Introduction

In computer architecture, one of the key considerations in designing new systems is optimizing for particular aspects. Specifically, computer architects optimize for properties such as area, performance, and energy. It is generally seen that increasing the quality of some of these properties leads to excess expense. For example, a more performant processor will be able to get more work done (in one interpretation), but will use up more energy and heat. This can thus limit scaling. In this work, we consider efficiency, which is the amount of work done per unit of energy.

The Problem/Opportunity

The MANIC [a19] design is a computer architecture to aims to optimize efficiency by using a vector dataflow. The goal of this work is to "...design a highly programmable architecture that hides microarchitectural complexity while eliminating the energy costs of programmability..." (MANIC). There are two very general ways that one could improve efficiency: either improve upon pre-existing computer architectures, or attempt to develop new ones. MANIC attempts to develop a new one, while at the same time hiding as much of the microarchitecture so that the programmer can work in a high level of abstraction.

Of note is that MANIC could see to benefit from a subsystem of fixed function accelerators. Specifically, this refers to a loosely coupled set of discrete modules that perform common, but specialized, operations in hardware. Some operations include Fast Fourier Transform, convolution, and sorting that are extremely important in digital signal processing, as well as form foundational functions in many algorithms.

The Approach

A large constraint on the design of these modules is the capabilities of the overall MANIC system. The MANIC system is not as performant, as it is efficient. Certain aspects of it lead to important design decisions. Loading and storing memory are expensive operations, and the overall system attempts to minimize that. This requires the exploration of
dataflows that minimize data movement by being able to reuse the data as much as possible.

**Related Work**

Large numbers of efficient designs have been created over the years. Many of these designs seek to operate on the fringe, in embedded devices where available power is very low or intermittently supplied. Clank [Hic17] introduces a new architecture for intermittent execution. Its application domain is on energy harvesting devices that build up energy and then use it before going back to sleep.

**Contributions**

We designed and implemented a new subsystem to the MANIC architecture that consists of a bus and arbiter, along with a core set of accelerators. These accelerators have passed correctness, and we seek to evaluate their performance later.

We have also implemented a variant of the Eyeriss architecture and the row stationary dataflow.

**Overall Design**

The design so far consists of sort and 2D convolution accelerators, as well as a bus. FFT is currently being worked on.

**Sort**

We implemented bitonic sort. The sort is parameterized to handle sorting arrays in sizes of powers of 2. Arrays that are not powers of 2 can still be sorted, but use the same exact data paths, and are just instead padded with dummy values (max value). This was the first accelerator that was implemented, and this was implemented fully combinatorially. While this was alright for our design, this could have been done differently by using sequential logic. The critical path for a fully combinational circuit is very long, and it
would have been good to split it up by having multiple registers and latches. In general, circuits with long combinational paths have longer critical paths, and are thus slower to operate because we need to clock slower in order to account for propagation delay.

The diagram below describes the general flow of bitonic sort. It has a $\log^2 n$ long critical path, with all of the wires able to perform the comparisons in parallel. The arrows indicate comparisons, with the arrow head pointing towards where the greater element would be swapped into.

**Bus**

The bus and arbiter were next worked on. This was necessary since we wanted all the accelerators to share a common set of data paths, and reuse the wires to get access to the core and as well as memory.

Since we would also have multiple accelerators attempting to use the same resources concurrently, we also implemented an arbiter to guard these accesses. This was simply
modeled as a round robin scheduler. For instance, when multiple accelerators wanted to access memory, the accelerators would be granted access for one time slice at a time (one clock cycle in our case) to handle their operations. This would repeatedly cycle through all outstanding requests in a fair order repeatedly.

This part of the system was developed generically to allow as many accelerators as possible. All that was needed for a new interface to interact with and use the bus was an adherence to a very basic interface.

**Convolution**

This was the most technically challenging accelerator to implement, and took the bulk of the time spent on this project. The dataflow was modelled after the Eyeriss Project [Che16].

In order to maintain efficiency in 2D convolution, it is very important to access data in ways to maximize reuse. A large amount of literature has been written about the topic in order to facilitate this reuse. Some of these strategies include weight stationary, output stationary, and row stationary [Che16].

For definitions, activations mean the (typically) large 2D input matrix we apply the convolutions, filters describes the weights that we repeatedly multiply the activations with, and the outputs/partial sums refer to the corresponding rows of the output that is a result of apply the filters to the activations.

We model the operation of convolution as a 2D matrix of Processing Elements (PE’s). These PE’s allow us to spatial represent the convolution operation. When we want to optimize for efficiency, we want to keep as much data as close to actual execution as possible. In this case, this means the PE’s. In all scenarios, we want to minimize the total number of accesses to main memory (which is very expensive), as well as any accesses to any sort of buffers.

When we consider a weight stationary dataflow, we keep the weights stationary in the PE’s, and repeatedly feed in activations until we have exhausted all of the operations which use that filter. The problem in this case is that it is impossible to accumulate the
partial sums on the PE’s, since the PE’s are unable to hold onto a very large amount of memory. We therefore are forced to flush the partial sums repeatedly to a buffer and accumulate there.

In the output stationary dataflow, the output for a particular row and column is kept stationary in a PE, while the activations and filters flow through. In this case, we are able to keep the partial sums completely in the PE until all of the sums necessary for an output are finished. In this case, we minimize the number of accesses to memory that are needed to accumulate the partial sums. The downside is that the activations and filters are forced to repeatedly be read from a buffer in order to calculate partial sums for other windows of the convolution.

Row stationary dataflow is what is implemented by Eyeriss, and is what we have modelled the dataflow as. In this case, the rows of the outputs are kept "stationary" in the PE array. This means the rows of partial sums are constantly being accumulated in order to compute a set of rows of outputs at a time. For a 2x2 PE array, we can keep 2 rows stationary at a time.

This can be seen in the diagram below:

![Figure 3: Row Stationary Dataflow](image)

A 1D convolution operation from the perspective of one PE is shown below, unfortunately the colors differ, but the meaning is defined:

Compared to our project, Eyeriss attempts a much more substantial and bandwidth
hungry implementation of the basic premise. For instance, our implementation consists of a 2x2 PE array, while theirs has been scaled up to around a 13x13 PE. This is massively larger, and more tuned towards raw performance.

In addition, their implementation utilizes a sophisticated routing and networking that is unnecessary for implementing the dataflow on a smaller scale.

We had also attempted to exploit some new data reuse possibilities in the way data flowed through the row stationary model that wasn’t as explored as in Eyeriss. In addition, bit packing was being worked on towards the end, although I have gone on to implement an FFT accelerator.

**Evaluation of Your Approach**

Evaluation of the approach has been mostly limited to correctness at this point. While I wanted to get some actual energy measurements and results from these accelerators, it is much tougher to get real numbers besides actually fabricating the chip. This is somewhat of an end stage that I hope to see happen. Although, it is possible to get a rough gauge of how well the design is doing by simulating the system. This is very variable, and sometimes can make or break a research paper in this field since the numbers can be wildly off.

With respect to correctness, multiple “fake” modules were implemented that simulate interactions from the core as well as inputs. These were repeatedly testing utilizing memory with the same characteristics as would be used, as well as adherence to handshakes and interactions between the accelerators, and the core and arbiter. So, in our case,
we’ve simulated the interactions of the accelerators both individually as well as operating concurrently.

A future goal is to get more concrete energy numbers.

**Surprises and Lessons Learned**

I think I find this area of computer science very interesting. This is definitely the lowest level I would go, and I have found it exciting to understand how things are actually implemented. In addition, I have found it valuable in order to think about algorithms in computer science in a very different way.

Grasping the syntax of SystemVerilog is not that tough of a task, although it does require considerable planning in comparison to something that I would implement in Python. This becomes apparent when refactoring or discovering a bug down the line. With SystemVerilog, it is much tougher to refactor, or attempt to quickly change the project. This is because it can require complete rewrites of hundreds of lines of code while in Python it might just be a couple lines.

One thing that I had expected, but had not fully grasped as I undertook this project is how laborous implementing algorithms in hardware is. I now fully understand why ”simply” moving things to FPGA’s and ASIC’s is not that straightforward. For the convolution operation, for instance, it took me approximately 3 months to implement a working model. Although, this was also under a full, overloaded course load, which definitely contributed to delays. I also understand why it is an incredibly expensive, although fully rewarding endeavor for mission critical systems.

I also think that this was a great way for me to get a foothold in research at CMU, and thought it was great to see meetings and the collaborations that happen between researchers in their attempts to get answers to questions.
Conclusions and Future Work

I hope to get a better understanding of the architecture in the future, and be able to work at a higher level with the theory. I think that I’ve accrued a decent amount of background knowledge in the field. While I hadn’t gone into this project completely with a lack of domain knowledge, the project has accelerated my learning in the field of computer architecture greatly.
References

