Custom MIMD Processor for Parallel Algorithm Applications

Thesis by
Hector Wilson

In Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science in Electrical Engineering

CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California

2024

Hector Wilson
ORCID: 0009-0005-9431-8486

ACKNOWLEDGEMENTS

Thank you to the Caltech EE department for their continued teaching and support during my time as an undergraduate at Caltech. I extend my heartfelt appreciation to my thesis advisor Professor Glen George for his mentorship and insightful feedback through the duration of the past year as I worked on this thesis. I am grateful to Caltech for fostering an academic environment that has aided me in all my endeavors.

ABSTRACT

Both industry and academia seek evermore powerful and faster computing systems to run algorithms with huge amounts of data. Hardware mechanisms to accelerate these algorithms utilize customized processor architectures to expedite compute time. There are domain-specific architectures and systolic arrays which offer very large speedups but are specifically designed for a target algorithm and are not programmable. GPUs are popular SIMD hardware accelerators that can greatly decrease runtime for algorithms while remaining programmable. However, little work has been done on programmable MIMD type hardware accelerators. These types of processors enable exploitation of instruction level parallelism which GPUs do not and could potentially offer an edge in performance for some algorithms. In this paper, a configurable MIMD processor is presented for implementation on an FPGA or custom ASIC. This processor is programmed to tackle various parallelizable algorithms from Smith-Waterman and knapsack 0-1 to creating a digital down converter for a software-defined radio.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Background on Computer Architecture

There is a constant effort in science and industry to accelerate the execution of algorithms on computers. This effort towards high performance computing has led to a greater degree of scientific advancement through accelerated simulation for quantum chemistry, computational fluid dynamics, and even weather modeling. Additionally, many large technology companies have been pursuing specialized processor architectures for accelerating large matrix computations in neural networks. Numerous domain-specific hardware architectures ranging from graphics processing units (GPUs) to tensor processing units (TPUs) have been created in the past decade seeking to reduce latency and accelerate algorithms for specific applications. Aside from domain-specific solutions, there is also a constant effort to improve performance amongst general processor architectures such as the multi-core CPUs developed by Intel and AMD among others.

In parallel with advancements in processor architectures, there has been a growing emphasis on software to leverage the capabilities of these specialized processing units. Researchers and engineers are constantly refining algorithms to exploit parallelism, minimize memory access overhead, and reduce computational bottlenecks. Techniques such as parallelization, vectorization, and algorithmic optimizations play a crucial role in maximizing the performance of applications running on modern computing systems. Furthermore, the development of high-level programming frameworks and libraries, such as CUDA for GPUs and TensorFlow for neural networks, has simplified the process of programming and deploying parallel applications. As hardware and software continue to evolve in tandem, the pursuit of faster and more efficient computing solutions remains at the forefront of scientific and technological innovation.

The numerous processor architectures can be characterized by the 4 classifications of Flynn's taxonomy (Figure 1.1): Single Instruction Stream, Single Data Stream (SISD), Single Instruction Stream, Multiple Data Streams (SIMD), Multiple Instruction Streams, Single Data Stream (MISD), and Multiple Instruction Streams, Multiple Data Streams (MIMD). SISD architectures are prevalent in single

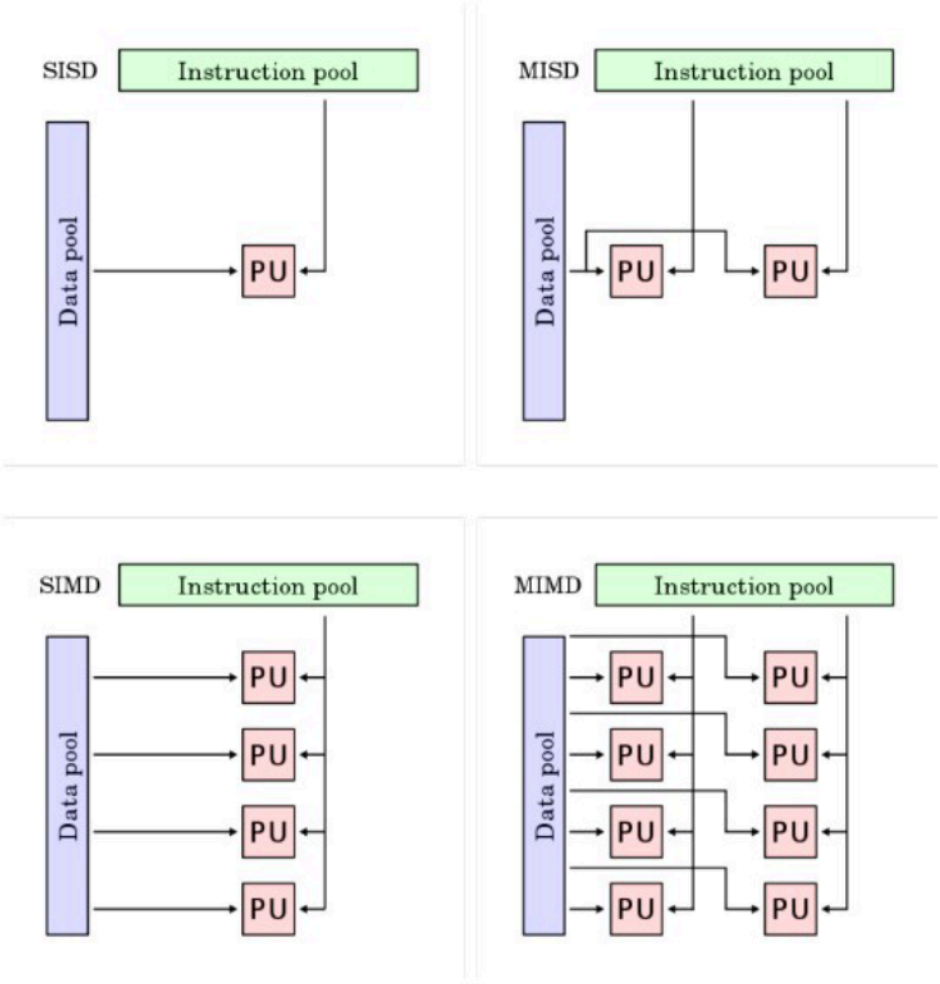

Figure 1.1: Flynn's taxonomy of computer architectures.

core CPUs. Most modern multi-core CPUs, however, exhibit MIMD, an architecture where different processing units (PU) or cores can be running independent programs with different data. Most hardware accelerators and domain-specific processors, such as GPUs, exhibit a Single Instruction, Multiple Thread (SIMT) mixed architecture that most closely resembles SIMD. That is, the multiple threads within a block in a GPU all execute the same instruction but draw from different data. The most rare is the MISD type of processors but an example is the Space Shuttle flight control computer where multiple cores would run different programs on the same set of data.

Historically speaking, MIMD computers are a relatively rare-breed of computers due to the complexity needed to design and program them. There are two categories of MIMD computers based on the memory system they use, shared memory and distributed memory. Shared memory MIMD computers use a shared common

memory space across all PUs in the processor. Distributed memory MIMD computers contain local memories, one for each PU. Distributed memory MIMD computers, then, must communicate through message passing systems. Within distributed memory MIMD computers, there are an additional two categories: massively-parallel processors and computer clusters where the former uses hardware interconnects and the latter uses networks. An early example of a shared memory system was the Cray-1 supercomputer developed in 1976 by Seymour Cray. Cray-1 revolutionized high-performance computing with its novel MIMD architecture featuring multiple vector processors working in parallel. Its design allowed for significant performance gains in scientific applications from fluid dynamics to weather modeling, making it one of the most iconic supercomputers of its time. On the other side of things, the Intel Paragon supercomputer was a distributed memory MIMD system developed in 1982. The Paragon featured distributed memory with a large grid of compute nodes powered by Intel microprocessors. The Intel Paragon was used for a wide range of parallel computing tasks, including numerical simulations, data analysis, and computational fluid dynamics. Most modern supercomputers are distributed memory MIMD systems. Fugaku is a distributed memory supercomputer developed in Japan between Fujitsu and RIKEN which became operational in 2020. Fugaku features over 158,978 Arm-based A64FX processor nodes and custom Tofu-D interconnects which enable unparalleled computing capabilities and stands as the world's fastest supercomputer as of 2024. Like its predecessors, Fugaku has been used in a wide range of applications from climate modeling to artificial intelligence.

Systolic arrays are SIMD-type digital logic systems designed specifically for the target algorithm. These systems offer the most pronounced speed-up due to their simplicity and resulting lack of flexibility. Systolic arrays are the topic of most research papers seeking to design systems that offer the largest speedup for a given algorithm. However, what if you want a processor that can be reprogrammed to compute many different algorithms? The first thing that comes to mind is general purpose GPUs which have become popularized in recent years due to their ability to accelerate parallelizable algorithms while having extensive open source guidelines that makes programming them relatively easy to learn. However, an MIMD architecture that offers instruction level parallelism as well as having many PUs like a GPU should accelerate many algorithms even further. The pursuit of adding many features to large SoCs, as well as having a limited core count, have led to the common MIMD multi-core CPU seen today being quite slow. However, if one were to strip an MIMD computer down to the bare necessities, the benefits of this

added level of parallelism should be revealed given a reasonably large amount of PUs within. In this research paper, an MIMD processor is constructed used to tackle a series of applications such as Smith-Waterman and the knapsack 0-1 algorithms. Aside from the common multi-core desktop CPU, little literature exists on using MIMD processors to accelerate these algorithms, so this paper hope to shed some light on how the performance varies with this architecture.

1.2 Background on Parallel Processing Applications

Smith-Waterman Algorithm:

The Smith-Waterman (SW) algorithm is a highly parallelizable NP-hard algorithm that does local sequence alignment and is used in genetic sequencing to find similar regions among two nucleic acid or amino acid strings. Unlike global alignment algorithms such as Needleman-Wunsch, which aim to align entire sequences, SW focuses on identifying local regions of similarity, making it particularly suitable for detecting conserved motifs, domains, or functional regions within larger sequences. The algorithm operates by filling out a scoring matrix of dimension n by m, where n and m are the sizes of the two sequences, and then tracing back to find a pair of sequences corresponding to the most similar region between the two sequences. While there are many variations of this algorithm this paper will only focus on the local linear gap cost version. In this version of the algorithm, there must be a defined match/mismatch score (s(a_i,b_j)), which is set as +3/-3 for this project, as well as a gap penalty (s(a_i, –) for a gap in sequence b or s(–,b_j) for a gap in sequence a) which is set to -2. Refer to the image below demonstrating the algorithm at work using this scoring criteria:

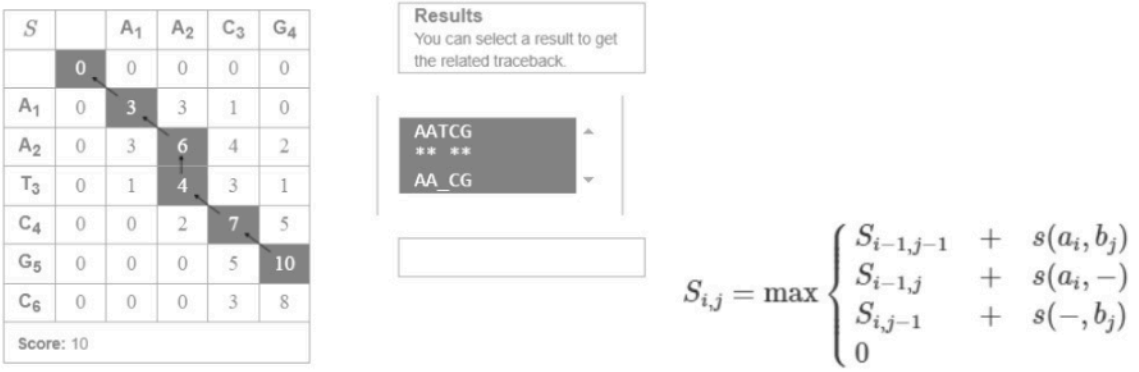

Figure 1.2: Example 5x5 Smith-Waterman Scoring Matrix

The algorithm starts by situating two DNA sequences on the top and left sides

of the scoring matrix as references. For example, the reference on the top of scoring matrix could be a DNA sequence from a large database and the reference on the left could be a specific target one is trying to find in the database (Top = $DNA_{database}$ or a, Left = DNA_{target} or b). Each cell $S_{i,j}$ in the scoring matrix is computed by comparing and taking the maximum of 3 numbers. To get the first of these, the cell will take its top-left neighbor $S_{i-1,j-1}$ and add the match score if $a_i = b_i$ or mismatch score if $a_i \neq b_i$. The resulting number from this operation represents adding the nucleotides a_i and b_j to the traceback sequences. Second, the cell will take its left neighbor $S_{i-1,j}$ and add the gap penalty. This corresponds to adding the nucleotide a_i to one sequence and a gap (-) to the other sequence. Third, the cell will take its top neighbor $S_{i,j-1}$ and again add the gap penalty. This corresponds to inserting a gap (-) to the top sequence and the nucleotide b_j to the other sequence. The cell will take the three numbers from these operations and find the maximum of them to become its value $S_{i,j}$. Lastly, if this maximum value is less than zero then the cell is set to 0. This is how the scoring matrix is filled out.

Dynamic programming comes into play as a means of speeding up the compute time for this scoring table. Each cell scoring matrix is computed based on the results from its respective top, left, and top left cell. Thus, cells on the anti-diagonals along the scoring matrix, top right to bottom left, can be computed in parallel. This is the basis for the SW algorithm and this high level of parallelism makes it perfect for implementation with SIMD systems. In fact, much research has already been done accelerating this algorithm in SIMD systems like GPUs and novel systolic arrays. A common metric for evaluating the performance of these hardware systems is cell updates per second (CUPS), where cell refers to a cell in the scoring matrix.

Knapsack 0-1 Problem:

The knapsack problem is another NP-hard problem involving dynamic programming that does combinatorial optimization. The problem is formulated as follows: Given a set of items, each with a weight $\{w_1, w_2, ..., w_n\}$ and a profit $\{p_1, p_2, ..., p_n\}$, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total profit is as large as possible. For this paper, the focus will be on the 0-1 knapsack problem in which you either include or exclude each item and don't allow multiples of any item. Formally, this algorithm seeks to do the following:

$$\text{maximize: } \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to: } \sum_{i=j}^{n} w_j x_j \leq W \text{ and } x_j \in \{0, 1\}$$

where: p_j is the profit of item j
x_j is either 0 or 1 (include/exclude item j)
w_j is the weight of item j
W is the max capacity of the knapsack

Much like Smith-Waterman, the knapsack 0-1 problem requires an (n) x (m) table to be filled. This time, n is the number of items and m is one more than the max capacity of the sack (m=W+1). Each column (i) in the table corresponds to a weight from 0 up to the actual max weight W. Each row (j) in the table corresponds to an item and will have a corresponding weight w_j and profit p_j. With this in mind, each cell in the table is computed as follows:

$$S_{i,j} = \begin{cases} 0 & \text{if i = 0 or j = 0} \\ S_{i,j-1} & \text{if j > 0 and i < } w_j \\ max(S_{i,j-1}, p_j + S_{i-w_j,j-1}) & \text{if j > 0 and } i \geq w_j \end{cases}$$

Based on the formula above, each cell in a row can be computed simultaneously, and is the parallelism targeted by computing systems. Again, like Smith-Waterman, much research has been done implementing the knapsack problem with systolic array SIMD type systems as well as GPUs and will be the subject of comparison against our MIMD processor.

Digital Down Converter:

Both Smith-Waterman and knapsack 0-1 contain a high degree of parallelism that SIMD systems are able to exploit. In order to explore an area that would benefit from instruction level parallelism in MIMD processors, this paper looks towards digital down converters (DDC) for software-defined radios (SDR). A DDC is typically composed of a cascaded integrator-comb (CIC) filter followed by a finite impulse response (FIR) filter. In practice, when a part of an FM receiver,

a DDC operates on I and Q input data samples fed out of an analog-to-digital converter (ADC) and then outputs this filtered data for further bandwidth reduction and eventual demodulation. In this case, a CIC filter is used as a decimator and has the structure shown in Figure 1.3.

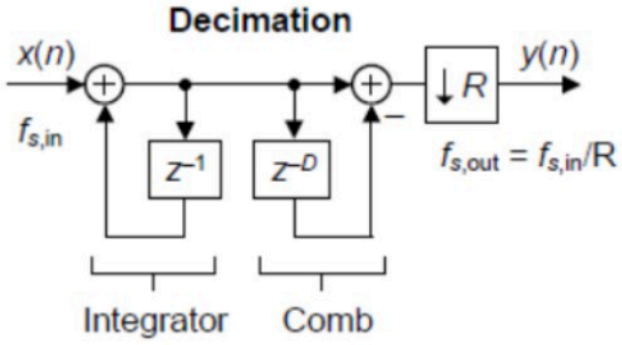

Figure 1.3: CIC Filter Decimator Diagram

CIC filters are technically a class of FIR and lowpass filters. They are able to attenuate frequencies that lie outside the passband range and the downsampling factor causes all input samples to be discarded except for every Rth sample. A stand alone CIC filter as pictured above, however, will have numerous negative effects on the signal quality. In order to improve anti-aliasing and image-reject attenuation, a common method is to cascade multiple stages of CIC filters. In hardware, CIC filters can be implemented by converting delay components into registers or flip-flops. Even after cascading multiple CIC filters there remains some negative effects in the frequency response. Namely, CIC filters naturally tend to have a drooping, rather than flat, main lobe in the passband region. This is why FIR filters are typically used on the CIC filter output in order to flatten the passband gain and sharpen the CIC frequency rolloff. FIR filters take the structure shown in Figure 1.4.

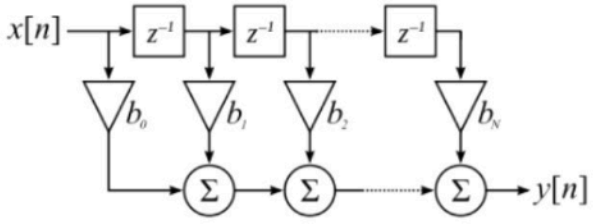

Figure 1.4: FIR Filter Diagram

Again, delay components can be realized by flip-flops or registers but unlike CIC filters, FIR filters require multipliers. Commonly, an FIR filter is said to contain

N+1 taps, or mathematically speaking, FIR filters perform a weighted average of the N+1 most recent input samples. In most systems, FIR filters contain over a hundred taps. The mathematical representation of this is the following:

$$y[n] = b_0x[n] + b_1x[n-1] + ... + b_Nx[n-N]$$

$$y[n] = \sum_{i=0}^{N} b_ix[n-i]$$

where: N is the number of taps

b_0 to b_N are the FIR coefficients

x[n] to x[n-N] are the N+1 most recently input samples

y[n] is the FIR filter output

The greater the number of taps in an FIR filter, the better response will be observed on the output. However, increasing the tap count also requires a significant increase in the necessary hardware. Multipliers in particular require a lot of power and an extensive amount of hardware to implement, especially when working with larger registers widths. A team of researchers from Illinois Institute of Technology programmed an FIR filter with CUDA onto an Nvidia GeForce 8800 GTS 512 GPU and found a x100 speedup in comparison to standard C++ code on an Intel Core 2 6400 running at 2.13 GHz (Rebacz, Oruklu, and Saniie, 2010). DDCs are most typically implemented with specialized hardware or with RTL on an FPGA, so this paper seeks to compare these implementations to the programmable MIMD processor. Instruction level parallelism and other techniques are used across the MIMD processor to program a full DDC with some PUs running CIC and other PUs running FIR filters.

In summary, this paper will present an MIMD processor architecture capable of running highly parallelizable algorithms. The paper will demonstrate on this processor implementations of common SIMD algorithms such as Smith-Waterman and knapsack 0-1. Additionally, to explore the power of instruction level parallelism in MIMD systems, the paper will discuss an attempt to program a digital down converter (DDC) onto the processor to operate on an input data stream.

Chapter 2

METHODS

2.1 Top-Level Architecture

In order to tackle the aforementioned algorithms and others, an MIMD processor was constructed consisting of many PUs with various fixed interconnects used to communicate between them. The MIMD processor was designed in VHDL for implementation on a Spartan 6 FPGA. The VHDL code describing the MIMD processor can be found in appendix B. Many components of the processor are parametrized for customization depending on the intended set of algorithms. When implemented on an FPGA a variable 'c' number of PUs can be created for the system. Additionally, a variable 'm' number of multipliers can be instantiated to communicate with the last m PUs in the MIMD system. Due to this, only the last m PUs can be programmed to do the MUL instruction. Each of the PUs contain 16 registers, each register with a width of 18-bits. This width was chosen to accommodate more intensive DSP algorithms and for usage in the DDC, although the registers width and count is also configurable. The PUs contain a reduced instruction set designed to contain a minimal number of instructions necessary to produce the targeted algorithms and other similar algorithms. The instruction set for the PUs consists of a series of generic load, branch and skip instructions (Table A.1); a series of arithmetic instructions (Table A.2); and a series of communication instructions used to transfer data between PUs (Table A.3). All of these tables with the complete instruction set can be found in appendix A.

Using these instructions one is able to implement the knapsack 0-1 and Smith-Waterman algorithm as well as program a DDC onto the system. The interconnect instructions listed in appendix A.3 enable the PUs to communicate with each other as they operate. The IN/OUT instructions allow each of the PUs to read/write from/to a total of 4 interconnect buses, IO(0-3). IO(0) and IO(2) are inputs to a PU while IO(1) and IO(3) are outputs. When writing algorithms for the PU is it necessary to ensure that the program only ever does IN instructions with an address of 0 or 2 and OUT instructions with an address of 1 or 3. Additionally, there are three synchronization instructions. SYNC is a global sync which will stall a PU until all other PUs execute a SYNC as well. The local synchronization instructions L1SYNC and L2SYNC

are parametrized to connect a variable number of PUs anywhere in the processor to synchronize them independently. Also worth noting, the processor will output the last component of the global SYNC bus to communicate with external devices that the first PU is ready to read in data. The first unit of the MIMD processor is the only PU that has external facing inputs as the input bus IO(2) will connect externally. This complete interconnect architecture between PUs (or CPUs) can be visualized in the block diagram of Figure 2.1.

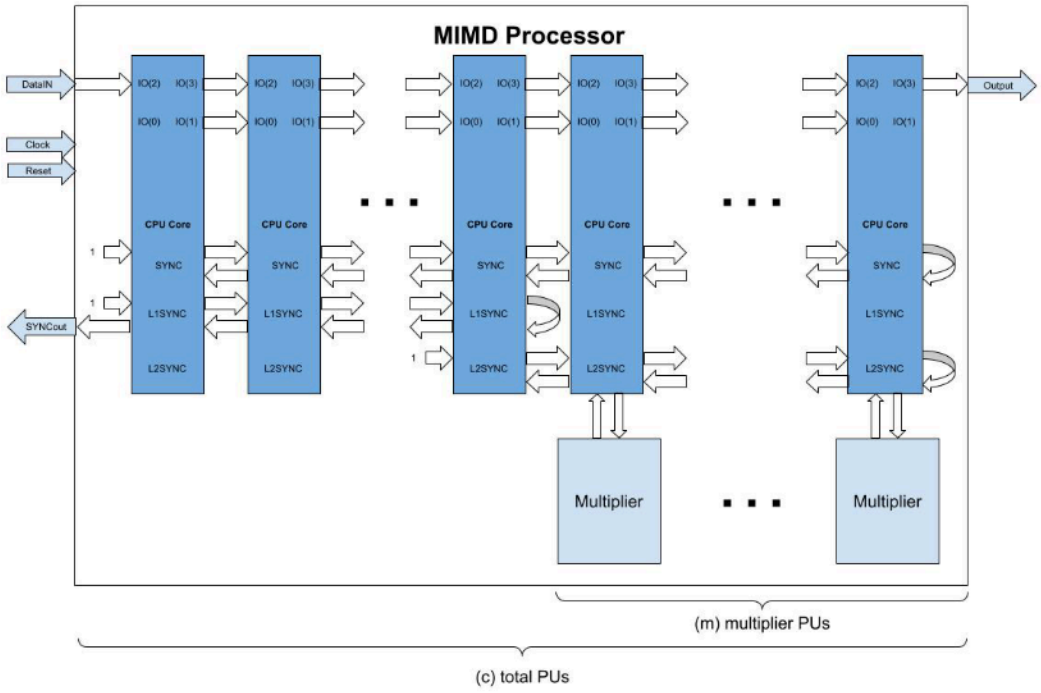

Figure 2.1: Top-level Processor Interconnect Architecture

2.2 Processing Unit Architecture

Within each PU there is a basic arithmetic logic unit (ALU), status register, control unit, register array, program memory address unit (PAU), program memory, and a stack. The PUs do not contain any data memory and data for any algorithm must either be loaded in through program memory, using LDI instructions, or shifted into the PU from an external source through DataIN (connected to IO(2) on the first PU). As mentioned before, each PU contains 2 data input buses and 2 data output buses that can be used to shift data through all the PUs. The register array contains 16 registers each with a width of 18 bits. There are two register array output lines

that feed registers to the ALU, IO ports, and elsewhere in the system. There is one input line for the register array used to write data to the registers.

In Figure 2.1 synchronization interconnects run into each PU on one side where they are AND-gated with an internal control signal that toggles when the PU is on a sync instruction. When this control is on, the SYNC In connects to SYNC Out. With this mechanism, once all connected PUs are on a sync instruction the interconnect effectively turns into a wire which is then fed back through the PUs. When a PU sees the back-propagating sync toggle, it knows that all of the PUs are synced and it thus deasserts the internal sync control signal, allowing the PU to move on to the next instruction.

The control unit decodes the machine code instruction and produces control signals to the other blocks of the PU. The full instruction set can be found in appendix A. For simplicity's sake, the architecture does not include any double word instructions nor data memory instructions. The stack is configured to have a relatively small depth of only 8 bytes which while small, has enough space for the algorithms targeted here in this paper. The program memory is configured to have

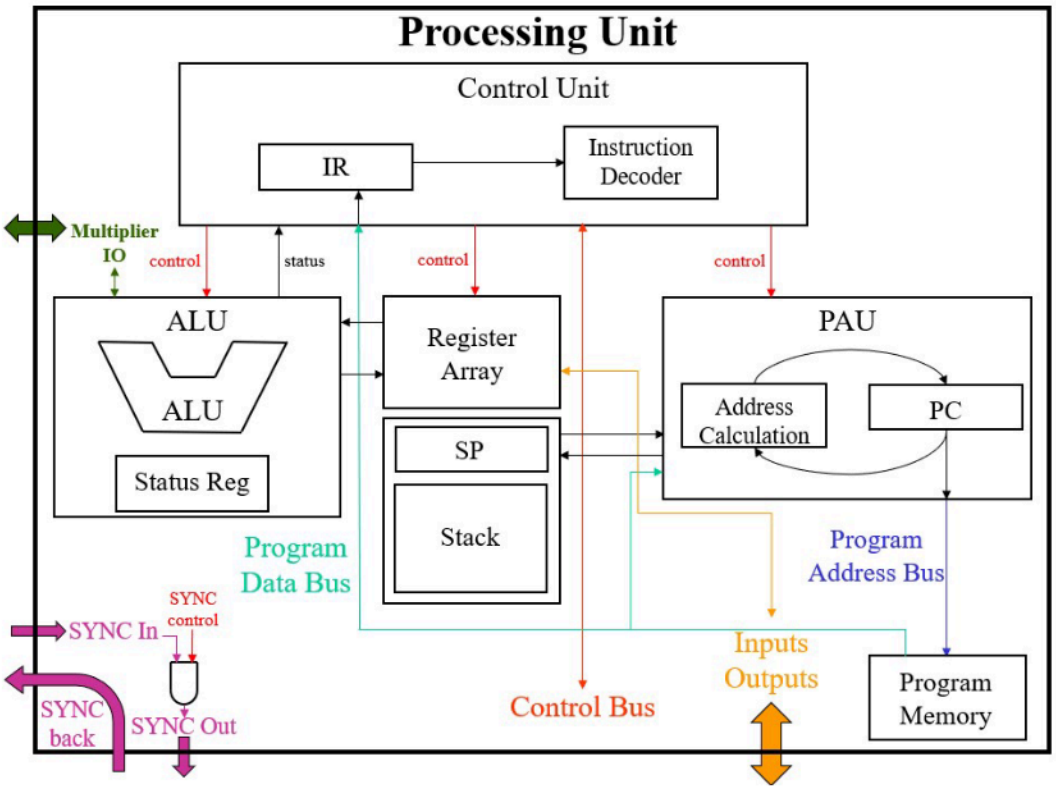

Figure 2.2: Processing Unit (PU) Block Diagram

128*2 bytes or 128 words of instructions. For implementation on an FPGA, the stack was configured as read and write block RAM and the program memory was configured as a read-only block RAM. A full block diagram for each PU is included in Figure 2.2.

The program counter (PC) is used to address the program memory. The PAU takes in the PC as a source and an array of offsets with control signals indicating which offset to select for an operation. Depending on the instruction, the PAU either increments or decrements the PC source and then, depending on if the operation is doing pre/post addition, adds the offset to either the incremented/decremented PC or plain PC. Therefore, there are a total of two 16 bit adders in the PAU. The PAU outputs both the offset-added PC as well as the incremented/decremented PC. Under normal operation, the PC will just be incremented but for jump or branch instructions an offset is added to the PC. These jump instruments can be found in the appendix A.1. A more comprehensive block diagram of the PAU can be found in Figure 2.3.

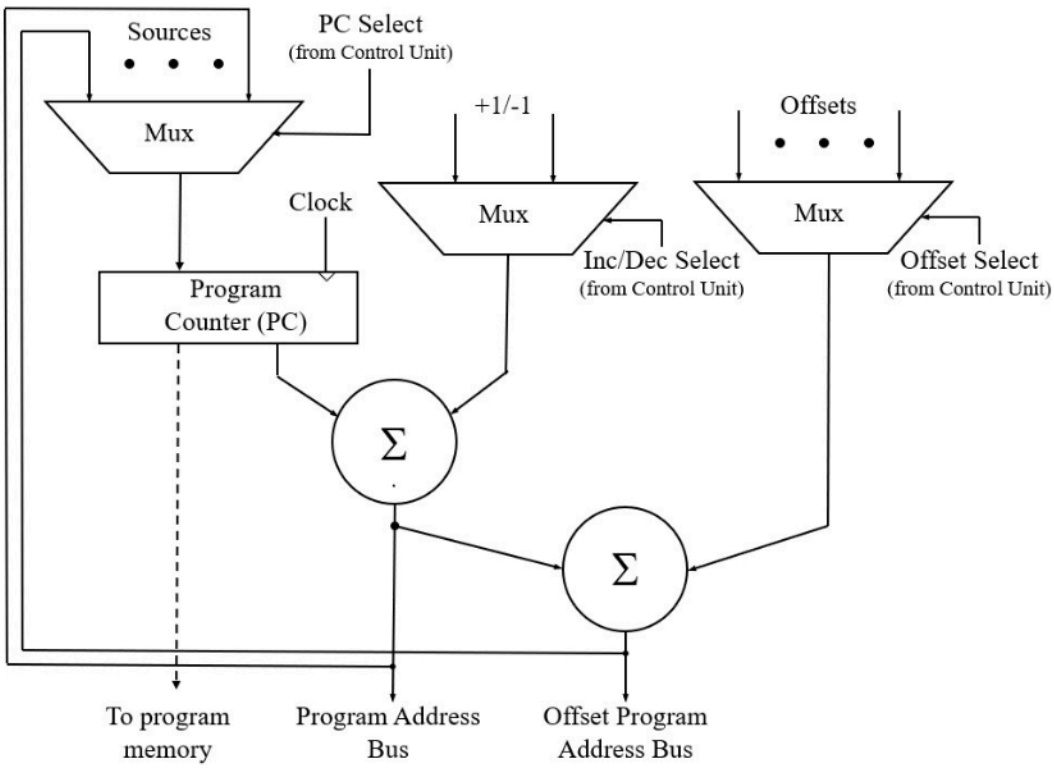

Figure 2.3: Program Address Unit (PAU) Block Diagram

The ALU is register-based and can perform all basic operations on up to two operands, operand A and operand B. The ALU is composed of three blocks: the

shift block, F logic block, and adder. The adder takes in the result of the F block as its second operand to perform operations like subtraction. After computing the result for a given instruction the status register connected to the ALU will compute the flags from the operation. Different instructions will affect different flags. In total, there is the carry flag, zero flag, negative flag, overflow flag, sign flag, and half-carry flag (C, Z, N, V, S, H). A description and the effected flags of each arithmetic operation for the ALU is listed in the appendix A.2. A block diagram of the ALU can be found in Figure 2.4.

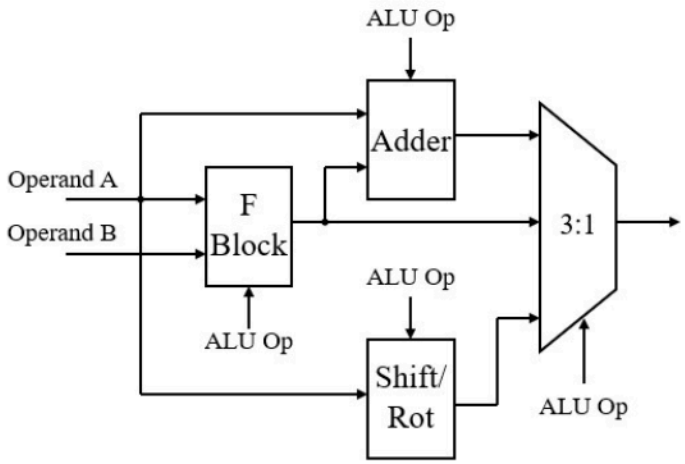

Figure 2.4: Arithmetic Logic Unit (ALU) Block Diagram

2.3 Multipliers

Additionally, as mentioned before, one may allocate external multipliers to communicate with the PUs intending to perform the MUL instruction. For FPGA implementation, these multipliers were created from the DSP units embedded within the Spartan 6 FPGA. The multipliers are configured to do signed multiplication and therefore the two operands are signed extended to 36 bits before multiplying them. The product then has 72 bits though only the high order 36 bits are used. The MUL instruction lasts two clocks and thus the low 18 bits are stored on the first clock and the high 18 bits are stored on the second clock.

2.4 Knapsack 0-1 Adaptation

In the following sections, the paper will discuss how the processor architecture was adapted for the various algorithms tackled in this paper. The assembly code

programmed onto each PU can be found in appendix C. The knapsack 0-1 problem is the first algorithm this processor was adapted for. For this algorithm, the processor was configured to have no multipliers as they are unnecessary here. The algorithm was written to fill out the scoring matrix abiding by the rules set by the knapsack 0-1 problem. Data for this algorithm was stored through program memory, rather than shifting in data externally. The LDI instruction used to load in data can only store 8 bits of data, however, despite the PU using a register size of 18. Since program memory is used to store item profits and weights, the size of program memory will grow linearly as one increases the size of the scoring matrix or number of items to choose from. The knapsack 0-1 algorithm uses global synchronization to synchronize all PUs as the algorithm performs its computations. The output of the knapsack 0-1 algorithm will be contained in the bottom right hand corner of the final scoring matrix. This value is output by the last PU to IO(3) which is connected to the output of the MIMD processor.

As mentioned earlier, in the knapsack 0-1 scoring matrix each cell in a row can be computed simultaneously. Therefore, each PU in this hardware adaptation is responsible for filling out a single cell in each row or a column of the complete scoring matrix. As each PU is responsible for a column, each will have intermediate capacity of 0 up to the actual max weight W. Thus, the max capacity for the knapsack 0-1 problem is limited by the number of PUs able to fit on an FPGA. On the flip side, all the items weight's and profit's are stored in program memory so the number and values of items able to be included depend only on the amount of program memory or block-ram available for usage. Additionally for this algorithm, as described in section 1.2, in each iteration or for each item, j, the PUs must share between them w_j data elements. Therefore, the compute time for each iteration of this algorithm is linear in the magnitude of w_j. The assembly code programmed onto each PU for this knapsack 0-1 implementation can be found in appendix C.1.

2.5 Smith-Waterman Adaptation

The Smith-Waterman algorithm also does not use multiplication and so no multipliers are configured for this algorithm. For SW, data is loaded in externally and so program memory size will be constant as the sequence length grows in size. Rather than global synchronization, the processor uses local synchronization 1 which tells the external memory to shift in new data. The L1SYNC is implemented to encompass all of the PUs so it acts, in effect, as a global sync. The Smith-Waterman algorithm does not necessarily have the maximal value in the bottom

right corner of the scoring matrix. Thus, after the table fill completes, a traceback algorithm should be conducted to find the most similar sequence alignment. This traceback process, however, was not approached in this project and all the focus is on the table-fill.

The parallelism in the Smith-Waterman algorithm comes from the anti-diagonals in the scoring matrix. Therefore, each PU in this adaptation is responsible for filling out one cell along the anti-diagonal of each iteration. On the first iteration, only one PU will be computing a matrix cell and the others will be computing non-matrix cells. On the second iteration, two PUs will compute matrix cells and the rest compute non-matrix cells. etc... Refer to Figure 2.5 for a detailed look at this at play. It is necessary to tell the PUs to return 0 for the cell value if it is a non-matrix cell so that it doesn't influence the actual scoring matrix results. On each iteration of the SW algorithm, each PU reads in two data elements from the neighboring PU (row above) corresponding to its top and top left neighbors. Its left neighbor will already be contained in the PU. Refer to Figure 2.1 to see how data is shared between PUs. Each PU is responsible for computing a row of the table as the algorithm progresses. Due to this, the max number of rows, or max target sequence (DNA_{target}) length, in this direction is limited by the number of PUs able to fit onto the FPGA. In the other direction, the number of columns is limited by the amount of external memory being used to store and shift the database sequence ($DNA_{database}$) into the processor nucleotide by nucleotide. The assembly code programmed onto each PU for this Smith-Waterman implementation can be found in appendix C.2.

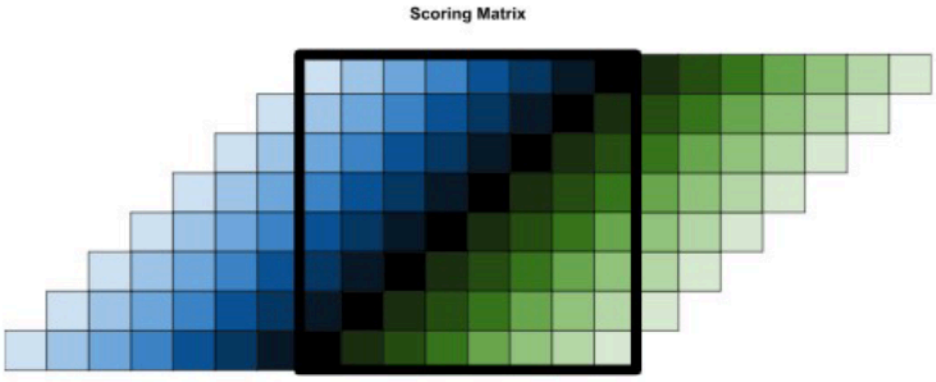

Figure 2.5: Smith-Waterman 8x8 Scoring Matrix Implementation.

2.6 Digital Down Converter Adaptation

A DDC consists of two components, the CIC filter and the FIR filter. Data is input to the MIMD processor externally and all data is represented as signed binary

numbers with a fixed binary point. In practice, this input data would be fed into the MIMD processor from a high-speed ADC. For this implementation an input size of 10 bits was used with a fixed point format of Q2.7, with a high sign bit. The input data is sign-extended to 18 bits when loading into the CIC PUs so data in the CIC filter has representation Q10.7. A total of 12 PUs were used for the DDC with 2 going towards the CIC and 10 going towards the FIR. One PU was dedicated for the CIC integrator and another PU is dedicated for the comb and decimator of the CIC filter. The CIC filter uses N = 4 stages with a decimation (R) and differential delay (M) both of value 2. This was done to accommodate B_{in} of 10 bits and a register size of 18 bits with the bit growth requirements of a CIC filter being the following:

$$B_{out} = \lceil N * log_2(RM) + B_{in} \rceil$$

Also worth noting, the gain of the CIC filter is $(RM)^N = 256$. The FIR filter contained 10 dedicated PUs each of which had its own multiplier for use in the algorithm. The multipliers are configured to do signed fixed point multiplication between the Q10.7 output from the CIC and the FIR (b) coefficients. Each PU was responsible for computing 12 taps with the entire filter containing 120 taps. The b coefficients of the FIR filter are designed to implement a lowpass filter such that the cutoff -6 dB point is 0.2π rad/sample and the passband region is $\omega \leq 0.2\pi$ rad/sample. The b coefficients were stored via program memory in each of these 10 PUs with fixed point format Q2.5 which are sign-extended to 18 bits as Q12.5. 36 bits or 2 registers were dedicated to storing the accumulated output of the FIR with a fixed point representation of Q23.12. The data itself will be no greater than Q19.12 so there are 4 extra bits and there is no possibility of overflow. This bit gain can be mathematically computed as follows:

Fixed Point Multiplication with b coefficients:

$$x[n-i] * b = Q10.7 * Q2.5 = Q12.12$$

worst case (signed numbers): $(2^{10}) * (2^2) = 2^{12}$

so max of 12 integer bits, Q12.12 in the product

120 Taps of FIR:

$$Q12.12 + Q12.12 + ... = Q12.12 * 120 = Q19.12$$

worst case: $2^{12} * 120 < 2^{19}$

so max of 19 integer bits, Q19.12 in the sum

The L1SYNC signal was used to synchronize the CIC filter PUs and indicate the system is ready to take in new Q2.7 data from an external source. The CIC gain expands this to 18 bits (Q10.7) which is then output from the CIC filter to the FIR filter. The CIC section of the processor runs at a faster rate than the FIR section so the L2SYNC signal was used to synchronize the FIR filter PUs with the decimator PU in the CIC. This signal indicates that the FIR is ready to take in data. Since the CIC filter computes much faster than the FIR filter, the CIC filter PUs are expected to stall at certain points as they wait for the FIR section to be ready to take in data. The amount of decimation done in the CIC, among other factors, can determine how long this stall lasts. The assembly code programmed on the PUs for this DDC implementation can be found in appendix C.3.

Chapter 3

DISCUSSION

3.1 Results

The knapsack 0-1 optimization problem has been studied in numerous research papers. The problem is applicable to many fields including finance, power management, and even cryptography. Many scientists have been pursuing the effort to create fast systolic array architectures to quicken the compute time for this problem involving a large number of items or weights. In 2007, Nibbelink et al. presented a systolic array design on FPGA for the knapsack 0-1 problem capable of containing up to 64 processing elements able to achieve a 16x speed-up over a 64-bit 3.4GHz Pentium 4. The design's processing elements use memory of the order O(W) where W is the capacity of the knapsack (Nibbelink, Rajopadhye, and McConnell, 2007). Escobar et al. demonstrated in 2017 a shared memory systolic array design for tackling the knapsack 0-1 problem. These researchers were able to configure their systolic array design for 58 processing elements capable of computing items with a max weight of 6144, using block-ram shared amongst groups of processing units. The results of this were a 16x speed-up over software on a Intel Xeon E5. The researchers here also found that the stall time amongst these processing elements increased as the max item weight increased and found stalls of up to 50% of the runtime when using items with the max weight of 6144 (Escobar et al., 2017). Clearly much work has gone into developing systolic arrays for this algorithm, however, GPU methods are also in circulation and offer the benefit of being implemented on programmable hardware. Huang et al. present a GPU method for computing the knapsack 0-1 problem and use a CPU for sorting the items while the GPU computes the knapsack algorithm. This mechanism takes advantage of the asynchronous call of kernel launch to overlap the sorting time on the CPU with kernel execution on the GPU. The researchers found that GPUs with a limited global memory bandwidth, such as on a RTX3070 (448GB/s), actually had a higher computing throughput in comparison to a V100 (900GB/s) because slower GPU runtime means more sorting can be done in parallel on the CPU. (Huang and Chou, 2022). This is an interesting factor to consider because in this paper the items for the knapsack 0-1 problem were assumed to be sorted beforehand.

The basic MIMD processor architecture without multipliers presented in this paper for the knapsack 0-1 problem was able to implement up to 54 PUs on a Spartan 6 FPGA, utilizing about 40,000 slice LUTs, and achieving a max clock frequency of 35 MHz. This means that the maximum capacity of the knapsack must be no greater than 53. Specifically, the MIMD processor architectures were implemented on the XC6SLX75 FPGA which contains 3096 Kb of block-ram or 3,170,304 bytes. Since each instruction's machine code is 2 bytes, the MIMD processor can store up to 1,585,152 instruction words in block-ram as program memory for the PUs. Now if a low number of 4 total items are included in the problem, the total program memory for each PU is 85 instructions. Therefore, 54 PUs require 4590 instructions to be stored in block-ram on the FPGA, this is clearly not a constraining factor (4590 < 1,585,152). Keeping the PU count at 54, each item added into consideration would add 8 instructions to each PU's program memory. This means the MIMD processor could have a maximum of approximately 3662 items to chose from in the knapsack with a capacity (W) of 53. The maximum item weight is limited by the amount of data able to be loaded into through LDI instruction which is only 8-bits so item weights can only go up to 255. Although post-load operations can be done to increase such weights beyond this limit, it would add latency to the algorithm.

The simple 4 item knapsack 0-1 problem found in appendix C took approximately 500 clock cycles to finish computing. During a portion of this compute time various PUs in the MIMD processor are stalling as they wait for other PUs to complete tasks. The previous systolic array accelerators presented above for the knapsack 0-1 problem have had processing elements compute column-by-column with each processing element computing its own row. In these systolic array implementations that compute along columns, the stalling time is linear on the item weight. In its worst case, one PU may be computing along a row with item weight = 1 while another PU may be computing along a row with item weight = 6144. This would infer a large delay on each iteration as the first PU stalls and waits for the second PU to finish computing. On the other hand, this MIMD processor version computes along rows rather than columns which means all PUs are executing the same item and thus same item weight on each iteration. For this version, the stall time is linear in the number of items in the knapsack to pick from. In the simple, 4 item version of the problem, the max stall time is on the last iteration where certain PUs stall for 24 clocks out of the 130 clocks taken to fill the last row. The details of this can be seen in the assembly code for the algorithm on each PU which can be found in appendix C.

The Smith-Waterman algorithm has as well been the subject of many research papers seeking to design the fastest systolic array processor architecture. In 2002, Steven A. Guccione and Eric Keller reported the Jbits processing element design capable of running up to 3225 GCUPS with 11,000 processing elements while requiring runtime reconfiguration. In 2003, C.W. Yu et al. proposed a systolic array for running the Smith-Waterman algorithm on a Virtex FPGA capable of doing 814 GCUPS with 4032 processing elements and no runtime reconfiguration requirement (Yu et al., 2003). Oliveira et al. present a systolic array for accelerating both the forward and back tracing parts of the SW algorithm with capabilities of up to 79.5 GCUPs (giga-CUPS) (Oliveira, Dias, and Fernandes, 2022). Additionally, SW has been adopted for running on GPUs. Barnes provides a comprehensive review of various GPU algorithms for SW with performances of up to 300 GCUPs on CUDASW++ 2.0 SIMT (Barnes, 2020).

On the MIMD processor, the Smith-Waterman algorithm programmed onto the PUs updated cells every 40 clock cycles. When implemented on a Spartan 6 FPGA, the processor with no multipliers was able to fit 54 PUs with a max global clock frequency of 35 MHz. This corresponds to 0.875 or approximately 1 MCUPS per PU. So for 50 PUs, a throughput of approximately 50 MCUPS can be achieved by the system. At first glance, this number appears low compared to the large CUPS for the systolic arrays above. However, there are numerous ways in which the CUPS for this system could be increased. On FPGAs, the design is heavily constrained by the amount of logic resources. The 50 PUs that fit on the Spartan 6 utilized approximately 40,000 slice LUTs. The latest NVIDIA GPU, however, the GeForce RTX 4090, contains 16,384 PUs so if the design were able to match this it could run at approximately 16.4 GCUPS. On the architecture side of things, if pipelining were added to the PU architecture, the clock speed would improve drastically. Additionally, if this MIMD system were to be implemented on a ASIC with state-of-the-art standard cells it is expected that this timing could be improved even more, perhaps up to 5-fold for each PU. All of this while maintaining the programmability that a systolic array lacks.

Digital Down Converters are used in software-defined radios to convert raw radio signals down to lower frequency and lower sample baseband signal for further processing. Most often, SDRs are implemented with specifically designed RTL hardware systems on FPGAs or custom ASICs to achieve low latency and high efficiency. Panoradio SDR is a website providing detailed explanations on SDRs

and digital down converters. Panoradio provides a full open-source SDR that uses SSB Weaver Demodulation and displays all signals from 0-100 MHz simultaneously (SDR, 2016). The DDC that Panoradio uses has a high decimation rate of 2048 and reduces bandwidth to 11 kHz and therefore is easily able fit on a 48 kHz sound card.

Implementing the DDC version of the MIMD processor, including 12 PUs and 10 multipliers, on the Spartan 6 FPGA consumes approximately 8,800 slice LUTS or 19% of the total available with a max clock frequency of 51 MHz. With this in mind, the DDC definitely has room to expand and utilize more logic elements. Alternatively, more SDR blocks could be added to the FPGA to further process output of the FIR filter such as a demodulator, mixer, or even more DDCs to further filter the data. The DDC algorithm had 193 clock cycles between consecutive outputs from the FIR filter. There was stalling of the PUs computing the CIC filter for the DDC since they compute quicker than the 120-tap FIR filter PUs. Due to this stalling, there is some margin for when the first stage integrator on CIC filter will need to sample in data. The first sample happens after only 17 clocks but the second sample occurs after 176 clock cycles as the CIC filter waits for the FIR filter to sync. If this were hooked up to an actual high-speed ADC with FIFO then the sampling time on the ADC would be (17+176) / 2 = 97 clocks. As the max clock frequency is 51 MHz, the maximum sampling frequency is thus (51 MHz/97 clocks) = 0.52 MHz or 520 kHz on the ADC. This is a step down from the Panoradio discussed above which can display signals up to 100 MHz, however, there are things that can be explored to improve the MIMD system.

3.2 Limitations

The knapsack 0-1 adaptation of this processor stored data through program memory. This means that the latency of this algorithm on the processor will depend on the number of items included in the problem. Additionally, since a part of this algorithm depends on reading data between processors a distance of w_j apart, the latency will also be determined by the values of the weights included in the design. That is, larger weights for the items means that the PUs will have to cycle for a longer time to share data. This will lead to adverse effects on the timing when scaling up this algorithm to include large numbers of items and/or having large weights for these respective items. This is also the main source of stalling in the processor while computing because certain PUs are able to skip this step of the algorithm and are thus left waiting. Methods should be explored to reduce or remove this effect.

The Digital Down Converter on the MIMD processor took a total of 193 clock cycles to run the CIC and FIR filter. Without stalling, the CIC filter only takes about 40 clock cycles with both samples occurring about 20 clocks apart. With stalling from the FIR filter, however, the first sample occurs after 17 clocks and the second after 173 clocks. Having PUs stall is sub-optimal for latency and timing purposes and possible remedies should be explored. Additionally, the parameters on the CIC filter as well as the number of taps on the FIR filter present some limitation on the level of filtering able to be done on the input data stream.

3.3 Future Work

General: The goal for this project was to analyze the potential of an MIMD type processor for accelerating various algorithms. The paper demonstrates a configurable MIMD for implementation on FPGA that can be tailored to these algorithms. In the future, certain portions of the MIMD processor could be more optimized to reduce excessive logic and sub-optimal timing. Also in the future, to explore these results further, each of the adaptations of this processor can be finalized and implementation on an ASIC with standard cells to improve timing and power performance. Certain alterations will need to made to accommodate implementation on an ASIC. In regard to the algorithms themselves, certain work can be explored in the future to remedy the limitations discussed above and add more functionality to the design.

Knapsack 0-1: With regard to the knapsack 0-1 problem, it would be interesting to explore incorporating data memory into the PUs for this problem. This would enable the system to avoid using program memory to store item weights and profits. If this is done, the size of program memory for the PUs would no longer scale linearly upwards as more items are included to choose from. It would also be interesting to explore computing simultaneously along columns rather than rows. The PUs could use the data memory to keep a backlog of past matrix values which would eliminate the wasted time used to repeatedly cycle data between PUs. To compute a matrix cell, a PU would be able to get all the data it needs from its neighboring PU rather than having to cycle through a variable amount of PUs depending on the weight as it is now. This would however introduce latency and stalling in other mechanisms as mentioned before. For the knapsack 0-1 problem and beyond, it would be interesting to consider the benefits of introducing distributed or shared memory, such as in Escobar et al., on this processor to improve the results. Additionally, future work could be done to adapt the MIMD processor architecture

here to incorporate other versions of the knapsack problem such as the bounded and unbounded versions.

Smith-Waterman: The Smith-Waterman algorithm tackled in this paper used a local linear gap cost. In the future, the MIMD processor and algorithm could be further adapted to support more complex affine gap penalties. Affine gap penalties involve a discrepancy between starting a gap in one of the two sequences and extending a gap. For example, starting a gap could involve a penalty of 3 but extending this gap further could mean a penalty of 1. Incorporating this would necessarily involve a mechanism in which PUs know if a gap is being opened or extended.

Digital Down Converter: The DDC algorithm and MIMD processor adaptation could be improved to reduce the stalling of certain PUs while computing. Three things could be done for this: (1) increase the filtering and thus compute time for the CIC filters, (2) split the FIR filters into even more parallel chunks to expedite their compute time, or (3) distribute the workload differently. For the first idea, it would be interesting to explore adding a greater degree of decimation or increasing the number of stages in the CIC filter. This would improve the output of the CIC filter and also increase the latency for the CIC filter PUs so they aren't stalling as much. An added benefit to this method is that adding more decimation would allow the system to decrease sampling time and increase the sampling frequency on the ADC. When no stalling is present, the CIC filter samples data every 17 clocks so if all stalling could be removed then the max sampling frequency would go up to (51 MHz / 17 clocks) = 3 MHz. More parallelization strategies could be explored to increase this max ADC sampling frequency even further. For the second idea, rather than running 10 PUs each doing 12 taps, it would be interesting to explore using 20 PUs each doing 6 taps, or perhaps 30 PUs each doing 4 taps, each of which could be easily accommodated given the space on the Spartan 6 FPGA. Adding more parallelism in this regard would decrease the latency of the FIR filter section of the MIMD processor so that the CIC filter section does not stall for too long or at all. This second idea would also decrease the overall latency of the DDC. For the third idea, the number of PUs for the CIC filter could be reduced from two down from to one. Putting the integrator, comb, and decimator all on one PU would increase the runtime of the CIC filter which would better match the runtime of the FIR filter and reduce stalling in the system. This would also potentially free up resources for the FIR filter.

BIBLIOGRAPHY

Barnes, Richard (2020). "A Review of the Smith-Waterman GPU Landscape". In: URL: https://api.semanticscholar.org/CorpusID:232108500.

Escobar, Fernando A. et al. (2017). "Scalable shared-memory architecture to solve the Knapsack 0/1 problem". In: *Microprocessors and Microsystems* 50, pp. 189–201. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2017.04.001.

Huang, En-Ming and Jie Chou (2022). "Optimization of multi-class 0/1 knapsack problem on GPUs by improving memory access efficiency". In: *Journal of Supercomputing* 78, pp. 13653–13679. DOI: 10.1007/s11227-022-04425-3. URL: https://doi.org/10.1007/s11227-022-04425-3.

Nibbelink, Kevin, Sanjay Rajopadhye, and Ross McConnell (2007). "0/1 Knapsack on Hardware: A Complete Solution". In: *2007 IEEE International Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 160–167. DOI: 10.1109/ASAP.2007.4429974.

Oliveira, Fernando F, Luiz A Dias, and Marcelo A C Fernandes (2022). "Proposal of Smith-Waterman algorithm on FPGA to accelerate the forward and backtracking steps". In: *PloS one* 17.6, e0254736. DOI: 10.1371/journal.pone.0254736. URL: https://doi.org/10.1371/journal.pone.0254736.

Rebacz, Jeff, Erdal Oruklu, and Jafar Saniie (2010). "Exploring scalability of FIR filter realizations on Graphics Processing Units". In: *2010 IEEE International Conference on Electro/Information Technology*, pp. 1–5. DOI: 10.1109/EIT.2010.5612114.

SDR, Panoradio (2016). *Dynamic Digital Correction for Zoomable Fast Fourier Transform*. Website. URL: https://panoradio-sdr.de/ddc-for-zoomable-fft/.

Yu, C. W. et al. (2003). "A Smith-Waterman Systolic Cell". In: *Field Programmable Logic and Application*. Ed. by Peter Y. K. Cheung and George A. Constantinides. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 375–384. ISBN: 978-3-540-45234-8.