Quantum Compiler Optimizations

by Jeff Booth
boothjmx@cs.washington.edu / boothjm@uw.edu

A senior thesis submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science with Departmental Honors
Computer Science and Engineering - University of Washington

Abstract

A quantum computer consists of a set of quantum bits upon which operations
called gates are applied to perform computations. In order to perform quantum
algorithms, physicists would like to design arbitrary gates to apply to quantum
bits. However, the physical limitations of the quantum computing device restrict
the set of gates that physicists are able to apply. Thus, they must compose a
sequence of gates from the permitted gate set, which approximates the gate they
wish to apply.

Austin Fowler proposes a method that finds optimal gate sequences in
exponential time, but which is tractable for common problems [1]. In this paper,
I present several optimizations to this algorithm. While my optimizations do not
improve its overall exponential behavior, they improve its performance by one to
two orders of magnitude.

Presented on Friday, March 16th, 2012

Thesis and Presentation Approved By:

_______________________________________________________________

Date Approved

_______________________________________________________________
Table of Contents

Background
Fowler’s Algorithm
Experimental Goal
  Code Profiling
    Calculation Time vs. Fowler Distance
    Time vs. Sequence Length
    Unique Sequences Per Sequence Length
Ways to Improve Performance
  “Meet in the Middle” Bidirectional Search
  Optimized Unique Matrix Lookup
  FPGA Parallelism and Pipelining
FPGA Implementation
  Numeric Representation
  Algorithm Space Requirements
  Measurements
    The CPU;
    The Board;
  Error Propagation
“Meet in the Middle” Bidirectional Search
  The Middle Structure
  Performing the Search
  Results
Future Work
  Using k-d trees for the bidirectional search middle structure
  Map-Reduce Parallelism
  FPGA Tree Memory
Summary
References
Appendix A: Unoptimized Code Profile
Background

In classical computing, we can generally rely on the correctness of hardware because of the size of the circuit components. For example, if an atom on a hard disk drive changed its spin orientation, or lost an electron, the hard drive’s functionality would not be impaired because it takes many thousands of atoms to represent and store a single bit of data. However, in quantum computing, data is stored in quantum bits, which are represented by tiny particles like trapped ions. These bits are very easy to perturb, potentially corrupting calculations based on them. Thus, we use redundancy, in the form of error-correcting codes, to minimize the impact of individual errors.

The Steane code is one representation of a quantum bit. It uses seven physical qubits to represent one Steane code qubit, and can tolerate an arbitrary error in one of the seven qubits. We can perform any desired operation on a Steane code qubit by applying a combination of H (Hadamard) and T gate operations [2]. T gates are complicated to implement, so we seek to avoid them. For practical purposes, we can use not just the H gate, but the gates X, Z, S, and $S^\dagger$. The gates $H$, $X$, $Z$, $S$, and $S^\dagger$ form a group under multiplication, called the Clifford group. Thus, any sequence of gates we choose will alternate between a member of the Clifford group and a T gate.

A quantum gate compiler finds sequences of gates which yield matrices that are “close” to a gate we would like to apply to a quantum bit. Each gate has a corresponding matrix that represents the operation it would perform on a quantum bit. How “close” one gate is to another is given by the Fowler distance[1]:

$$ dist(U, U_1) = \sqrt{\frac{2-|tr(U^\dagger U_1)|}{2}} $$

The longer the gate sequence is, the more closely it can approximate a desired target gate. However, a longer gate takes more time to compute on a real quantum computer, increasing the probability of a computation error. An optimal quantum compiler will find gate sequences which:

1. have a minimal Fowler distance from the target gate.
2. have a minimal length, to reduce the probability of computation errors.

Fowler’s Algorithm

Austin Fowler presents an algorithm that iterates over sequences in order from smallest to largest [1]. For each sequence, it multiplies the sequence gates together to generate a unique matrix representing the complete operation that sequence would perform. The simple brute-force iteration runs in exponential time for a given number of gates, since for each sequence item, the item preceding it iterates through all of gates in the gate list.

To reduce the run time, Fowler’s algorithm intelligently skips redundant sequences. The
algorithm creates a list of unique sequences for all sequences of length N. Then, for each
sequence S of length N+1, it searches for sub-sequences of length N. If a sub-sequence T is
not in the list, then it is not unique. That means it performs the same operation as a sequence V
which is in the unique sequence list. Since T and V are the same, then if you were to replace T
with V in your sequence S, you would get a sequence W that does the same thing S does.

Since the algorithm iterates over all sequences of length N, it will encounter W anyway.
Therefore, it should skip sequence S. In fact, it should increment the sub-sequence T until it is
a unique sequence U. Fowler’s algorithm contains a tree lookup structure which, for any given
sequence, records the next unique sequence U. The algorithm can determine what sequence
to skip to by simply accessing this tree. While this program stills run in exponential time, this
skipping optimization reduces the time to the point where interesting results can be obtained.

As I will demonstrate later, the tree data structure requires memory that scales exponentially
with the sequence length. Thus, the algorithm consists of two stages. During the first stage, it
builds the structure until it stores all unique sequences up to length W, where W = 15 for most
of the experiments. After the structure is built, it enters the second stage, where it generates
sequences -- and uses the structure to skip them -- but it doesn’t add them to the structure.
This dramatic change in behavior between stages explains some interesting features in the
following graphs, and it means I can only infer behavior on longer sequences from behavior in
the second stage.

**Experimental Goal**

In order to empirically measure the impact of my optimizations, I need a consistent experimental
goal to test on every version of the algorithm. For this research, I chose to approximate the
\[ \frac{\pi}{6} \]
gate to 10^{-7} accuracy. Since I know from prior knowledge that this approximation is currently
very time-consuming, I can use it to see if my enhancements have any meaningful impact.

**Existing Performance**

In order to better understand the performance characteristics of the Fowler algorithm, I modified
its C source code to obtain performance-related statistics. In this section, I present the data
I gathered, along with some explanations for unusual data features and speculations on
how the statistics should change for a meaningful performance improvement. Most of these
benchmarks ran on an Amazon Elastic Compute Cloud Medium computer, which contains a 2-
2.4 GHz processor and 3.75 GB of memory.

**Code Profiling**

I ran a profiler (gprof) to determine where the performance bottlenecks are. The resulting table
is in Appendix A. Initially, I thought that memory accesses would dominate the program’s
runtime, because of the size of the data structures involved. However, the program spends 92.49% of its time inside mathematical functions, meaning that calculation is the dominant operation.

- In the first stage of the algorithm, the program spends 98.5% of its time checking for unique matrices.
- In the second stage, it spends most of its time multiplying gate matrices together to calculate the matrix for a given sequence.

**Calculation Time vs. Fowler Distance**

The figure below indicates how much time will be required to obtain a given Fowler distance using Fowler’s original source code. For the purposes of this paper, the Fowler source code is “unoptimized”, as it does not contain my optimizations and is a baseline for comparison. This graph is perhaps the most important graph of them all, since we often want gates with a certain specific precision.

![Calculation Time vs. Fowler Distance](image)

Since there aren’t very many unique distances, there are not enough data points to establish a clear trend. A power function appears to fit the data somewhat closely, though. This power function predicts that the unoptimized version of the program will take about 110 years to approximate the gate to a distance closer than $10^{-7}$! This massive exponential expansion explains why Fowler’s original paper had no gates with a precision better than $10^{-4}$, since it would take at least a day for the $6$ gate to compile to even that precision!

**Time vs. Sequence Length**

This metric is related to the above metric because longer sequences tend to have better precision. However, the relationship between time and sequence length is much clearer, as can be witnessed by the much smoother curve. While this graph may not have as much practical significance, it is much easier to relate this graph to the underlying implementation of the
algorithm.

From sequence length 0 to 2, the line has a steep slope. This feature probably exists because the processor cache has not warmed up yet. Between 2 and 15, every sequence generated by the algorithm is checked against a list of unique sequences, to see if it’s unique. This check only occurs up to a certain sequence length: 15 in this case. After that, the algorithm speeds up very rapidly until it reaches about a sequence length of 30. Then, the graph becomes a clean exponential curve.

To improve performance, I will effectively need to shift this curve down, producing longer sequences in less time.

**Unique Sequences Per Sequence Length**

This metric provides insight into the algorithm’s storage requirements. It is clear that Fowler’s optimizations have not altered the fundamental exponential nature of the problem. For sequences longer than about 3, the number of unique sequences grows exponentially with the sequence length. Since I am more worried about time rather than space, I will not mind if this curve shifts up. However, I do need to make sure that my optimizations do not consume too much memory.

**Ways to Improve Performance**

To optimize the performance, I need to:

1. Speed up calculations such as matrix multiplication.
2. Reduce the number of calculations required for a given gate sequence length.

There are quite a few possible approaches to approaches 1 and 2. Some of these approaches were taken this quarter, yet others will be left for future work.

**“Meet in the Middle” Bidirectional Search**

A traditional “uni-directional” search seeks a path from a start state to a goal state by starting
from the start state and exploring all possible paths. A bidirectional search starts searching from the goal state as well. Thus, the search paths will “meet in the middle”: each search only has to take N/2 steps to meet the other search. Thus, instead of taking \( O(a^N) \) time, the algorithm only takes \( O(a^{N/2}) \) time. One will need some data structure to store the paths, but inserting into this data structure does not require exponential time. Thus, for a given amount of time, the algorithm could compute gate sequences that are twice as long. This approach is the most promising, and it was implemented in software.

**Optimized Unique Matrix Lookup**

The algorithm checks to see if a matrix is unique by calculating the distance between it and all other matrices. Since 98.5% of the application’s run time is spent in this function, optimizing it could yield significant improvements in performance in the first stage. However, in the second stage, no more unique matrix checks are performed; therefore, no time will be spent in this function. Unless the first stage lasts a long time, it may not be worth the implementation trouble. This optimization was easy to implement since the C++ standard template library provides a binary search tree.

**FPGA Parallelism and Pipelining**

FPGAs can speed up calculations significantly by exploiting parallelism and pipelining. Since calculation dominates the algorithm’s computation, one can clearly benefit by using as many multiplier units as possible. If one has enough multipliers on an FPGA, one can obtain an entire sequence every few clock cycles.

I attempted to test FPGA acceleration this quarter, but due to a number of unfortunate circumstances, I was unable to complete a full implementation in time. However, I will summarize my implementation efforts and estimated performance gains.

**FPGA Implementation**

This quarter, I began implementing a very simple FPGA quantum compiler. This compiler consists of several interacting hardware modules that implement a brute-force search over the space of all possible gate sequences:

1. The **sequencer** generates sequences of gates in increasing order by length.
2. The **sequence multiplier** calculates the product of the gate matrices, for the gates in the sequence.
3. The **solution checker** determines how close the sequence matrix is to the solution matrix.
4. The **coordinator** starts the process, and exchanges data between the computer and the other modules.
In the above diagram, I represent a gate by a decimal digit, so sequences of gates are represented by digits. Note that there are actually 25 gates, not 10. Aside from that difference, decimal numbers make an excellent analogy to the real operation of the sequencer. Once the sequencer has cycled through all the possible gates for the last position in the sequence, it increments the next-to-last position and starts over.

**Numeric Representation**

One special property of the matrices I’m working with in this project is that their entries do not significantly change their magnitude. Thus, I can use a fixed-point format to represent them, avoiding the complexity and calculation overhead of floating point numbers. The fixed-point format I chose represents numbers using 37 bits:

My complex number multiplier module strips off the sign bit, and gives the rest to a set of four 18-bit multipliers. Once the multipliers compute a result, the sign bit is added back. By not passing the sign bit to the multipliers, I obtain a whole extra bit of precision!
Algorithm Space Requirements
The source code for Fowler’s algorithm includes an example that calculates and stores all unique sequences of length 15. This example requires storage for 10,200 unique sequence entries. In order for the FPGA design to run the provided example code, it must be able to store the following data:

- **Sequence Product**: 100 entries (max # of gates) * 5 bits/gate = 500 bits
  The sequence product stores the elements in the sequence currently being examined. Since there are 25 gates for each entry in a sequence, 5 bits are required per entry.
- **Unique Product Lists**: 15 max sequence length * 10,200 entries * 5 bits/gate = 765 Kbits
  The gates in unique products are stored in this list. The Product Check Tree contains pointers to this list, to tell the sequence skipping algorithm which sequences to skip to.
- **Unique 2x2 Matrices**: 10,200 matrices * 37 bits/number * 8 numbers = 3.0192 Mbits
- **Product Check Tree**: 10,200 blocks * (# of gates + 2) = 265.2 Kbits

Technically, only the Sequence Product is necessary unless Fowler’s skipping optimization has been implemented. Implementing the sequencer design on a Terasic DE-1 prototyping board is still feasible, though, even with the optimization.

Measurements
To see if an FPGA would offer a performance boost over a general-purpose CPU, I intended to compare their matrix multiplication speed and sequence enumeration speed. I managed to do the former, but not the latter.

The CPU:
I ran a single-threaded benchmark with no vector (SIMD) instructions on a computer with an Intel Xeon CPU 2.8 GHz processor. The maximum number of matrix multiplies per second was about 2,952,962.

The Board:
I designed a complex matrix multiplier for the Terasic DE-1 prototyping board. This board contains an Altera CycloneII FPGA with 26 hardware multipliers. Based on my design, the maximum number of matrix multiplies/second would be 6,250,000 at 50 MHz. The matrix multiplier was actually downloaded onto the board, so this component is fully functional. The CPU could probably match this performance with Single-Instruction Multiple Data instructions, which can operate on two double-precision floating point numbers at once. However, I am pretty confident that I can almost double the clock speed on the FPGA, yielding 12,500,000 matrix multiplies per second. If I am lucky, I can run the circuit at 150 MHz, yielding 18,750,000 multiplies, but there may be some serious timing issues.

Error Propagation
The FPGA’s hardware multipliers have less precision than the computer’s floating point unit. While an Intel processor’s floating point unit uses 80 bits per number, the FPGA has only 18-bit multipliers. Thus, a round-off error in the least significant bit changes the result by $3 \times 10^{-6}$. For large sequences, these errors could potentially accumulate, resulting in incorrect calculations. To address this concern, I combined multiple 18-bit multipliers in my design to create a larger 36-bit multiplier. This multiplier has higher latency, but can operate without pipelining at 50 MHz. With this precision, a round-off error would change the result by $3 \times 10^{-11}$, which should allow large errors to accumulate before adversely affecting precision.

“Meet in the Middle” Bidirectional Search

Searching for the correct gate is like searching through nodes in a tree: for a given sequence of gates, the computer must choose which gate to add to the sequence to come closer to the target gate. In the diagrams below, the arrows represent a choice of gate, and the boxes represent matrices. When an arrow is drawn from some box A to a box B, box B is the matrix resulting from multiplying A by some gate matrix.

In the example shown in these figures, the existing code must go through five levels of searching in order to reach the target gate. Since each new level considers adding all of the available gates to the sequence, each step multiplies the number of matrices to consider by 25. Thus, for a sequence of length N, there will be $25^N$ operations. The “meet in the middle” figure reveals that starting the search from the start and the goal results in the computer exploring fewer levels. Note that each side would only have to explore half as many levels since the searches meet in the middle. Thus, instead of $25^N$ operations, the computer can ideally perform $2 \cdot 25^{N/2}$ operations using the MITM (meet in the middle) algorithm.

The Middle Structure

The critical component of the MITM algorithm is the structure that allows the paths to connect. This structure effectively creates the red arrow in the MITM figure above, matching up left matrices with right matrices. It must be designed carefully to ensure optimal performance of the
algorithm. For a given left matrix, it should find a minimal number of right matrices which are close to the left matrix. Thus, the data structure needs a way to parameterize all of the matrices stored in it, using parameters that are related to the Fowler distance between two matrices.

The simplest approach is to choose some reference matrix M, and store the right matrices in a tree map, using their distances from M as keys. Then, to find right matrices that are “close” to a left matrix L, the algorithm simply measures the distance from L to M, and performs a range query for all right matrices that have about the same distance to M. This trick works because the Fowler distance measure obeys the triangle inequality: if two matrices L and R are within some distance d of each other, then the difference in their distances to some other matrix M will not be greater than d. In the figure below, this fact is true for all matrices inside the circle.

For my implementation, I use the target gate as the reference matrix, and I choose d to be $10^{-10}$ less than the smallest distance found so far. Since the left matrix must check its distance from the target gate anyway, we can re-use the distance calculation without having to cache it. Note that it is possible for two matrices to be far away from each other while still having the same distance to M. Thus, the range query may return false positives, which are shown between the red lines in the figure. The triangle inequality property simply guarantees that the range query will not leave out potential candidates.

**Building the Structure**

For each sequence S the algorithm generates, a corresponding matrix M is generated. M represents the transformation that S would perform on a quantum bit. The algorithm usually assumes that S is a prefix of the solution, meaning that other gates will be added to the end of S to reach the target gate G. However it’s also possible to consider S as a suffix, in which gates are added onto the beginning of S. In this case, S would work backwards from G, attempting to come close to the identity matrix, rather than the other way around. If the computer knows M, it can work backwards by multiplying the inverse of M with G to get a matrix M2. Then, prefix sequences can see if S is their suffix by comparing their matrices to M2. If a prefix matrix is close to M2, then it would be close to G if it were multiplied by M.

Therefore, the middle structure simply needs to store as many matrices N as possible, with pointers to their corresponding sequences. It stores a list of binary search trees by sequence.
length, so that all short sequences can be examined before long sequences.

The middle structure only has so much room to store entries, though. Since the number of unique sequences scales exponentially with the sequence length, the structure can store entries up to some length \( L \) before running out of memory. Thus, the MITM algorithm does not always cut the number of search levels in half; instead, it subtracts \( L \) from the number of search levels required to find a solution. This approach replaces the \( O(2^n) \) cost of exploring sequences of length \( N \) with a (roughly) \( O(2^{n-L}) \) cost, since a well-optimized middle structure should not have an exponential lookup time.

**Performing the Search**

Whenever the algorithm finds a new unique sequence \( P \), it checks the middle structure to see if one of the suffixes \( S \) can connect it to the target gate \( G \). Since suffixes are searched by ascending length, the first result should be of optimal length. The search function is given a distance parameter that indicates the maximum tolerable Fowler distance for the match; all matrices that are farther away are skipped. If a result is found, the search function also returns the distance \( D \) from \( P \)'s matrix to \( S \)'s matrix, so that the distance threshold can be reduced to \( D - \epsilon \) (some small value). That way, future searches will only return more precise matches.

One problem that I noted after obtaining my results is that the real sequence may not be of optimal length. The Clifford group contains elements that are composed of multiple real gates, but each Clifford group element is considered to be one gate in this algorithm. Since every sequence alternates between Clifford group elements and \( T \) gates, the number of real gates in the sequence of length \( n \) returned by the algorithm is about \( n/2 \) plus \( 3(n/2) \). However, the resulting sequence will still have an optimal real length: the Clifford group elements are ordered such that the ones comprised of multiple real gates are visited later by the algorithm, meaning they are added to the structure at a later time. Thus, if the structure uses a stable sort, these longer sequences will be considered later. I am not entirely certain that my structure does so, however, which would be a good topic for future research.

Another potential problem is that a very good suffix may be skipped because a “sufficient” suffix was encountered first. For speed, the MITM algorithm returns the first suffix that is within the desired distance threshold. Technically, if this event occurs, the improved suffix would be discovered at the next search level, so this problem should not impact correctness. However, that means the best result might not be returned as early as possible. One sufficient correction would be to continue the search; it won’t impact performance because new sequences are rarely found. This fix could be implemented in future work.

**Results**

As the graph below shows, the “meet in the middle” (MITM) optimization improved performance by an order of magnitude. Instead of taking about one hour to calculate a gate sequence that is within \( 10^{-3} \) of the target gate, it takes about ten minutes. The Unoptimized and MITM Width 15
lines both used a “width” of 15, meaning that the middle structure and Fowler’s data structures stored sequences of length 15. The actual improvement appears to depend on the width of the middle structure: when sequences of length 30 are stored in it, the time is cut by two orders of magnitude instead of one.

Note that Fowler’s unoptimized algorithm also improves performance when the width is increased, because his data structures can cache more data. Thus, it makes sense that increasing the width to 30 from 15 results in a larger improvement than just turning on the MITM optimization.

The memory requirements are much clearer as well: the number of unique sequences increases exponentially with the sequence length. I omitted data for sequences of length less than five because they adversely affect the exponential curve fit.
Finally, I noticed that the number of sequences per unit of time was much larger in the optimized versions than in the unoptimized versions, confirming my hypothesis. It clearly makes sense to keep expanding the middle structure if possible: beyond sequences of length 30, the MITM implementation with width 15 slows down relative to the implementation with width 30. However, in the long run, the MITM optimization does not change the base of the exponential that governs the algorithm run time: notice that all of the lines are roughly parallel towards the right side of the graph.

\[
y = 52.192e^{0.3495x}
\]

\[
R^2 = 0.9999
\]

I managed to approximate the \( \frac{\pi}{6} \) gate to precision in about 3 hours and 5 minutes. Here's the result:

\[
6.8 \times 10^{-5}
\]
Future Work

In future quarters, someone could try the following optimizations. Most of these optimizations will probably only reduce the computation time by some constant factor, though.

Using k-d trees for the bidirectional search middle structure

The bidirectional search middle structure only uses one parameter to index the right matrices. For the reference matrices M which I chose, many matrices had similar Fowler distances to M. Thus, while the algorithm was able to avoid iterating over some right matrices, it still had to iterate over many matrices that were not “close” to a given left matrix.

To solve this problem, I can use an additional reference matrix N, that is not close to M, to index matrices. Then, I can use a two-dimensional range query data structure, such as a k-d tree, to obtain all matrices whose distances from M and N are within a certain range. Since the calculation cost of evaluating the distance between matrices is so great, minimizing the number of right matrices matched to a left matrix should speed up evaluation remarkably.

Map-Reduce Parallelism

The Fowler algorithm can be broken down into a cycle for each sequence length. Each cycle is essentially a map-reduce job. During the map phase, we assign one gate to each computer, and that computer will consider all sequences of length N which start with that gate. Once all computers have finished the cycle, the reduce phase will merge the data structures for unique matrices, as well as the discovered gate sequences.

There are several advantages to map-reduce parallelism:

1. Unique sequence data structures can be shared with all the units between cycles. Thus, all units can benefit from each unit’s work in each subsequent calculation cycle.
2. If you keep track of the data structure contents after the final stage, you can restart the algorithm from this final stage.
3. No specialized hardware (such as a FPGA) is required. Anyone with access to Amazon’s Elastic MapReduce service, or a Hadoop cluster, can use a map-reduce algorithm.

Map-reduce parallelism will probably divide the algorithm’s run-time for a given sequence length by the number of computers involved. Thus, if there are 25 computers (for 25 gates), then the algorithm ought to run up to 25 times faster. However, since all of the computers must merge their data after each cycle, the faster computers must wait for the slower ones. Due to the complexity of the map-reduce setup, this method was not implemented this quarter. However, Amazon provides a map-reduce framework that should be straightforward to use and scale, should someone decide to adapt the program.
FPGA Tree Memory

An FPGA can have numerous tiny “node memories” dispersed throughout the board, where each node memory is assigned to a tree node. Node memories at each level of the tree are connected to their parent via a shared bus. The tree nodes at each level are connected to a bus indicating the sequence number at that level. All nodes are also connected to a very long bus, onto which the final tree value is placed. This memory provides a latency of the tree depth, and a throughput of one lookup per clock cycle. If the system runs out of node memories, it can instead return memory addresses for continuing the lookup process. Thus, main memory will still be used, but it will not receive as many requests.

Summary

This quarter, I considered a variety of optimizations to Fowler’s quantum compiler algorithm. I designed an FPGA compiler to evaluate the usefulness of a hardware solution. Then, I implemented the “meet in the middle” algorithm in software, and presented the results here. While the algorithm certainly provides a dramatic performance boost, it also requires a lot of memory to maintain the middle structure. Future work involves attempting further software and hardware-based optimizations to improve the algorithm runtime.

Acknowledgements

I performed most of this research independently, but received significant guidance and assistance from the following individuals and organizations. Without their involvement, this research project would not have happened!

- **Paul Pham** – the UW graduate student who suggested the research topic for this project, and who provided essential quantum computing context and advice. I had weekly meetings with him, and I worked with him on his pulse sequence board two years ago. He is working on his own quantum compiler based on the Solovay-Kitaev Theorem.
- **Austin Fowler** - a Research Fellow in quantum computer science at the University of Melbourne. He wrote the original paper describing the sequence-skipping optimization, upon my research is based. He also supplied the C source code to his algorithm, so that I could test my optimizations.
- **Aram Harrow** – my faculty advisor, who came up with smart suggestions for error accumulation analysis and calculation optimization. He also indirectly proposed the MITM algorithm at the beginning of this research project.
- **Carl Ebeling** – my former faculty advisor for previous honors research projects. Carl specializes in hardware development, so he offered advice for my FPGA implementation of the quantum compiler.
- **University of Washington Computer Science and Engineering Department** - provided a Terasic DE-1 prototyping board with an Altera CycloneII FPGA for testing components of my FPGA implementation.
References


Appendix A: Unoptimized Code Profile

Flat profile:

<table>
<thead>
<tr>
<th>time</th>
<th>seconds</th>
<th>cumulative self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.93</td>
<td>5.17</td>
<td>5.17 51973016 0.00</td>
<td>0.00</td>
<td>md(Matrix, Matrix)</td>
<td></td>
</tr>
<tr>
<td>26.66</td>
<td>9.78</td>
<td>4.61 208030118 0.00</td>
<td>0.00</td>
<td>cm(Complex, Complex)</td>
<td></td>
</tr>
<tr>
<td>17.48</td>
<td>12.80</td>
<td>3.02 207892064 0.00</td>
<td>0.00</td>
<td>cc(Complex)</td>
<td></td>
</tr>
<tr>
<td>14.36</td>
<td>15.28</td>
<td>2.48 155988072 0.00</td>
<td>0.00</td>
<td>ca(Complex, Complex)</td>
<td></td>
</tr>
<tr>
<td>7.12</td>
<td>16.51</td>
<td>1.23 10923 0.11</td>
<td>1.56</td>
<td>is_unique_product()</td>
<td></td>
</tr>
<tr>
<td>3.07</td>
<td>17.04</td>
<td>0.53 51973016 0.00</td>
<td>0.00</td>
<td>my_cabs(Complex)</td>
<td></td>
</tr>
<tr>
<td>0.58</td>
<td>17.14</td>
<td>0.10 17256 0.01</td>
<td>0.01</td>
<td>mm(Matrix, Matrix)</td>
<td></td>
</tr>
<tr>
<td>0.41</td>
<td>17.21</td>
<td>0.07 1 70.02 70.02</td>
<td>cs(Complex, Complex)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.26</td>
<td>17.26</td>
<td>0.05</td>
<td></td>
<td>frame_dummy</td>
<td></td>
</tr>
<tr>
<td>0.12</td>
<td>17.28</td>
<td>0.02 10192 0.00</td>
<td>0.00</td>
<td>insert_product_in_tree()</td>
<td></td>
</tr>
<tr>
<td>0.06</td>
<td>17.29</td>
<td>0.01 10923 0.00</td>
<td>0.00</td>
<td>skip_product(int)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 15463 0.00</td>
<td>0.00</td>
<td>test_product(int, int)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 10971 0.00</td>
<td>0.00</td>
<td>increment_product(int)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 10923 0.00</td>
<td>0.01</td>
<td>calculate_product(int)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 10923 0.00</td>
<td>0.00</td>
<td>add_matrix_to_unique()</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 10923 0.00</td>
<td>0.00</td>
<td>insert_product_in_list()</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 10923 0.00</td>
<td>0.00</td>
<td>add_matrix_to_structures()</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 10923 0.00</td>
<td>0.00</td>
<td>unique_greater_than_product(int)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 3863 0.00</td>
<td>0.00</td>
<td>overwrite_product(int, int)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 321 0.00</td>
<td>0.00</td>
<td>mi(Matrix*)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 221 0.00</td>
<td>0.00</td>
<td>print_gate(_IO_FILE*, int)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 26 0.00</td>
<td>0.00</td>
<td>mz(Matrix*)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 21 0.00</td>
<td>0.00</td>
<td>print_product(_IO_FILE*, int*, int)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 1 0.00</td>
<td>0.00</td>
<td>pm(_IO_FILE*, Matrix)</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>17.29</td>
<td>0.00 1 0.00</td>
<td>70.02</td>
<td>minv(Matrix)</td>
<td></td>
</tr>
</tbody>
</table>

% cumulative a running sum of the number of seconds accounted for by this function and those listed above it.

seconds the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls the number of times this function was invoked, if this function is profiled, else blank.

self ms/call the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total ms/call the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.
Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.06% of 17.29 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99.7</td>
<td>0.00</td>
<td>17.24</td>
<td>main [1]</td>
<td>is_unique_product() [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.23</td>
<td>15.80</td>
<td>10923/10923</td>
<td>calculate_product(int) [9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.10</td>
<td>10923/10923</td>
<td>minv(Matrix) [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.07</td>
<td>1/1</td>
<td>add_matrix_to_structures() [14]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.02</td>
<td>10923/10923</td>
<td>skip_product(int) [15]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>10923/10923</td>
<td>md(Matrix, Matrix) [3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>28/17256</td>
<td>mm(Matrix, Matrix) [8]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>10923/10971</td>
<td>increment_product(int) [20]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>26/26</td>
<td>mi(Matrix*) [25]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>7/321</td>
<td>pm(_IO_FILE*, Matrix) [29]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>98.5</td>
<td>1.23</td>
<td>15.80</td>
<td>10923/10923</td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.17</td>
<td>10.63</td>
<td>51962823/51973016</td>
<td>is_unique_product() [2]</td>
</tr>
<tr>
<td>3</td>
<td>91.4</td>
<td>5.17</td>
<td>10.63</td>
<td>51973016</td>
<td>md(Matrix, Matrix) [3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.60</td>
<td>0.00</td>
<td>207892064/208030118</td>
<td>cm(Complex, Complex) [4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.02</td>
<td>0.00</td>
<td>207892064/208030118</td>
<td>cc(Complex) [5]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.48</td>
<td>0.00</td>
<td>155919048/155988072</td>
<td>ca(Complex, Complex) [6]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.53</td>
<td>0.00</td>
<td>51973016/51973016</td>
<td>my_cabs(Complex) [7]</td>
</tr>
<tr>
<td>4</td>
<td>26.6</td>
<td>4.61</td>
<td>0.00</td>
<td>208030118</td>
<td>minv(Matrix) [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>6/208030118</td>
<td>mm(Matrix, Matrix) [8]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.60</td>
<td>0.00</td>
<td>207892064/208030118</td>
<td>md(Matrix, Matrix) [3]</td>
</tr>
<tr>
<td>5</td>
<td>17.5</td>
<td>3.02</td>
<td>0.00</td>
<td>207892064</td>
<td>cm(Complex, Complex) [4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>207892064/208030118</td>
<td>cc(Complex) [5]</td>
</tr>
<tr>
<td>6</td>
<td>14.4</td>
<td>2.48</td>
<td>0.00</td>
<td>155919048/155988072</td>
<td>mm(Matrix, Matrix) [8]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.53</td>
<td>0.00</td>
<td>51973016/51973016</td>
<td>md(Matrix, Matrix) [3]</td>
</tr>
<tr>
<td>7</td>
<td>3.1</td>
<td>0.53</td>
<td>0.00</td>
<td>51973016</td>
<td>my_cabs(Complex) [7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>28/17256</td>
<td>main [1]</td>
</tr>
<tr>
<td>8</td>
<td>0.6</td>
<td>0.10</td>
<td>0.00</td>
<td>17228/17256</td>
<td>calculate_product(int) [9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.10</td>
<td>0.00</td>
<td>17228/17256</td>
<td>mm(Matrix, Matrix) [8]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>138048/208030118</td>
<td>cm(Complex, Complex) [4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>69024/155988072</td>
<td>ca(Complex, Complex) [6]</td>
</tr>
<tr>
<td>9</td>
<td>0.6</td>
<td>0.00</td>
<td>0.10</td>
<td>10923/10923</td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>138048/208030118</td>
<td>calculate_product(int) [9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>138048/208030118</td>
<td>mm(Matrix, Matrix) [8]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>69024/155988072</td>
<td>mi(Matrix*) [25]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>314/321</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.4</td>
<td>0.00</td>
<td>0.07</td>
<td>1</td>
<td>minv(Matrix) [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.07</td>
<td>0.00</td>
<td>1/1</td>
<td>cs(Complex, Complex) [11]</td>
</tr>
</tbody>
</table>
This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:

index  A unique number given to each element of the table.
        Index numbers are sorted numerically.
        The index number is printed next to every function name so
        it is easier to look up where the function in the table.

% time  This is the percentage of the 'total' time that was spent
        in this function and its children. Note that due to
        different viewpoints, functions excluded by options, etc,
        these numbers will NOT add up to 100%.

self   This is the total amount of time spent in this function.

children This is the total amount of time propagated into this
        function by its children.

called This is the number of times the function was called.
        If the function called itself recursively, the number
        only includes non-recursive calls, and is followed by
        a '+' and the number of recursive calls.

name   The name of the current function. The index number is
        printed after it. If the function is a member of a
        cycle, the cycle number is printed between the
        function's name and the index number.

For the function's parents, the fields have the following meanings:

self   This is the amount of time that was propagated directly
        from the function into this parent.

children This is the amount of time that was propagated from
        the function's children into this parent.

called This is the number of times this parent called the
        function '/' the total number of times the function
        was called. Recursive calls to the function are not
        included in the number after the '/'.

name   This is the name of the parent. The parent's index
        number is printed after it. If the parent is a
        member of a cycle, the cycle number is printed between
        the name and the index number.

If the parents of the function cannot be determined, the word
'&lt;spontaneous&gt;' is printed in the 'name' field, and all the other
fields are blank.

For the function's children, the fields have the following meanings:

self   This is the amount of time that was propagated directly
        from the child into the function.

children This is the amount of time that was propagated from the
        child's children to the function.
called  This is the number of times the function called
this child '/' the total number of times the child
was called. Recursive calls by the child are not
listed in the number after the '/'.

name   This is the name of the child. The child's index
number is printed after it. If the child is a
member of a cycle, the cycle number is printed
between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole. This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The '+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.

Index by function name

  [19] test_product(int, int)  [14] add_matrix_to_structures()  [27] mz(Matrix*)
  [28] print_product(_IO_FILE*, int*, int)  [23] unique_greater_than_product(int)  [29] pm(_IO_FILE*, Matrix)