Christopher Fenton
CBMF W4761
8/16/09

CBMF W4761 Final Project Report

Introduction

Performing large scale database searches of genomic data is one of the largest problems in computational genomics. When performing a search, a computationally demanding 'alignment score' needs to be calculated between the query sequence and each individual sequence in the database in order to find the most similar sequences. Additionally, the amount of data in these databases has been growing rapidly in recent years, thanks to huge advances in the field of DNA sequencing, and has far outpaced the performance increases seen by desktop computers over the same period. Extremely fast heuristic techniques (such as FASTA and BLAST) have been developed to approximate the optimal alignment score, but the most sensitive techniques (which are guaranteed to find the best result) still require full dynamic programming algorithms such as the Needleman-Wunsch Algorithm (global alignment) or the Smith-Waterman Algorithm (local alignment). These algorithms are $O(n^2)$ in terms of computation. The FASTA package includes a program called SSEARCH34 that is capable of performing Smith-Waterman searches, but it is still quite slow compared to heuristic methods.

SSEARCH34 is a fairly simple program that takes in both a 'query' DNA sequence and a 'library' of DNA sequences to be searched. It loops through the library, computing the Smith-Waterman alignment for each sequence relative to the query sequence, and outputting the score (and optionally the alignment as well). Nearly all of the time (up to 98.6%) spent executing SSEARCH34 is spent calculating the score matrices[1]. The goal of this project is to evaluate the ability of an off-the-shelf Field-Programmable Gate Array (FPGA) to accelerate these alignment score calculations by implementing the dynamic programming algorithms directly in hardware. Due to the time

[1] Mergerm, Steve. (2006). **Reconfigurable Computing in Real-World Applications. http://www.fpgajournal.com/articles_2006/20060207_cray.htm**

constraints for this project, a subset of SSEARCH34 (and the global alignment equivalent) was implemented that only calculates the alignment score (not the actual alignment).

There have been commercial ASICs, built by now-defunct companies like Paracel, as well as commercial FPGA implementations by companies such as TimeLogic, that have been designed to accelerate these calculations. Clearly the idea is sound, but there are no known openly-available implementations. Unfortunately, hardware solutions often take much longer to develop, require a large amount of customization for the intended platform, and generally are not part of the same open-source-friendly community that benefits scientific software so much. Still, the benefits of dedicated hardware are potentially large enough to make this an attractive option for someone with the hardware background necessary to implement it.

The Problem

Both the Needleman-Wunsch algorithm and Smith-Waterman algorithm find the 'lowest cost' alignment between two DNA sequences via dynamic programming. For an m-long query and n-long database sequence, they calculate an m x n matrix of score values. Positive matches are rewarded, while mismatches and gaps are penalized. The primary difference between the two is that the Smith-Waterman algorithm replaces negatively-scoring cells with '0,' so that 'local' alignments aren't penalized. Additionally, an affine gap-penalty model was used in order to provide more flexibility and accuracy in alignment. Assuming that the reader is already familiar with dynamic programming methods like these, this project uses a two-matrix implementation, with matrix M representing the scores for matched/mismatched regions, and matrix I representing the scores for Indel (gapped) regions. The recurrence relations used are:

Global Alignment

$M(i,j) = \max \{ M(i-1,j-1) + s(xi,yi) , I(i-1,j-1) + s(xi,yi) \}$

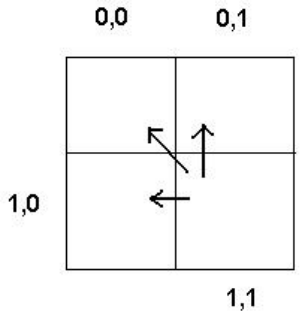
$I(i,j) = \max \{ M(i,j-1) - d, I(i,j-1) - e , M(i-1,j) - d , I(i-1,j) - e \}$

Local Alignment

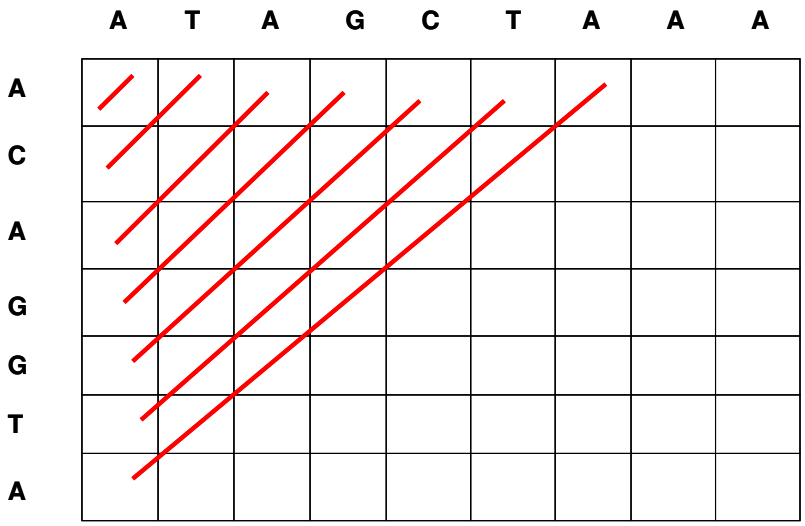
$M(i,j) = \max \{ M(i-1,j-1) + s(xi,yi) , I(i-1,j-1) + s(xi,yi) , 0 \}$

$I(i,j) = \max \{ M(i,j-1) - d, I(i,j-1) - e , M(i-1,j) - d , I(i-1,j) - e , 0 \}$

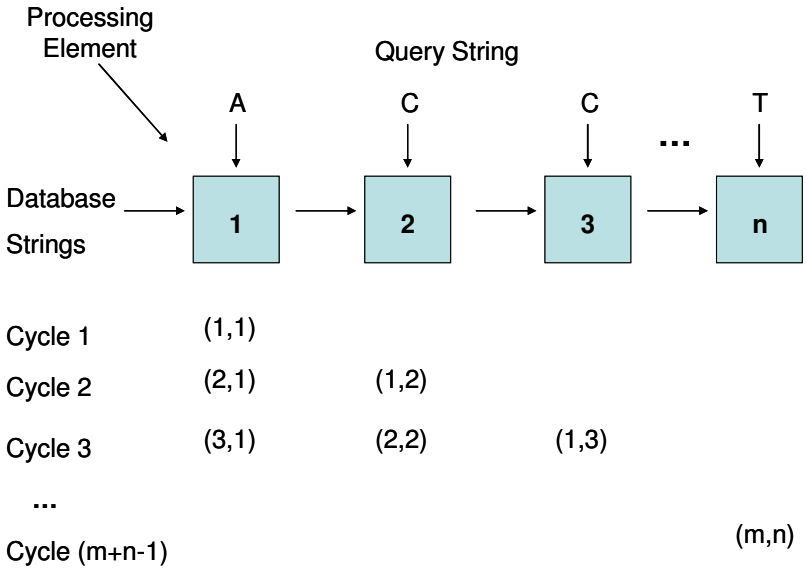
The calculation of each cell requires the scores of the neighboring left, upper, and upper-left diagonal cells.



This means that, given sufficient computational resources, we can calculate all cells that are diagonal neighbors of one another in parallel, as a 'diagonal wave' across the matrix.

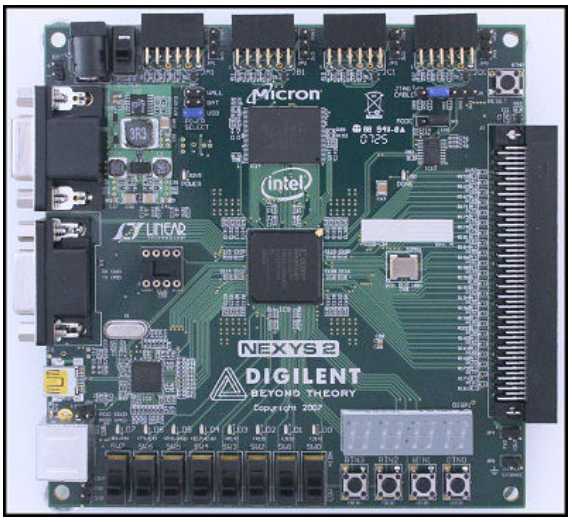


A common way of performing calculations like these in hardware is to implement a 'systolic array' of small processors, or 'processing elements' (PEs). Each processing element calculates the score in one column of the array (or both the M and I arrays, in our case). Each PE has a single nucleotide of the query sequence (the one along the top of the matrix) as an input, and the current database sequence is streamed through the array. Since we are only interested in the 'alignment score,' either the lower-right corner cell in the case of global alignment, or the highest scoring cell encountered for local alignment, we only need to store the cell values needed to compute the next cell we are interested in (as well as the highest score encountered so far). This method allows us to calculate the final alignment score in both O(n+m) space and time.



The Hardware

Although two boards were initially investigated for this project (a Dragon PCI board from KNJN.com and the USB Nexys2 board from Digilent Inc.), due to an overwhelming advantage in logic capacity, the Digilent Nexys2 board was used for the final implementation. The board contains a Cypress EZ-USB 8-bit interface, as well as a Xilinx Spartan3E XC3S1200 FPGA. The board also includes a 50 MHz oscillator, which was used as the main system clock. Digilent supplies both a binary Windows-only driver as well as a simple API for interfacing with the board, and example interface code for both the FPGA[2] and the Host[3] computer.



The Digilent Nexys2 board: The USB-enabled test board used for this project

Besides the FPGA board, all other testing and development was performed on a 3 GHz Pentium 4 processor with 2 GB of RAM, running Windows XP. Development was done in a combination of Python 2.6, C++ (using MS VC++ 6.0), and Verilog 2001 (using Xilinx ISE 8.1).

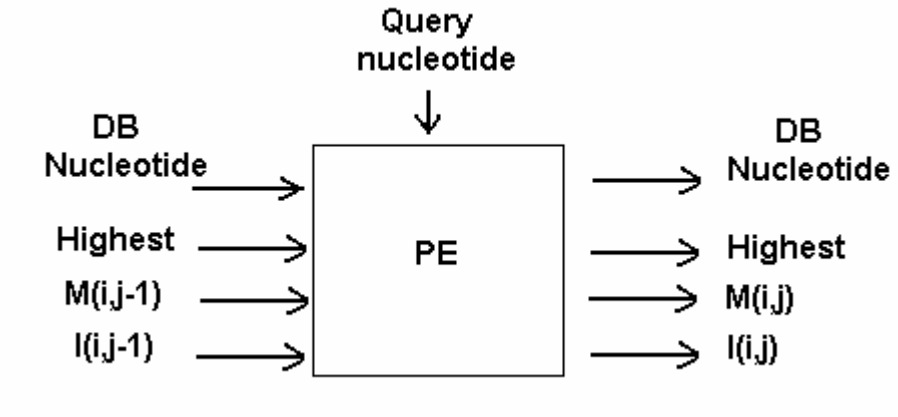
---

[2] A slightly modified version of their VHDL reference interface block was obtained from http://www.echelonembedded.com/fpgaresources/, and then further modified for this project.
[3] A modified version of their provided DPCDEMO was used to interface with the board.

Implementation

The main building block of the design is the "processing element," a simple processor that takes in a single nucleotide from the query string, and computes one column of the Match and Indel matrices as the database sequence is fed through it. Processing elements are daisy-chained together to form a 'systolic array,' and the number of elements in the design is solely limited by FPGA logic 'capacity.' In this implementation, the length of the query string is restricted to be less than or equal to the number of PEs in the design[4].

Each PE is connected to its neighboring PE with a 2-bit data bus, which carries the next nucleotide in the database sequence, and 3 score buses that carry the previously computed values for 1) the Match Matrix cell, 2) the Indel Matrix cell, and 3) the highest score seen so far (for local alignments). Due to the large number of comparators, adders and multiplexers used, the amount of logic required per PE is heavily dependent on the score width. In the tests used for this project, scores were stored as 11-bit values.



In order to actually calculate the next set of cell values, each PE needs to know the scores of the cell above it, to the left of it, and to the upper-left diagonal of it. Each PE calculates a column of the matrix, so the "above" cell values are just the values computed on the previous clock cycle (and currently being sent out via the score buses to

[4] In a more advanced design, the results of the last column could be stored and fed back into the first column, allowing arbitrarily long query sequences. This requires more area, however.

the rightmost neighbor). The “left” values are sampled directly from the neighboring PE via the score buses, and the upper-left diagonal cell values are merely the “left” values from the previous clock cycle, so they also get stored internally. Using the ‘iteration’ formulas for affine gap penalties, the new cell values are calculated in each PE on each clock cycle. An internal register also compares the new cell values to the previously highest score it has encountered as far, as well highest score the left-neighboring PE has seen so far.

After (m + n – 1) clock cycles, the last PE in the array holds the final score values. In the case of local alignment, the final score value is present on the “high” score bus, and in the case of global alignment, the final score value is present on the higher of the other two score buses (Match and Indel).

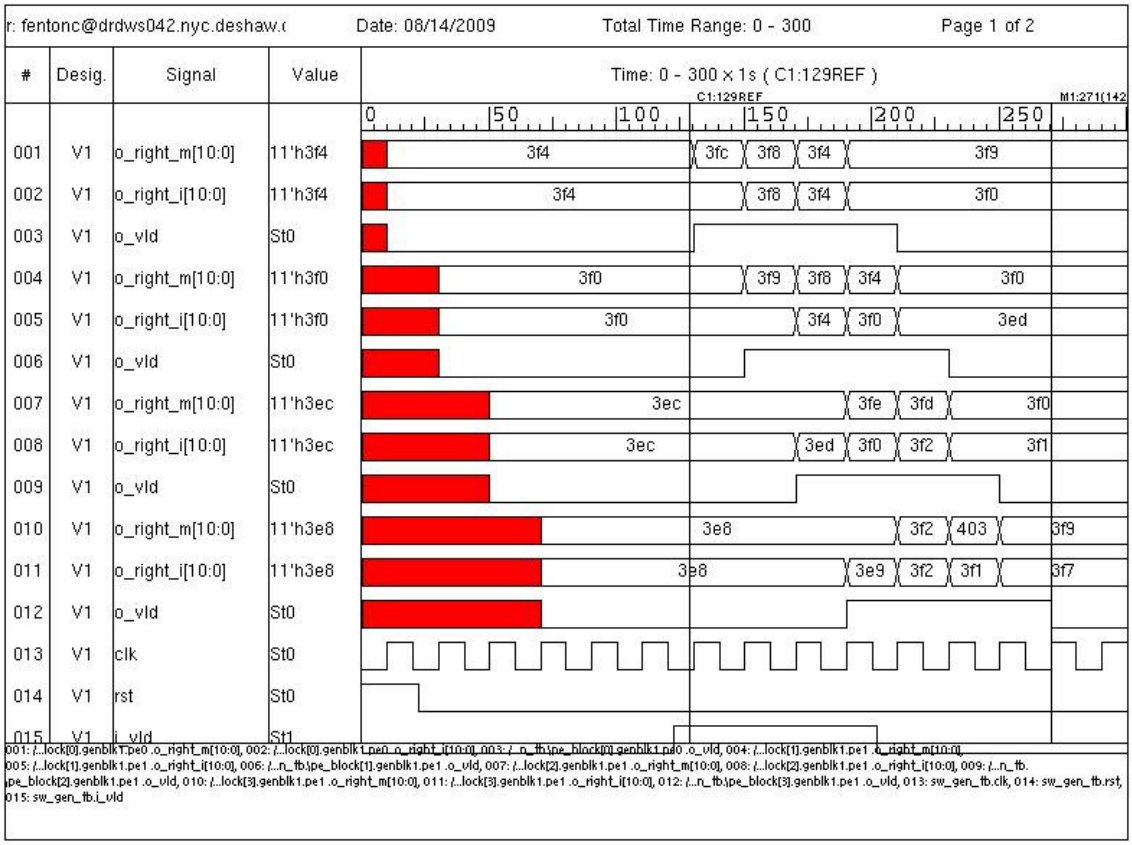
Verification and Testing

On a typical hardware development project, hardware is rigorously verified over a period of months by a team of Design Verification engineers using software packages to exhaustively test logic paths. Due to the short time constraints, validation was focused primarily on the correctness of the execution pipeline rather than the robustness of the interface logic. The execution pipeline was validated as follows:

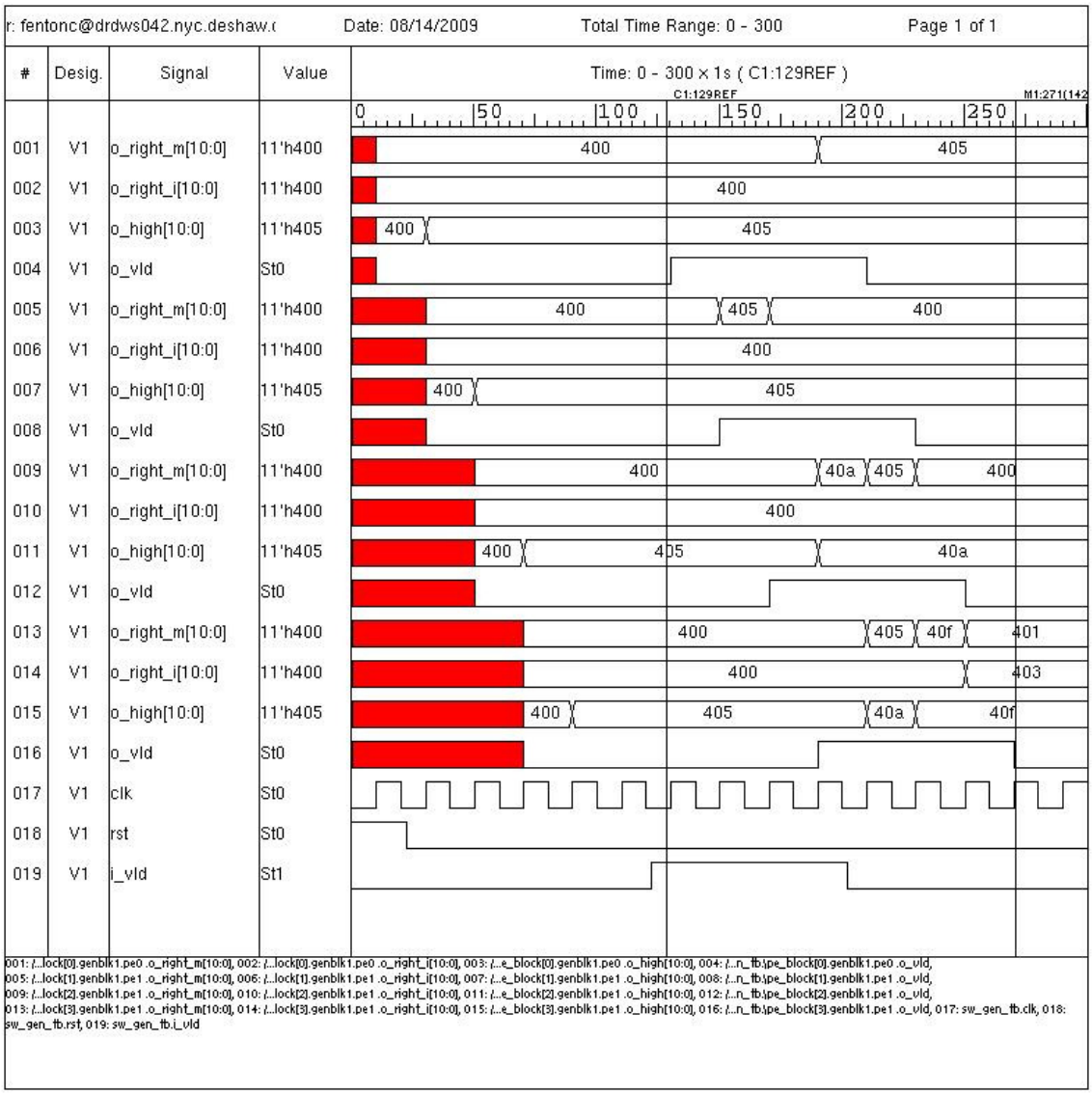
First, several short alignment examples were executed both on a hardware simulator and by hand, checking to ensure the validity of the results. This was performed for both global and local alignment modes. Calculating the score matrix by hand quickly becomes tedious as the matrices grow larger, so manual checking was only performed on matrices up to 8 x 8 nucleotides. Additionally, it was not really needed, as there are only two unique ‘types’ of processing elements: The first one in the chain, which needs to properly account for the “initialized” column of scores (M(i,0) of the score matrix), and then all other elements in the chain.

One can see below an example using 4 Processing Elements, and comparing the "query" sequence 'AGTT' against the sequence 'GTTA,' in both local and global alignment modes. The appropriate final score is produced by PE[3] 7 clock cycles (m + n -1) after the calculation is started. The matrix cell values calculated on each clock cycle are stored in the "o_right_m" (Match score) and "o_right_i" (Indel score) of each PE. The last value output by PE[3] is the 'score' for a global alignment, and the last value output on the 'o_high' output of PE[3] is the 'score' for a local alignment. The waveform clearly shows the cell values being computed as in the above diagram showing the "diagonal wavefront."

Simulation Settings	
Match Score	+5
Mismatch Score	-4
Gap Open	-12
Gap Extend	-4
Score Offset	1024 (11-bit score values)
Score Range	0-2047
Nucleotide Encodings	{A=00,G=01,T=10,C=11}



Global Alignment: A global alignment of the strings AGTT and GTTA using a 4 PE array. The "right_m," "right_i" and "vld" (data valid) signals for each PE are listed, with PE[0] on top. The comparison sequence is clocked into the array starting at time 130, and the final score arrives at time 270 on the "o_right_m" output of PE[3]. An offset value of 0x400 was used to initialize the matrix, so the final score is 0x3f9 (-7).



Local Alignment: A local alignment of the strings AGTT and GTTA using a 4 PE array. The score is calculated in the same manner as with a global alignment, only with different initialization values. The 'current highest score' is propagated along the "o_high" bus between PEs, so the highest cell value encountered still appears on the "o_high" output of PE[3] after 7 (m + n – 1) clock cycles. In this case, the final alignment score can be seen at time 270, coming out of the lowest "o_high" output in the list, and having a value of 0x40f (+15), the correct local alignment score.

Next, the local alignment mode was somewhat more thoroughly tested using a sort of 'constrained random testing.' A python script (included with the supplementary materials as 'randgen.py') was written to randomly generate a "query" sequence, and then generate a modest number of permutations of that sequence to form a database of

similar sequences. The script outputs files in both FASTA format and the custom format I utilized in my communication software. The final alignment scores are then calculated using both the custom hardware and the SSEARCH34 program (a part of the FASTA package), which performs database searches using the Smith-Waterman algorithm. SSEARCH34 was run using the default scoring matrix (match=+5, mismatch=-4) and gap penalties (Open=-12, Extend=-4), along with the "-n" command-line option to force it to evaluate the strand as DNA (otherwise it defaults to proteins, and gives incorrect results).

Query Sequence	FPGA Score	SSEARCH Score
TCTAAATACTCTCAATGCCGGGGGGGATTATC		
DB Sequence		
TCTGAATACTCTCAACGCCGGGGGGGATTATC	142	142
TCTGAATACTCTCAGGGCCGGGGGGGATTATC	133	133
TCTGAAGACTCTCACGGCCGGGGGGGATTATC	124	124
ACTGAAGACTCTCACGGCCGGGGGGGATTATC	119	119
ACGAAAGACTCTCACGGCCGGGGGGGATTATC	119	119
ACGAAAGACTCGCACGGCCGGGGGGGATTATC	110	110
ACGAAAGACTCGCACTGCCGGGGGGGATTATC	119	119
ACGAAAGAGTCGCATTGCCGGGGGGGATTATC	110	110
ACAAAAGAGTCGCATTGCCGGGGGGGATTATC	110	110
ACGAAAGAGTCGCATTGCCGGGGGGGATTATC	110	110
ACGAAAGAGTCGCATTGCCGGGGGGGATTATC	110	110
ACGACAGAGTCGAATTGCCGGGGGGGATTATC	92	92
AAGACAGACTCGAATTGCCGGGGGGGATTATC	100	100
AAGAGAGACTCGAATTGCCGGGGGGGATTATC	100	100
AAGGTAGACTCGAATTGCCGGGGGGGATTATC	99	99
ATGGTAGACTCGAATTGCCGGGGGGGATTATC	99	99

Example Comparison results: This shows the scores output in the "TEST1" example included in the supplementary material

In general, the two produced identical results. There is one exception I was able to uncover, in the case of sequences that only share one common nucleotide. As an example, comparing the sequence GGGC to the sequence CCCG should produce a local alignment score of "+5", with one aligned nucleotide:

GGGC------

-------CCCG

My FPGA correctly scores this and results in a "+5," but for some reason SSEARCH34 produces a score of 0. There may be some sort of minimum score SSEARCH34 requires, but I was unable to determine the official cause.

Performance

It is clear that this architecture is capable of performing sequence comparisons in linear time (m + n -1 clock cycles), unlike a typical serial x86 processor. Given a sufficiently large FPGA and sufficiently low-latency interconnect, this architecture is capable of far outpacing the performance of any one processor. Unfortunately, a high-end FPGA was not available for this project, and we compute with the hardware we have, not the hardware we want. That being said, a reasonable effort was put forth to accelerate performance on the Digilent Nexys2 board used for this project.

Theoretical Peak

This architecture is capable of performing 1 Cell-Update per PE per cycle. The theoretical on-chip compute capability (in Cell-updates per second) is then:

CUPS = (frequency) * (number of processing elements)

There were two final designs that underwent performance testing: A 48-PE design with affine gap support, and a 92-PE design with only support for linear gaps.

48 PE design: CUPS=(50 MHz) * (48 PEs) = 2.4 GCUPS

92 PE design: CUPS=(50 MHz) * (92 PEs) = 4.6 GCUPS

Theoretical performance scales linearly with both clock speed and number of processing elements, so a more modern FPGA's performance numbers get impressive quite quickly (something we'll explore more a bit later).

Bottlenecks

The overwhelming bottleneck to performance with the FPGA is the USB interface to communicate with it. The Digilent board uses a Cypress EZ-USB interface chip with a custom firmware that emulates an 8-bit parallel port interface. The address and data bits are multiplexed across the same lines. A Windows-only driver and API are supplied by Digilent, which provide three different methods of interfacing:

[Get/Put]Reg: Write single 8-bit address + read/write 1 data byte per driver call
[Get/Put]RegSet: Stream an array of address + data pairs per driver call
[Get/Put]RegRepeat: Stream an array of data writes (or reads) from the same location

Latency

The following data was obtained by polling a free-running 50 MHz counter running on the FPGA. The combination of driver call + USB overhead gives a single-byte latency of ~370 microseconds. Using the [Get/Put]Reg calls, this latency is incurred on every operation, severely impeding performance. Using the [Get/Put]RegSet calls still incurs the initial 370 microsecond penalty, but testing showed that each additional byte only incurred a 10 microsecond penalty. The fastest method by far is the [Get/Put]RegRepeat calls, which only incur an additional penalty of ~180 microseconds (9 clock cycles at 50 MHz) for each additional byte, but require the use of a dedicated hardware FIFO. The cause of the large additional-byte latency difference between the

[Get/Put]RegSet and [Get/Put]RegRepeat calls is unclear, but likely lies within the binary-only driver supplied.

As mentioned earlier, the actual calculation only takes (m+n-1) clock cycles to compute, so for a 92 PE array, with a 128-nucleotide comparison string, it only takes (92 + 128 – 1) = 219 clock cycles. Using a 50 MHz clock, this only takes 4.38 uS (only 3.5 uS in the case of 48 PEs), compared to the 370 uS delay of starting a new write operation. The performance is clearly dominated by the USB-related latency.

The Numbers

Once the FPGA has been setup, the actual search operation involves repeating the following steps for every entry in the database:

1. Write the reset bit
2. Write the (up to) 32-byte database string
3. Write the database string length
4. Write the start bit
5. Read the 2 resulting score bytes

The following chart shows sustained performance (in CUPS) vs. various driver and hardware optimizations that were attempted over the course of this project. Nearly all tests were done using a 92-PE, linear-gap penalty array.

# of PEs	Description	Performance	% of Theoretical
92	[Get/Put]Reg calls (35 writes + 2 reads)	850 KCUPS	.00185
92	35 PutRegSet calls + 2 GetRegSet reads	6.8 MCUPS	.15
92	Reset/start commands linked to writes of the database string. Otherwise same as above.	7.33 MCUPS	.16
92	Added on-chip 256-entry FIFO to store scores, eliminating the 2 read calls. Otherwise same as	11.84 MCUPS	.256

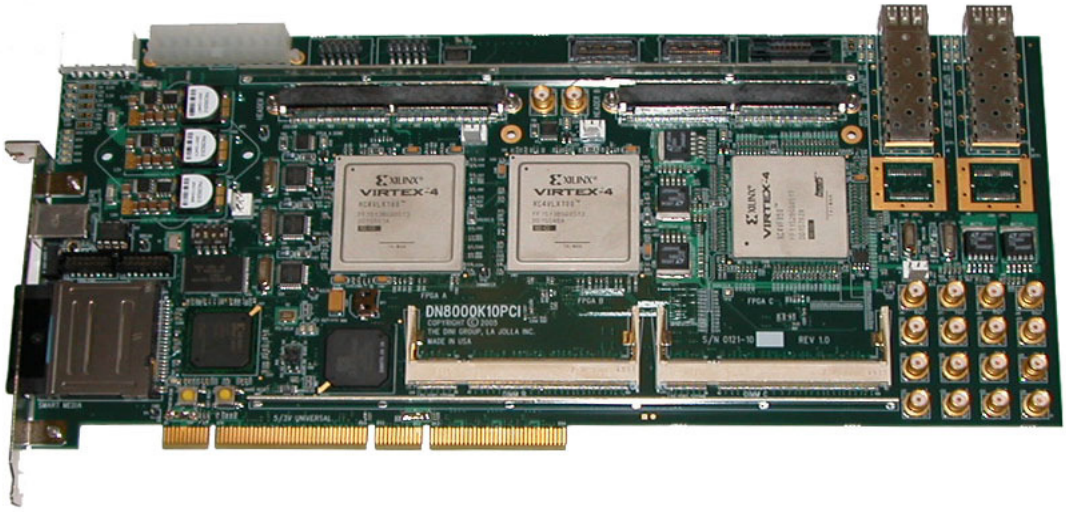
	above.		
92	Added a FIFO to receive the DB string. Uses 33-byte PutRegRepeat calls. Otherwise same as above.	960 MCUPS	20.8
92	Start receiving new DB string while previous DB string is still being processed.	1.125 GCUPS	24.45
92	Same as above. Now with 1024-nucleotide DB strings.	1.84 GCUPS	40

Testing showed that the Digilent board had a maximum bandwidth of 1 byte every 9 clock cycles over its USB interface. Each nucleotide only requires 2 bits, so each incoming byte can store 4 nucleotides. A nucleotide only requires 1 clock cycle for the array to 'consume,' however, so a single incoming byte can only keep the array 'fed' for 4 clock cycles, even though it takes 9 clock cycles to receive the next byte. Despite the peak 'theoretical' performance of the array (4.6 GCUPS in the case of the 92-PE array), due to the USB interface, you are actually limited to (4/9) * (Peak Theoretical Performance). The 92-PE array therefore has an upper limit of (4/9)*(4.6 GCUPS)=2.04 GCUPS, of which I was able to sustain 1.84 GCUPS in small bursts, or 90%!

In addition to the quite reasonable performance I was able to obtain, using even this modest piece of hardware, the performance gets even better when I include a few additional numbers. First, the board measures a mere 5" x 5" in area, and is less than an inch thick. In terms of GCUPS/inch^3, the Digilent board likely scored significantly better than any CPU-based solution. Additionally, the Digilent board excels in terms of GCUPS/Watt, an increasingly important measurement for high-performance computing. The entire board runs off of less than 2.5W (it is powered directly from the USB port), and the Spartan-3E chip remains cool to the touch at all times. All in all, I was quite pleased with the results in terms of both raw performance and relative performance.

Projected Performance

The critical path in this architecture, in terms of timing, is entirely between neighboring PEs. Due to this property, the architecture can theoretically scale linearly with the number of PEs. Although unavailable for this project due to time constraints, a high-end DINI 8000k10 FPGA board, previously used for ASIC emulation, was available through my work.



Rather than the single XC3S1200 Spartan-3E FPGA and rather pokey USB interface of the Digilent board, the DINI board has a 66-bit, 66 MHz PCI link with dedicated PCI Interface controller, as well as 3 Virtex-4 FPGAs (2 x LX-200 chips and an FX-100 chip), all connected via a high-speed parallel bus. Performance is bound only by the number of PEs I can fit onto the three chips, and the clock speed I can run them at. Projected performance was very roughly estimated by simply scaling up the number of PEs based on the available logic cells in the larger chips (I was unable to actually synthesize the design, as I lacked a license for that class of chip). The Virtex-4 LX200 chip has ~200,000 logic cells vs. ~20,000 in the Spartan-3E. The Virtex-4 FX100 has ~100,000. For the affine gap-penalty PE, that means we can fit ~480 on each of the larger chips, plus an additional ~240 on the smaller Virtex-4. With an estimated total of (480*2 + 240) = 1200 PEs, and an extremely conservative clock speed of 100 MHz, that would give us (1200 PEs) * (100*10^6 Hz) = 120 GCUPS (vs. a peak of 12.6 GCUPS for

similar calculations on the Cell Broadband Engine inside the Sony Playstation 3[5]). Due to the need to increase the number of bits used to account for the score, this number would likely be lower, but still impressive, especially considering that multiple cards could be installed in a single workstation.

Conclusion

This project shows the potential power of accelerating computational genomics algorithms using dedicated hardware, and the advantages it can provide in terms of raw performance, size and power consumption, even on relatively modest hardware. This is a topic that has been relatively well explored in Academia (and commercially) over the years, but appears to be rarely used in practice. I believe there are two primary reasons for this: 1) To non-hardware engineers, FPGAs can be intimidating devices with extremely high learning curves, and 2) there has been little attempt to leverage the previous work of others by supporting open-source efforts. Researchers in the computational genomics field have benefited greatly from readily available software tools like the FASTA and BLAST packages (and countless others). Researchers wishing to explore hardware solutions are required to 're-invent the wheel,' or buy expensive proprietary solutions from commercial companies (which have an unfortunate tendency to declare bankruptcy and leave their products unsupported).

There is not much that can be done about the first problem – Verilog is an awkward language even for those used to thinking in terms of gates and flip-flops. In the hopes of attempting to somewhat alleviate the second, I hope to upload the source-code for this project to both my personal website and the opencores.org website (the closest thing hardware designers have to Sourceforge). Computational genomics is an exciting and rapidly growing field, but it needs computer hardware that can keep up with the ever-expanding datasets being produced if it is to live up to its full potential.

---

[5] Farrar M S (2008). Optimizing Smith-Waterman for the Cell Broadband Engine - http://farrar.michael.googlepages.com/smith-watermanfortheibmcellbe