High-Frequency Foreign Exchange Currency Trading (Forex HFT)

Spring 2016

Members: Graham Gobieski, Kevin Kwan, Ziyi Zhu, Shang Liu

UNIs: gsg2120, kjk2150, zz2374, sl3881

Table of Index

1. Background

High frequency trading is a trading platform that uses computer algorithms and powerful technology tools to perform a large number of trades at very high speeds. Initially, HFT firms operated on a time scale of seconds, but as technology has improved, so has the time required to execute a trade. Firms now compete at the milli- or even microsecond level. This has led to many firms turning to field programmable gate arrays (FPGAs) to achieve greater performance.

Our project focuses on triangular arbitrage opportunities on the foreign exchange market (Forex). The Forex market is a decentralized marketplace for trading currency. All trading is conducted over the counter via computer networks between traders around the world. Unlike the stock market, the Forex market is open 24 hours for most of the week.

Currencies are priced in relation to each other and quoted in pairs that look like this: `EUR/USD 1.1837`. The currency on the left is the base currency and the one on the right is called the cross currency or quote. The base currency is always assumed to be one unit, and the quoted price is what the base currency is equal to in the other currency. In this example, `1 Euro = 1.1837 USD.`

Triangular arbitrage takes advantage of pricing inequalities across three or more different currencies to make a profit. In a three currency situation, one currency is exchanged for a second, the second for a third currency, and finally the third back to the original currency. For example, if the exchange rates for the following currency pairs were `EUR/USD 1.1837`, `EUR/GBP 0.7231`, and `GBP/USD 1.6388` a trader could use 11,847 USD to buy 10,000 Euros. Those Euros could be sold for 7231 British Pounds, which could then be sold for 11,850 USD, netting a profit of 13 USD. Unfortunately, acting on these price inefficiencies quickly corrects them, meaning

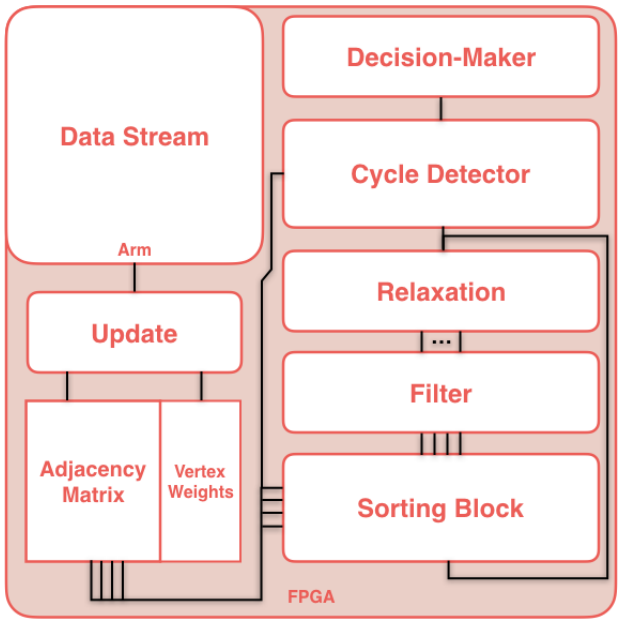
traders must be ready to act immediately when an arbitrage opportunity occurs.

Our group hopes to implement a Forex arbitrage calculator on an FPGA using a parallelized Bellman-Ford algorithm. We believe this will be fast enough to detect and act on arbitrage opportunities in near real time.

---

2. Design Overview

Figure 2.1: Design Overview



2.1 Identification of Arbitrage

Triangular arbitrage opportunities arise when a cycle is determined such that the edge weights satisfy the following expression:

$$w_1 * w_2 * w_3 * \ldots * w_n > 1$$

However, cycles that adhere to the above requirement are particulary difficult to find in graphs. Instead we must transform the edge weights of the graph so that standard graph algorithms can be used. First we take the logarithm of both sides, such that:

$$\log(w_1) + \log(w_2) + \log(w_3) + \ldots + \log(w_n) > 0$$

If instead we take the negative log, this results in a sign flip:

$$\log(w_1) + \log(w_2) + \log(w_3) + \ldots + \log(w_n) < 0$$

Thus, if we look for negative weight cycles using the logarithm of the edge weights, we will find cycles that satisfy the requirements outlined above. Luckily, the Bellman-Ford algorithm is a standard graph algorithm that can be used to easily detect negative weight cycles in O(VE) time. Please see Algorithm 5.1 for further discussion.

2.2 Pipeline

Generally speaking, in order to identify arbitrage opportunities we process raw Forex time-series data in four main steps (please see Figure 2.1 for a visual representation of that explained below).

1. First on the ARM CPU we process a CSV file of historical data, take negative logarithm of the closing rates, and round result the nearest integer.
2. Then we stream the data over the AMBA bus to an update module which is responsible for updating the internal representation of the graph. We choose to represent that graph in adjacency matrix format as a two-dimensional vector of integers. In addition there is also a one dimensional vector of integers that represents the weights of each vertex.
3. After the graph has been updated, the modules associated with the Bellman-Ford algorithm are started. First each edge is

processed in the comparator and relaxation modules. Once all edges have been processed the shortest path from a source to each vertex is known. Additionally, it is a trivial task forthe cycle detector module to loop through all of the edges again and check for a negative weight cycle.

4. If a negative weight cycle is found, the decision-making module will display the cycle on screen and show trades required to take advantage of the arbitrage opportunity.

---

3. Data and Communication

3.1 Raw Data

We will use historical exchange-rate time-series data downloaded from the internet. The website HistData provides the data for sixty-six Forex pairs broken into yearly chunks. The pairs include:

Figure 3.1 Forex Pairs

EUR/USD (2000/May)	EUR/CHF (2002/March)	EUR/GBP (2002/March)	EUR/JPY (2002/March)
EUR/AUD (2002/August)	USD/CAD (2000/June)	USD/CHF (2000/May)	USD/JPY (2000/May)
USD/MXN (2010/November)	GBP/CHF (2002/August)	GBP/JPY (2002/May)	GBP/USD (2000/May)
AUD/JPY (2002/August)	AUD/USD (2000/June)	CHF/JPY (2002/August)	NZD/JPY (2006/September)
NZD/USD (2005/August)	XAU/USD (2009/March)	EUR/CAD (2007/March)	AUD/CAD (2007/July)
CAD/JPY (2007/March)	EUR/NZD (2008/March)	GRX/EUR (2010/November)	NZD/CAD (2008/March)

The data arrives as a CSV file with the following format. We will use the closing rate in our simulations.

Figure 3.2 Sample Forex Data

Date	Time	Open	High	Low	Close	Volume
2016.02.01	00:00	1.08481	1.08494	1.08481	1.08494	0
2016.02.01	00:01	1.08492	1.08492	1.08471	1.08471	0
2016.02.01	00:02	1.08471	1.08476	1.0846	1.08474	0
2016.02.01	00:03	1.08474	1.08474	1.08468	1.08472	0
2016.02.01	00:04	1.08471	1.08472	1.0847	1.08471	0
2016.02.01	00:05	1.08473	1.08477	1.08473	1.08476	0
2016.02.01	00:06	1.08477	1.08477	1.08469	1.08472	0
2016.02.01	00:07	1.08473	1.08477	1.08473	1.08477	0
2016.02.01	00:08	1.08478	1.08484	1.08477	1.08481	0

3.2 Preprocesing and Floating-Point Approximation

Unfortunately, floating-point arithmetic on the FPGA is not a trivial task and may require a custom implementation of various operations. As a result, we have decided to utilize the floating-point arithmetic resources of the AMD chip onboard the FPGA. As such, we propose a two-step process that manipulates the data in such a way where only integers are streamed to the FPGA. This preprocessing operation is described below:

3.2.1 Logarithm

As part of the algorithm to detect arbitrage the logarithm of rate is required so that negative-weight cycles are possible (please see seciont 5 for more discussion on the algorithm). We will use the logarithm mechanism on the AMD chip to calculate the logarithm of each rate.

3.2.2 Rounding

Once the logarithm has been taken we will convert the resulting floating point to an integer by multiplying by a sufficiently large factor of 10 (the greater the factor the higher the precision) and then throw-away the remaining decimal. In this way we will be left with large integers that can be streamed and operated on in the FPGA efficiently.

3.3 Communication

After preprocessing we will stream the data via the Amba bus to the FPGA using a custom memory-mappied I/O device driver like we did Lab 3. The integers that we will stream will be fixed at 16 bits, and we hope to stream as fast as possible with minimum delay so that we can effectively simulate reality. Initially, the data will be wholly historical, but if time permits we hope to stream live data to the FPGA.

More specifically, we plan to interface with the internet and the CSV files with Python and then call C functions that will communicate (via memory-mapped I/O operations) to the FPGA.

---

4. Graph Storage

There are two standard ways to store a graph: an adjacency list and and an adjaceny matrix. We choose the second format because it is easier represented on the FPGA using a large two-dimensional vector. In addition the Bellman-Ford algorithm is just as capable at processing an adjacency matrix as it is an adjacency list.

Specifically speaking, there are several pieces of information that must be stored. The weights of edges between vertex *i* and vertex *j* denoted w(i, j) will be stored in the adjacency matrix. Additionally the predecessor, denoted p(i), and the weight, denoted w(i), must be stored for each vertex. These values will be stored in a large one-dimensional vector in which the index will correspond to the vertex.

Using rough calculations we estimate the total memory usage in the following way:

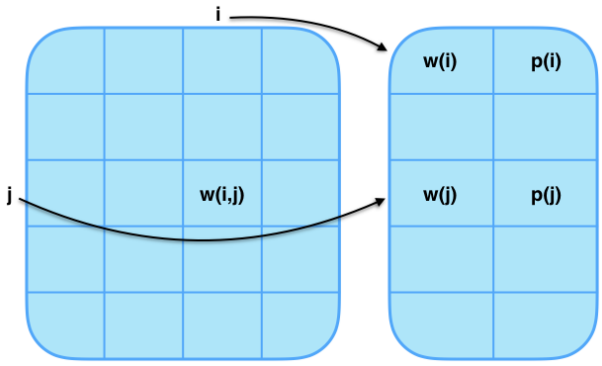
V = 66 nodes

Edges = 66^2 = 4356 edges

Total Bits = (66 nodes) * 2 * (7 bits/node index) + (4356 edges) * (10 bits/weight of edge)

+ (66 nodes) * [(10 bits/weight of vertex) + (7 bits/predecessor of vertex)] = 45606 bits ~ 5.7kb

Figure 4.1 Graph Storage



---

5. Bellman-Ford Algorithm

5.1 Bellman-Ford Algorithm

Algorithm 5.1: Standard Bellman-Ford

- Let *G(V, E)* be a graph with vertices, *V*, and edges, *E*.
- Let *w(x)* denote the weight of vertex *x*.
- Let *w(i, j)* denote the weight of the edge from source vertex *i* to destination vertex *j*.
- Let *p(j)* denote the predecessor of vertex *j*.

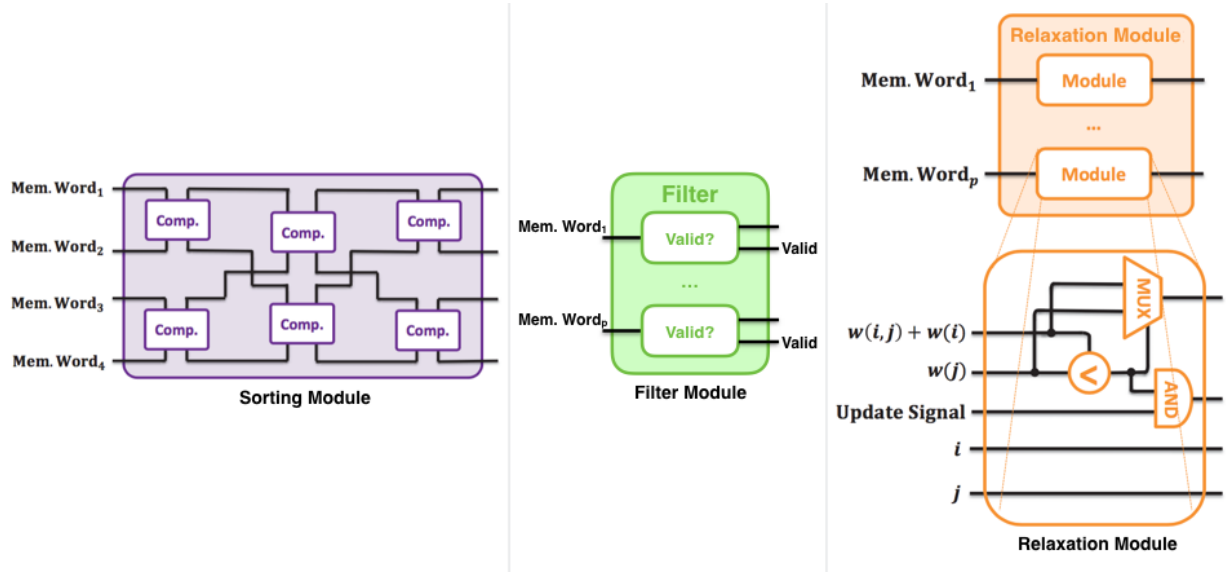
```
for each vertex x in V do
    if x is source then
        w(x) = 0
    else
        w(x) = INFINITY
        p(x) = NULL
    end if
end for

for i = 1 to v - 1 do
    for each edge(i, j) in E do
        if w(i) + w(i, j) < w(j) then //Relaxation
            w(j) = w(i) + w(i, j)
            p(j) = i
        end if
    end for
end for

for each edge(i, j) in E do
    if w(j) > w(i) + w(i, j) then
        //Found Negative-Weight Cycle
    end if
end for
```

The Bellman-Ford algorithm is a standard graph algorithm that seeks to solve the single-source shortest path problem. Mainly this problem describes the situation in which a source node is selected and the shortest paths to every other node in the graph need to be determined. In unit graphs, breath first search may be used, but in graphs that have non-unit edge weights the Bellman-Ford algorithm must be used.

Briefly, in the Bellman-Ford algorithm "each vertex maintains the weight of the shortest path from the source vertex to itself and the vertex which precedes it in the shortest path. In each iteration, all edges are relaxed [w(i) + w(i, j) < w(j)] and the weight of each vertex is updated if necessary. After the ith iteration, the algorithm finds all shorest paths consisting of at most i edges." After all shortest paths have been identified, the algorithm loops through all of the edges and looks for edges that can further decrease the value of the shortest path. If this case then a negative weight cycle has been found since a path can have at most v-1 edges. Proof of correctness can be found in *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein.

5.2 Bellman-Ford on FPGA

Figure 5.1: Bellman-Ford on FPGA



The implementation of the Bellman-Ford algorithm on FPGA that we propose is wholly based on the work in "Accelerating Large-Scale Single-Source Shortest Path on FPGA." However, instead of using off-chip DRAM as was done in the paper, we will simplify the algorithm by using onboard SRAM.

The implementation is composed of three main modules, two of which (sorting module and the relaxation module) are taken from the text. Together the modules can process up to four edges at once, although this can be extended to further parallel implementations.

5.2.1 Sort Module

The sorting module implements a static version of biotonic sort to determine which of the four edges loaded are valid candidates for relaxation. A high-order bit is reserved as an update signal bit that is toggled on and off by the sorting module to signify which edges are valid candidates. The module determines the validity of each edge by using a comparator module with pseudocode given in Algorithm 5.2. The comparator module compares two edges at time making a distinction between two cases: if the edges have the same destination vertex then the edge with the least weight and coming from the least-weighted vertex is considered the valid candidate; otherwise if the edges do not have the same destination vertex, both edges are considered valid unless one or both have low update signals.

Algorithm 5.2: Comparator

- Let e_k denote the edge (k = 0, 1)
- Let j_k denote the destination vertex of the edge (k = 0, 1)
- Let w_k denote the update value (w(i)+w(i,j)) of the edge (k = 0, 1)
- Let u_k denote the update signal of the edge (k = 0, 1)

```
if u_0 = u_1 = 1 then
    if j_0 = j_1 then //Same destination vertex
        if w_0 < w_1 then
            return e_0
        else
            return e_1
        end if
    else
        if j_0 < j_1 then
            return e_0
        else
            return e_1
        end if
    end if
else if u_0 != u_1 then
    if u_0 = 1 then
        return e_0
    else
        return e_1
    end if
else
    return e_0
```

5.2.2 Filter Module

The filter module looks at the update signals of each of the four edges in consideration and determines how many relaxation modules need to be utilized to determine which paths needs to be updated. A high update signal means that the edge is a valdi candidate, a low update signal means that the edge is no longer a candidate. Note that each destination vertex will have only one valid edge candidate at the end of this process, such that updating the adjacency matrix becomes an atomic operation.

5.2.3 Relaxation Module

This module compares the weight calculated from the valid edge candidate path to the current weight of the destination vertex. If the weight is less then the weight of the vertex in the list stored of vertex weights, the value is updated, otherwise nothing occurs and algorithm continues.

5.3 Cycle Detection

Once all edges have been processed (in (Total Edges/Number of Edges Processed in Parallel) cycles), the cycle detector module can take over and run a loop in parallel that looks for edges that decrease the weight of shortest paths calculated. Once an edge of this sort is found it can be passed to the decision-making module.

---

6. Decision-Making

When a negative cycle is detected by the Bellman-Ford algorithm outlined in the previous sections, we will trace back through the path of the cycles, determine the order of trades required and send a signal to execute these trades. While we could include our trades in further simulation we have decided to focus on cycle detection and as such we do not plan, at the moment to have these tradfe effect future rates.

---

7. Milestones

7.1 Milestone 1 (March 31st)

- Python interface for CSV files containing historic Forex data
- Static storage of graph on FPGA
- Comparator Module
- Relaxation Module

7.2 Milestone 2 (April 12th)

- Tested Bellman-Ford algorithm
- Kernel module that implements memory-mapped I/O ioctl call
- Start VGA-display support

7.3 Milestone 3 (April 26th)

- All components of implementation talking to each other (i.e. python frontend => C frontent => custom kernel module => FPGA => update module => comparator module => relaxation module => decision module)
- Full VGA-display support so we can visualize the cycles
- Realtime data streamed from the internet

---

8. References

1. Fundamental-reading about high frequency trading https://en.wikipedia.org/wiki/High-frequency_trading
2. Discussion of different types of arbitrage https://en.wikipedia.org/wiki/Arbitrage
3. Bellman-Ford implementation on FPGA. This is the work that we base our implementation upon. Accelerating Large-Scale Single-Source Shortest Path on FPGA
4. StackOverflow discussion that explains some of the theory behind calculating triangle arbitrage http://stackoverflow.com/questions/2282427/interesting-problem-currency-arbitrage

---