High Frequency Trading on FPGA

Kaustubh Dighe
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
kdighe@mit.edu

***Abstract*—In the high frequency trading (HFT) space, there has been an increasing drive to use FPGAs in the recent years. Reducing the latency of computation is paramount. When we have exhausted almost all avenues of optimization in a program written in a programming language like C, we look for other external avenues. When an ultra optimized C program runs on a regular computer, it still has to deal with the CPU being a general hardware that will first process instructions and then send them to GPU if needed. There will still be an operating system that will keep scheduling other processes along with our program and do context switching between the processes and the kernel. Also, communicating over the network takes up lot of time with general-purpose network cards which deal with lot of other packets coming to the system as well. It will also have to deal with complex memory heirarchy that slows processes down if they are not appropriately engineered to extract the best out of the caches. To solve all this, making use of an FPGA to run the same trading logic eliminates the operating system and also the hardware is specialized enough to be faster. More parallelization can be made into such FPGA as well. I propose and end-to-end trading system that communicates with ethernet and uses eigenvalue decomposition of the correlation of returns among a portfolio of stocks to make trading decisions.**

***Index Terms*—FPGA, HFT, Eigenvalues, Stocks**

I. Outline and Math

I am building a trading system where the FPGA gets new prices of all the stocks it is keeping track of periodically. Once it gets this data, the FPGA should, on the basis of prior information it has stored of those stocks, generate orders to place in order to earn profits. To do this, following is the outline of the steps to be taken.

- Step 1: The FPGA gets an array of new prices of all stocks. This is intended to happen over ethernet because we need network to get this data from a distant stock exchange.
- According to Portfolio Theory (Scott Rome, 2016), the eigenvalues and eigenvectors of the covariance of the stock returns are measures of risk and corresponding portfolios. Hence, we will evaluate the eigenvectors and pick one of them as our portfolio.
- Step 2: Find returns using

$$r_t = \frac{p_t - p_{t-1}}{p_{t-1}}$$

  when we get this new price data.
- Step 3: Have an incremental module which keeps updating covariance matrix for the returns as we keep getting new samples of returns. This matrix will in a way store everything we need to know about all the stocks' behavior till now.
- Step 4: Do the eigendecomposition of this covariance matrix
- Step 5: Find the second-highest eigenvalue (as the highest one corresponds to the highest risk one and most correlated with the market, so it will give high returns when market is growing but susceptible to the volatility of market) and pick the eigenvector corresponding to it. Scale it so that the sum of it's elements is one. This gives us the target portfolio.
- Step 6: Based on the current (and just previous) prices of stocks and target portfolio, place orders and evaluate cumulative returns.

II. Block Diagram

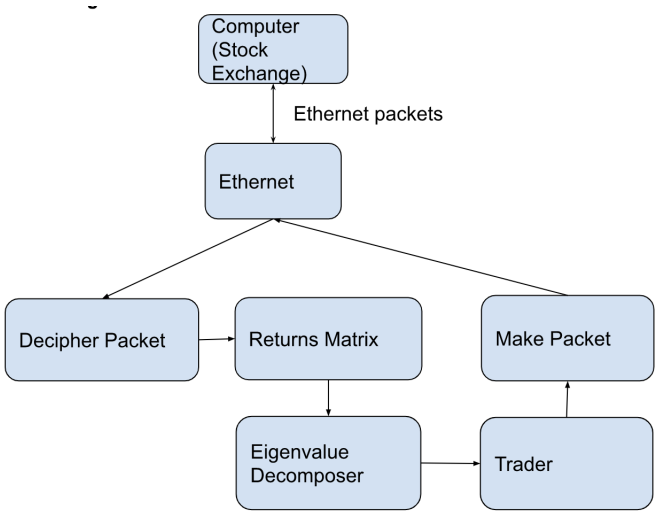

Fig. 1. Block Diagram

III. Components

A. Make Packets and Data

A Python script obtains data from API given by Rapid API to access the Yahoo financial data set of historical stock prices. This consists of initial and final prices of a given stock at intervals of 15 minutes. I also found data more easily from Python's pandas data reader library and I went ahead with that.

B. Ethernet

This module receives the packets sent by the computer over ethernet protocol. It corrects bit ordering, checks if there is any error using cyclic redundancy checking (CRC) and also determines if the packet was intended for the FPGA or not by looking at the MAC address. This module accumulates the entire data in the packet and sends it forward.

C. Deciphering Packet

This module deciphers what the packet says. Our packet protocol is very simple. Each packet contains 32 bits and each 16-bit portion of it is 16-bit updated stock price of the two stocks we are dealing with. This can be easily extended to have packets of 16N bits where N is the number of stocks. Pooling information like this makes the ethernet header overhead small in comparison to actual data and also takes into account the fact that multiple stocks may change at the same time. This module passes this information in arrays to next module.

D. Returns

For every stock at every packet arrival (we will refer to it as a timestep now onwards), we find the relative returns per share of that stock. This is evaluated as follows for timestep t :

$$r_t = \frac{p_t - p_{t-1}}{p_{t-1}}$$

Here, p_t is the price of that stock at timestep t. This step is done in the top level module itself. Once we get this array of returns, we evaluate the covariance matrix of the returns array.

E. Exponential Moving Covariance

Regular covariance and mean is found by taking all samples of a data together and weighing them equally. The subscript t refers to the values after t timesteps i.e t samples (denotes by $\mathbf{x} \in \mathbb{R}^{N_{STOCKS}}$) in this case.

$$\mu_\mathbf{t} = \frac{1}{t}\sum_{k=0}^{t} \mathbf{x_k} \in \mathbb{R}^{N_{STOCKS}}$$

$$\mathbf{Cov_t} = \frac{1}{t-1}\sum_{k=0}^{t} (\mathbf{x_k} - \mu_\mathbf{t})(\mathbf{x_k} - \mu_\mathbf{t})^\mathbf{T} \in \mathbb{R}^{N_{STOCKS} \times N_{STOCKS}}$$

This approach has several problems:

- We should give more importance to recent trends of covariance in the market. If we weigh all samples equally, after a point, new data samples will stop having any impact at all due to limited precision we have. Even ignoring precision issues and assuming infinite precision, the relative impact of new data samples would tend to zero. A moving average and covariance weighs all samples till now but the weight of recent samples is higher.
- We do not have all samples at once. We are getting new samples one by one. While we can certainly update covariance and mean as follows in this situation

$$\mu_\mathbf{t} = \frac{(t-1)\mu_\mathbf{t-1} + \mathbf{x_t}}{t}$$

$$\mathbf{Cov_t}^{ij} = \frac{t-2}{t-1}\mathbf{Cov_{t-1}}^{ij} + \frac{1}{t}(\mathbf{x_t^i} - \mu_\mathbf{t-1}^\mathbf{i})(\mathbf{x_t^j} - \mu_\mathbf{t-1}^\mathbf{j})$$

But there is unnecessary storage of state to keep track of t.

- Moreover, the above method involves having division. On FPGAs synthesizing division is extremely hard as either the divider takes up most of the area on the FPGA (it gets synthesized as a lot of complex look up tables) or if we make a sequential module, it eats up a lot of cycles. These cycles are either variable or equal to the number of bits of the divisor and dividend depending on implementation.

To solve all these issues, we have used exponentially moving average and covariance which is very easy to compute in programs and state machines as follows :

$$\mu_\mathbf{t} = (1-\lambda)\mu_\mathbf{t-1} + \lambda\mathbf{x_t}$$

$$\mathbf{Cov_t}^{ij} = (1-\lambda)\mathbf{Cov_{t-1}}^{ij} + \lambda(\mathbf{x_t^i} - \mu_\mathbf{t-1}^\mathbf{i})(\mathbf{x_t^j} - \mu_\mathbf{t-1}^\mathbf{j})$$

The parameter λ can be tuned as per requirement. We have kept it to be equal to 0.25 so that recent ones do not get weighed so highly that covariance loses meaning altogether. This is easy to compute because only additions and multiplications are involved and same register can be re-written to at each clock cycle in the state machine.

F. Eigenvalue decomposition

The next step is to evaluate the eigenvalues and corresponding eigenvectors of this covariance matrix. The diagonalization of the matrix is done by Jacobi rotation as this algorithm is easily parallelized on the FPGA and also makes a good state machine for checking convergence.

Algorithm 1 Jacobi rotation algorithm on matrix A

Require: A is symmetric (but that's fine because we are using it only for covariance matrices which are symmetric)

$D \leftarrow A$

$Q \leftarrow I$ ▷ Identity matrix

while Sum of off diagonal elements of D ¿ Threshold **do**

$i, j \leftarrow pivot(D)$ ▷ Max off diagonal element

▷ Jacobi rotation start

$\theta \leftarrow \frac{1}{2}\arctan\frac{2D_{ij}}{D_{jj} - Dii}$

$s \leftarrow \sin\theta$

$c \leftarrow \cos\theta$

▷ Update D

$D_{ii} \leftarrow c^2 D_{ii} - 2scD_{ij} + s^2 D_{jj}$

$D_{jj} \leftarrow s^2 D_{ii} + 2scD_{ij} + c^2 D_{jj}$

$D_{ii} \leftarrow (c^2 - s^2)D_{ij} + sc(D_{ii} - D_{jj})$

$D_{ik} = D_{ki} \leftarrow cD_{ik} - sDjk \forall k \neq i, j$

$D_{jk} = D_{kj} \leftarrow cD_{jk} + sDik \forall k \neq i, j$

▷ Update Q

$Q_{ki} \leftarrow cQ_{ki} - sQkj \forall k \neq i, j$

$Q_{kj} \leftarrow sQ_{ki} + cQkj \forall k \neq i, j$

end while At this point, D and Q are such that $A = QDQ^T$, D is a diagonal matrix with eigenvalues and columns of Q are corresponding eigenvectors.

In this algorithm, there is a use of arctan, sin and cos trigonometric and inverse trigonometric functions. This is handled in hardware by using CORDIC algorithm. This is an algorithm where instead of a single rotation (trigonometric functions are used in rotations and hence the math is interlinked) of arbitrary angles, repeated rotations of angles whose tangent functions are powers of two are used. In this way, repeatedly doing simple additions and shift operations, complex functions like trigonometric and others like logarithm and square root can be found. We have used Xilinx CORDIC IP for these modules.

G. Portfolio

A portfolio is the weight of all stocks we hold in terms of their money value. For example, if we have 3 shares of PQRS worth 30 and 2 shares of ABCD worth 5, the portfolio is 90% PQRS and 10% ABCD. Once we have eigenvalues and eigenvectors of the covariance matrix of returns, random matrix theories say that the eigenvalues are a measure of risk but also reward. Hence we can choose the second-highest eigenvalue and pick the eigenvector correspond to it. This eigenvector when scaled so that it has a sum 1, is the portfolio we want to have at that time step.

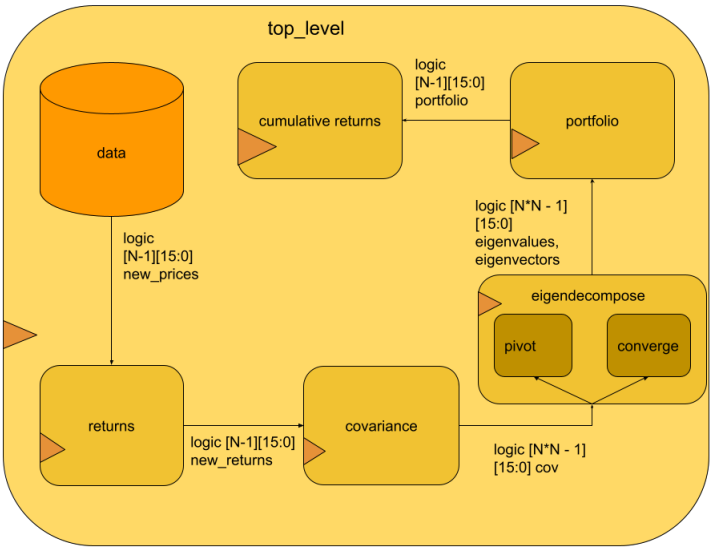

Fig. 2. System Diagram

H. Cumulative Returns

Cumulative returns are found as follows

- For each stock i,

$$CR_T^i = \prod_{t=1}^{T}(1 + r_t^i) - 1$$

- Money at time T

$$M_T = M_0 \sum_{i=1}^{N_{STOCKS}} (1 + CR_T^i) * w_T^i$$

IV. Design and Results

A. Dividers

I have striven to minimize the use of dividers due to their large footprint either on area or on time, both of which we do not want. But still, evaluating returns and scaling eigenvector to get portfolio required division. I used one from the web with appropriate license and modified it for signed fixed point numbers.

B. Fixed Point Numbers

Most math involved is not integers but involved real numbers. I have used fixed point math as opposed to floating point math as the range of values we want to deal with is not very wide but we do not want to focus too much on getting simple arithmetic correct.

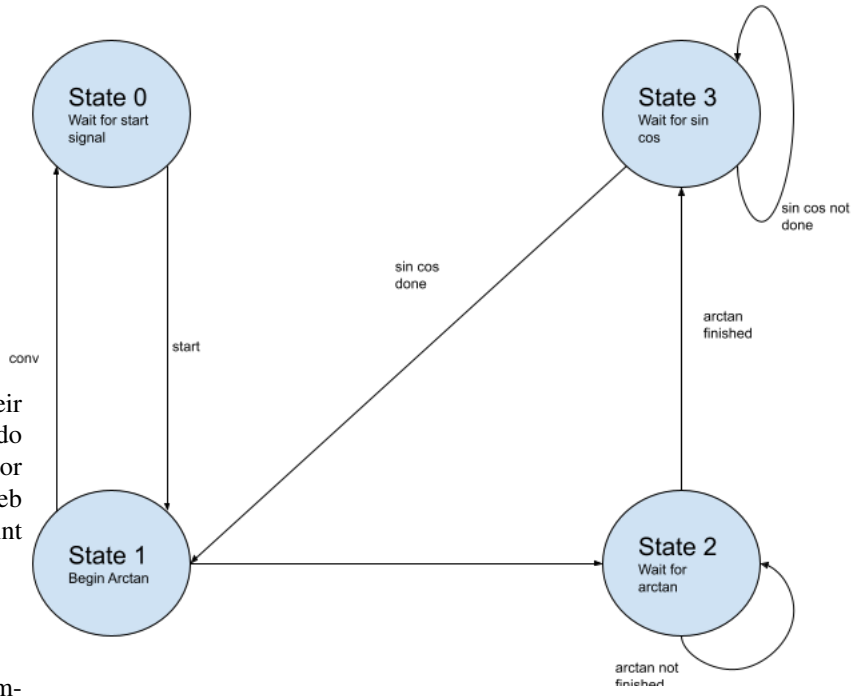

Fig. 3. FSM of eigendecompose

C. Latency

The main goal of going on an FPGA was latency. I have used a 100MHz clock and while the convergence of eigenvalue decomposition module has a variable cycle count because of we do not know when we converge but we can still count the number of cycles as follows :

- 1 cycle to get the data to returns dividers
- Each divider takes exactly 24 cycles to compute result because the algorithm for fixed point division takes number of cycles = total number of bits in number + fixed point width. So returns will be computed after 24 cycles.
- Covariance matrix gets updated in 1 clock cycle.
- Eigendecompose FSM is shown in figure 3. 2 cycles to go to state 2. The converge and pivot modules are combinational. I have used the CORDIC IPs in parallel configurations and for that the latency is N cycles for N bit input. I have 16 bit inputs so 16 cycles on state 3 and state 3 and then 1 cycle to return to state 1. This part of 34 cycles gets repeated unknown number of times but guaranteed to converge well.
- Portfolio module takes up a divider of 24 cycles and 1 cycle before it to find the 2nd highest eigenvalue.
- 1 clock cycle to find cumulative returns.

To sum it all, 53 + 34 * number of iterations of Jacobi cycles is the latency.

D. Goals accomplished

I have been able to get all modules described working separately. However, integration of the CORDIC module created problems. Here is a breakdown of commitment, goal and stretch:

- Commitment:
  - DONE: Decipher packets
  - DONE: Eigendecomposition of matrix
  - DONE: Find returns, covariance matrix and evaluate eigenvalues of that matrix
  - DONE: Trading logic
- Goal: PARTIALLY DONE: Integration with ethernet receive module to receive stock data from the computer.
- Stretch:
  - NOT DONE: Make ethernet transmit module to send orders back
  - NOT DONE: Make order books
  - NOT DONE: Have a Bayesian validator to keep checking the strategy

E. Future Modifications

- Having order books to tweak the price logic. Right now the prices are just taken from the market, but if we are trading large volumes we have the power to set prices and negotiate too.
- Ethernet transmit and integration with ethernet.

The code can be found at https://github.com/KaustubhDighe/Vyapaar

V. Challenges and Insights

- It is very annoying to deal with Verilog's quirks of multi-dimensional arrays being passed along modules. Unpacked arrays can get indexed in any manner but they disappear when they are signed and passed along various modules. Packed arrays connect well across modules but then they can only be indexed on the first dimension. I worked around this by having most of my matrices to be in row-major order to reduce extra dimensions.
- Testing each module separately is very helpful because bugs arise in such places
- As suggested by Fischer, having a reference implementation in a programming language helps flesh out all details of the math and also gives us reference values to check for at each step of the algorithms. I found this very helpful as I was getting stuck on writing the eigendecomposition on FPGA by just reading research papers. I then made most parts of the trading logic including eigendecomposition in C++ and integrated some parts as well. The way I made functions in C++ also gave me hints on what the different modules can be in Verilog design and what their inputs and outputs are.

VI. References

- https://srome.github.io/Eigenvesting-I-Linear-Algebra-Can-Help-You-Choose-Your-Stock-Portfolio/
- https://srome.github.io/Eigenvesting-III-Random-Matrix-Filtering-In-Finance/
- https://www.investopedia.com/articles/07/ewma.asp
- https://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm
- https://en.wikipedia.org/wiki/Covariance_matrix
- https://fanf2.user.srcf.net/hermes/doc/antiforgery/stats.pdf