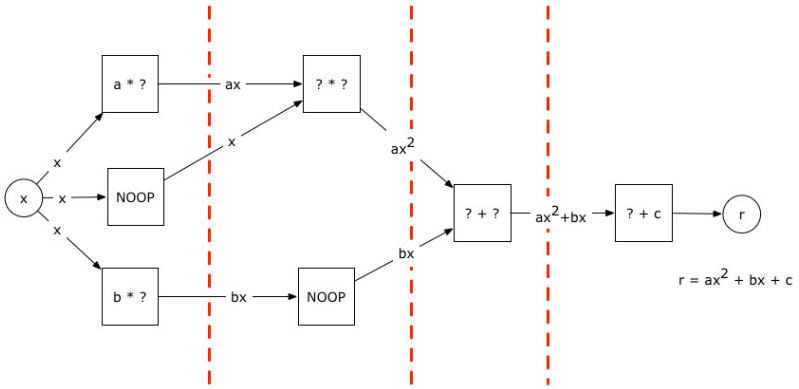

Figure 5: NOOP Net

From this stage, a layout process takes over, mapping each state vectors into a layer (line) of cores, reordered where appropriate to reduce the lateral movement of data. Additional layers are added, as necessary, to implement the 'plumbing' between layers. It is not sufficient to simply plot the most direct route because of the systolic constraint that tokens from the same wave cannot use the same path twice. We execute an efficient heuristic to obtain a layout and fallback to a Turing complete, but inefficient, algorithm if the heuristic fails.

3.4 Special FFT

For important algorithms such as the FFT, which is dominated by the cost of data transfers, the general technique does not generate a sufficiently good solution. Consequently we turned to a more specialised approach that respects the general approach but produces more compact code.

The FFT has a recursive implementation. This is compactly expressed in Python3 (below) and this implementation is useful to gain an insight into how the static dataflow is laid out.

```
import cmath
from cmath import pi
from functools import lru_cache

def recfft( x ):
  '''Recursive fast fourier transform.'''
  # len( x ) is n, the size of the input vector.
  if len( x ) <= 1:
    return x
  else:
    ( even, odd ) = separate( x )
    y = recfft( even )
    z = recfft( odd )
    lhs = [ y[i] + twiddle( i, len(x) ) * z[i]
      for i in range( 0, len(y) ) ]
    rhs = [ y[i] - twiddle( i, len(x) ) * z[i]
      for i in range( 0, len(y) ) ]
    return lhs + rhs

# Memoise the factors.
# 2j is literal for twice the square root of -1.
@lru_cache()
def twiddle( a, b ):
  return cmath.exp( -2j * pi * a / b )

def separate(x):
  '''
  Given a list x of length 2**N, unzips it into two sublists
  of the alternating members. The first list is all the
  even-numbered members and the second list is all
  the odd numbered members. If the list-length is not
```

```
a power of 2 we'll eventually get an index error,
protecting us from silent failure.
'''
return(
  [ x[i] for i in range( 0, len(x), 2 ) ],
  [ x[i] for i in range( 1, len(x), 2 ) ]
)
```

The crucial inductive step is the separation of the even and odd numbered elements of the input vector, which are each transformed recursively, followed by combining their values together. This gives rise to the insight that this inductive step can be decomposed into a series of n-permutations and map operations. The block diagram in Figure 6 illustrates this.

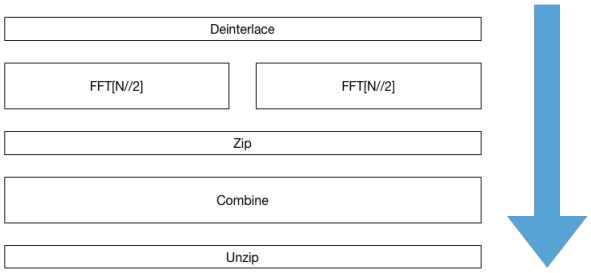

Figure 6: Recursive Decomposition of FFT

This exposed a particular weakness in our compiler toolkit. From the viewpoint of our architecture, the FFT is largely a problem of reordering state vectors using permutations. Because of the systolic constraint, permutations turn into a thick layer of plumbing - areas which are dedicated to routing data from one position to another. Such plumbing is undesirable because it performs no useful computation, making use of only the network portion of the cores that are involved, and reducing the utilisation and available computational power of the chip.

As a consequence, when two areas of 'plumbing' abut each other it is essential to be able to fuse the two together. We therefore implemented an optimisation for finding successive permutations and merging them to produce a single permutation. The block diagram in Figure 7 unrolls a level of recursion to show how successive permutations arise and the opportunities for their fusion.

3.5 Special Match-Filter and Energy Sum

The match-filter is simply the Hadamard product with a fixed complex vector, followed by mapping the absolute-square function over the vector

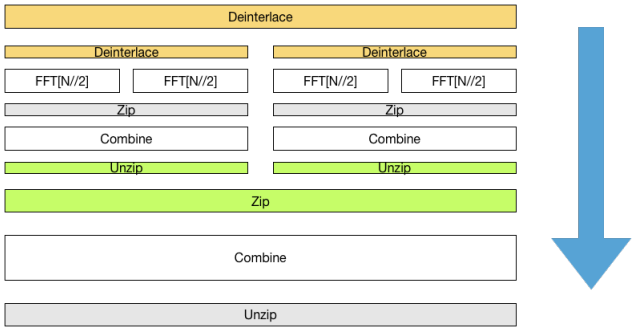

Figure 7: Fused Permutation

and finally taking the sum. All three steps had pre-existing library functions for real-valued vectors, so the work involved was generalising them to complex numbers.

Dealing with complex numbers raised two specific technical problems. The first problem is that complex vectors can exploit the fact that in each direction a core has two inputs and two outputs, that is to say precisely corresponding to real and imaginary components, or as alternating real/imaginary components with only one per core. We call these packed and unpacked representations and we needed to deal with both of these. Because of the need to have spare channels, the unpacked representation is typically more useful, which is, perhaps, unexpected.

The second problem is more fundamental, which is that the complex arithmetic implementation uses the Cartesian representation where the real and imaginary components are separated. This is not consistent with the current mathematical models of transcomplex arithmetic, which preserve angular information when adding or multiplying numbers with an infinite or nullity modulus [3] [1] [2]. Implementing this accurately would undoubtedly add a very considerable overhead to the implementation whilst the practical benefit would be very small. This remains an outstanding issue.

The solution was implemented by a custom library component whose argument is the number of complex data points, n. This generates code to compute FFT(n), the Fast Fourier Transform of size n. This component orchestrates lower-level compiler components to generate the configuration for a rectangular area of the chip that implements the FFT, followed by the match-filter, followed by the energy sum.

The custom solution was verified in simulation against two different implementations of the FFT at different sizes for a series of input vectors provided at sequential clock ticks.

Figure 8 shows a generated solution for FFT(16). The squares correspond to cores and the lines show how cells are connected from one to another; disconnected lines show optional inputs that are unused. Cores that are unshaded are used purely for routing whereas cores that are shaded are used for their arithmetic capability as well - the ratio of shaded to the total number of cells is the effective utilisation - and highlights the

significance of the plumbing overhead.

The flow of the solution is from left to right. The blank left hand columns (4 columns) show the initial permutation. Implementations of the FFT, embedded in radar and other signal-processing systems, often assume that data is presented pre-permutated. Hence these columns would be removed from such an implementation. The FFT itself occupies the 21 columns on the left. The right hand edge shows the tree-like reduction of the match-vector into an energy sum (3 columns) and to their right the two shaded-layers (7 columns) is the Hadamard product with the match-vector in unpacked format.

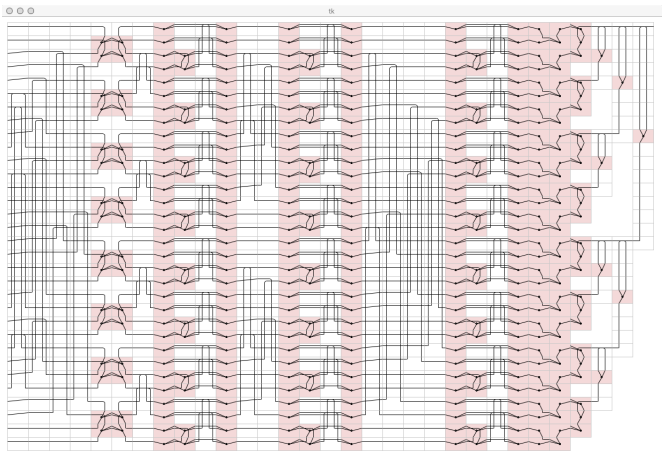

Figure 8: FFT(16)

Figure 9 shows just the FFT portion of the solution, annotated by hand to indicate the breakdown of the various layers. The thick borders indicates the recursive decomposition, where the upper and lower halves are the even/odd halves.

3.6 Results

The results are shown in Table 1. Programs are compiled for FFT(N) with N running from 2 to 256, inclusive, in steps of powers of 2. The corresponding number of real Inputs is 2N. The investigation focuses on the variable, FFT, part of the challenge, neglecting the Hadamard product, which is of fixed with, and the width of the tree sum, which is given, analytically, by $\log_2(2N) - 1$. The Width and Height of the array of cores is given for computing just the FFT. Note that the height is fixed at 2N by design. The travel time (TT) of a single wave (latency) is also given. Note that for slipstreamed solutions the latency represents the time for a

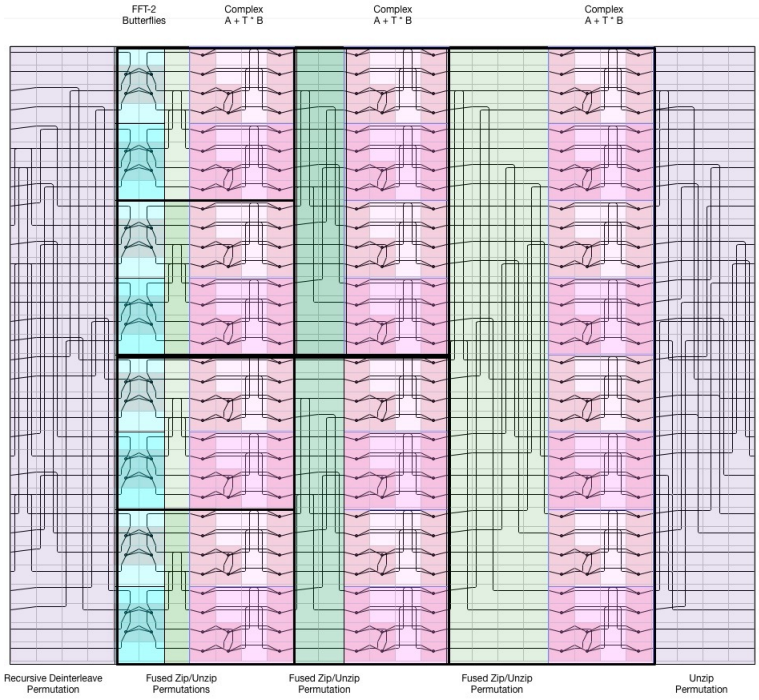

Figure 9: Annotated FFT(16)

single input vector to be processed but the throughput for a slipstreamed solution is one vector per clock-tick.

The Width of the solution depends heavily on the quality of the solutions found for the permutations. Inspection of Figure 10 shows that the Width is a good approximation to linear as 0.8N and inspection of Figure 11 shows that the travel time (TT) is a good approximation to linear, as 6.7N. The Width is visualised in Figure 12.

N	Inputs	Width	Height	Cells	TT
2	4	5	4	20	12
4	8	12	8	96	30
8	16	20	16	320	59
16	32	31	32	992	115
32	64	48	64	3,072	221
64	128	79	128	10,112	437
128	256	133	256	34,048	860
256	512	237	512	121,344	1,717

Table 1: Results For FFT(N) Only

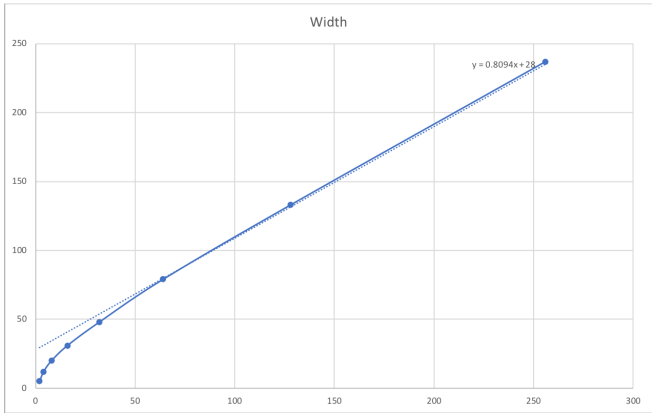

Figure 10: FFT Width

4 Conclusion

We demonstrate a fully pipelined Fast Fourier Transform, in the complex domain. We examine relatively small input vectors of size N complex

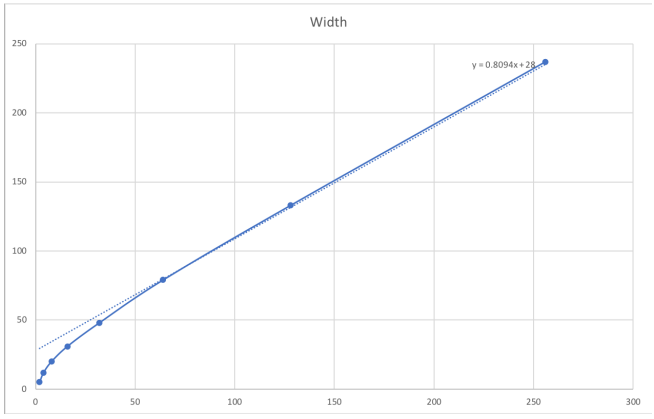

Figure 11: FFT Latency

terms, corresponding to 2N real terms and find that a simulated chip would process input vectors at the fastest possible rate (that is one vector per clock tick) with a latency slightly less than 7N clock ticks.

Larger input vectors require solutions that cross chip boundaries. This presents issues that have not been addressed here, such as reduced data-rates and longer delays. Commercial solutions typically use only real-valued input vectors, which leads to a substantial simplification in conventional processors. We expect the same to also apply to our architecture.

A substantial part of the area consumed by the solution is due to the necessity to implement non-interfering pipeline queues ('plumbing') to move vector data from one row to another. If the input data is presented at a lower rate then it is possible to very substantially reduce the width of the solution by employing fixed delays in the connecting pipelines. We did subsequently investigate this and, for example, the width of the FFT(256) was reduced from 237 to just 41 in the extreme case of inputing each signal at a speed of one datum per clock tick.

References

[1] T. S. dos Reis and J. A. D. W. Anderson. Transcomplex topology and elementary functions. In S. I. Ao, Len Gelman, David W. L. Hukins, Andrew Hunter, and A. M. Korsunsky, editors, *World Congress on Engineering*, volume 1, pages 164–169, 2016.

[2] T. S. dos Reis and J. A. D. W. Anderson. Transcomplex numbers: properties, topology and functions. *Engineering Letters*, 25(1), 2017.

[3] Tiago S. dos Reis and James A. D. W. Anderson. Construction of the transcomplex numbers from the complex numbers. In *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2014, WCECS 2014, 22-24 October, 2014, San Francisco, USA.*, volume 1, pages 97–102, 2014.

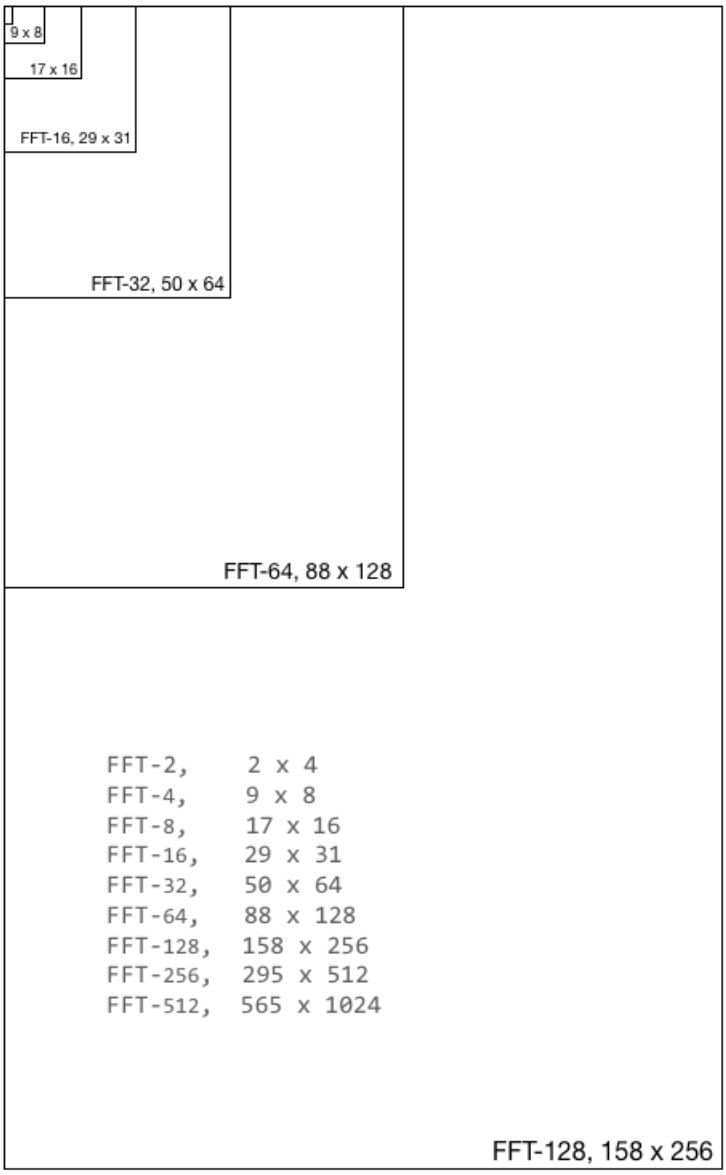

Figure 12: FFT Area of Cores