Synthesis of a Parallel Smith-Waterman Sequence Alignment Kernel into FPGA Hardware

David Greaves
Computer Laboratory
University of Cambridge
Cambridge CB3 0FD
United Kingdom
David.Greaves@cl.cam.ac.uk

Sutirtha Sanyal
Barcelona Supercomputing Center
Barcelona
sutirtha.sanyal@bsc.es

Satnam Singh
Microsoft Research Cambridge
Cambridge CB3 0FB
United Kingdom
satnams@microsoft.com

[1]

Abstract

This paper addresses the issue of making FPGA-based co-processors accessible to systems biologists who do not have an extensive knowledge of digital circuit design or hardware description languages like VHDL or Verilog. Our approach allows a software engineer to model the compute intensive core of some algorithm (the as a multi-threaded program which can then be automatically synthesized into a digital circuit. Key aspects of our approach include the ability to control the quality of results by adjusting the model to instantiate different numbers of threads and adjust how information flows between threads and well as the ability to program, debug and verify using regular software compilers and integrated design environments (IDEs).

We illustrate our approach using the kernel of the Smith-Waterman sequence alignment algorithm. This kernel is expressed as a parallel C# program which is automatically compiled into an FPGA implementation. We describe how to represent the parallel architecture of the desired circuit using multi-threaded code which models the key architectural aspects of the circuits. We argue that descriptions in a regular, high-level language that can be automatically compiled into circuits makes hardware-based accelerators more accessible to software engineers.

1. Introduction

This paper presents a technique for implementing a software algorithm for DNA sequence alignment as a hardware component which has significantly superior performance. A distinctive aspect of our approach is the use of a mainstream concurrent programming language to express the parallel architecture of the sequence alignment algorithm which is in turn used to generate a Verilog implementation. The novel contribution of our approach is that it a software developer to exploit FPGA-based co-processors to compute operations like DNA sequence matching without a requirement to be skilled in VHDL or Verilog level digital design. In particular, our flow allows a software engineer to exploit the power of an FPGA-based co-processor when a hand crafted and optimized Verilog level implementation would simply not be an option due to lack of resources or skills.

Currently, effectively exploiting an FPGA to accelerate computations that are typically performed in software involves considerable expertise in in the design of digital systems. Furthermore, the software components are typically designed in one language (e.g. C) and the hardware components in a totally different hardware language (e.g. Verilog) and the system is composed using buses (e.g. ARM's AMBA bus). Such complexity makes it very difficult for regular software engineers to exploit FPGAs to accelerate their compute intensive kernels. This is unfortunate since from a technology perspective FPGAs are becoming even more appealing as co-processors because they can be be slotted into Hypertransport sockets and onto Intel's front side bus (FSB). This results in significantly improved memory bandwidth which allows a greater class of computations to be effectively off-loaded to the FPGA and the shared memory model provides a convenient way for processors to communicate with FPGAs.

Our approach to this problem involves trying to bridge the gap between the level of abstraction at which the software engineers wish to think about

1. Proceedings of Many-Core and Reconfigurable Supercomputing Conference 2009, MRSC09, Berlin. (IEEE Computer Society).

their problem (i.e. in terms of a modern, high-level programming language) and the level of abstraction at which current tools operate for assembling embedded systems (e.g. embedded C compilers, VHDL/Verilog simulators and synthesis tools, system buses and tools like Xilinx's EDK for assembling such systems).

At first sight it may seem unwise to use a modern object-orientated language for hardware synthesis because of the need to support garbage collection, virtual methods and large run-time libraries. However, we adopt a pragmatic approach. We believe that by using even a subset of a language like C# or Java we can give considerable expressive power to the programmer and we can use features like annotations to help describe how programs should be turned into circuits. Our approach involves allowing dynamic memory allocation when the upper limits can be determined at compile time; allowing only static method calls to components that will be implemented in hardware; we do not perform any garbage collection of memory associate with hardware components; and we only support a limited run-time library.

There are many other advantages to a design flow that starts from a single description of an embedded system in a modern programming language. These include familiar syntax; readily available software development tools for debugging, profiling and other analyses; and the ability to simulate the system entirely in a single framework.

An interesting point of comparison are modern GPUs which now provide a very appealing parallel hardware co-processing platform. Initially people used GPUs for general purpose computation by re-expressing their computations in terms of graphics concepts (e.g. shaders) but now programmers have available a software based abstraction for GPU programming in C++ using the CUDA system. We believe it is important to develop similar software abstractions for programming FPGAs for co-processing.

2. Background

Much previous work has been focused on the synthesis of circuits from sequential programs. The task of taking a sequential program and then automatically transforming it into an efficient circuit is strongly related to work on automatic parallelization. Indeed, it is instructive to notice that C-to-gates synthesis and automatic parallelization are (at some important level of abstraction) the same activity although research in these two areas has often occurred without advances in one community being taken up by the other community. Both procedures are ultimately limited by the level of achievable parallelism in a program which, in turn, is limited by a number of well-known programming artifacts, such as the decidability of conditional branches and array pointer comparisons.

The idea of using a programming language for digital design has been around for at least two decades [7]. Previous work has looked at how code motions could be exploited as parallelization transformation technique [12].

Examples of C-to-gates systems include Catapult-C [20] from Mentor Graphics, SystemC synthesis with Synopsys CoCentric [4], Handel-C [10], the DWARV [21] C-to-VHDL system from Delft University of Technology, single-assignment C (SA-C) [15], ROCCC [5], SPARK [8], CleanC from IMEC [9] and Streams-C [6].

Some of these languages have incorporated constructs to describe aspects of concurrent behavior e.g. the **par** blocks of Handel-C. Our approach differs from Handel-C through the use of regular language constructs for expressing parallelism whereas Handel-C makes non-standard extensions to C. This allows our approach to use a regular compiler and its associated tools whereas the Handel-C technique requires special compilers and tools.

Jonathan Babb's group at MIT have developed an interesting system for synthesizing sequential C and FORTRAN programs into circuit by using the notions of *small memories* and *virtual wires* [3] and we have also applied to same technique to our flow. Just as we make use of an existing compiler framework based on .NET and its associated compiler support infrastructure the MIT work exploits the rich SUIF framework. We believe both of these approaches are complementary to the synthesis flow that we have developed: our flow already partitions a design into multiple, separate memory instances as possible, and there is no reason why virtual wires should not be incorporated into our system if they are required to reduce resource usage or improve performance.

A notable recent example of exploiting high level parallel descriptions for hardware design is the Bluespec SystemVerilog language [16] which provides a rule-based mechanism for circuit description which provides a new way of defining hardware behaviour while leaving the back-end considerable freedom. Indeed, we can use Bluespec as an alternative back-end for the flow presented in this paper.

Our approach involves providing hardware semantics for existing low-level concurrency constructs found in a language that already supports concurrent programming and then to define features such as the Handel-C **par** blocks out of these basic building blocks in a modular manner. By expressing concurrent computations in terms of standard concurrency constructs, we hope to make our synthesis technology accessible to mainstream programmers. Although SystemC descriptions may be very efficiently synthesized, they still require the designer to think like a digital circuit engineer. Our approach allows software engineers to remain in the software realm, to help them move computationally demanding tasks from executing on processors to implementation on FPGAs.

3. Hardware Synthesis with Kiwi

The operation of the **Kiwi** system has been described in a companion paper. In summary, it consists of a C# library of run-time functions and a compiler, called **Kiwic** that takes the CIL bytecode generated by a C# compiler and generates a synthesisable RTL description. Internally, **Kiwic** consists of a compiler front-end that converts from dynamic to static allocation and that recognizes low-level hooks for thread spawning and synchronization, followed by a conventional C-to-gates compiler that is invoked on each user thread in turn.

4. Smith-Waterman Case Study

The Smith-Waterman algorithm [19] is an example of a compute-intensive algorithm that many researchers have tried to accelerate by mapping it onto FPGA-based computing devices. However, variants of the FASTA [14] or BLAST [2], [1] algorithm offer significant improvements over the software versions of the Smith-Waterman algorithm and very effective implementations have been reported e.g. the Mercury BLASTP system [11] which trade off sensitivity for speed. However, in order to find an *optimal* alignment score it is necessary to use an exhaustive technique like Smith-Waterman since the best scored cases can be missed by FASTA or BLAST.

For this paper we focus on the Smith-Waterman algorithm because it is simpler to explain yet it has characteristics which are found in many other problems which could also benefit from FPGA-based parallelization using the approach we describe.

We implemented the algorithm twice. We wrote it as a parallel C# program that uses the **Kiwi** library and was compiled to FPGA by our **kiwic** compiler. We also wrote it by hand in Verilog RTL. The two implementations had identical signatures and can be run under the same simulation testbench and separately compiled to FPGA for comparison purposes. Both implementations are defined by the parameters in Table 1.

Scoring matrix	21x21 PAM 250
Score precision	16 bits
Width of PE, w	16 bases
Number of PEs, p,	8
Maximum query length, wp	128 bases
Unwind_factor	1

Table 1. Smith-Waterman, example design parameters.

4.1. The Smith-Waterman Algorithm

The Smith-Waterman algorithm finds an optimally-matched local subsequence between a given query sequence and and a standard sequence from the genome database. The algorithm was first proposed by T. Smith and M.Waterman in 1981. Still today it is widely used in many applications in bioinformatics.

In this work, we sped up the software implementation of the Smith-Waterman algorithm by mapping the compute-intensive part of the algorithm directly into hardware.

Typically a Smith-Waterman program consists of four major parts as shown in Figure 1 and Figure 2.

1) The query string and database strings are read-in from a FASTA formatted sequence specification file.
2) The query string and database strings are then arranged in a two dimensional matrix representing row and column respectively.
3) The core computational part of the algorithm (Described in section 6.2) walks over the entire matrix and calculates scores based on a similarity matrix (the one used in our experiments is PAM-250, a 20x20 predefined matrix storing similarity scores for all amino acids).
4) Finally, a traceback step starts from the cell having highest score and traces back the entire path that leads to that cell starting from a cell having 0 score.

It is the third step which is very high in computational intensity. But the algorithm has inherent parallelization opportunity that is commonly exploited in hardware implementations.

Part 2 is also implemented in software *but* using the custom attribute Kiwi.Hardware which dictates that this method should be synthesized into a Verilog module.

The database sequence is shifted-in and matrix values are computed dynamically. Each processing element calculates in parallel. The complete architecture of one processing element is described in section 6.3. After processing, scores are sent back to the software by using api *read_from_fsl*. At each time step a new

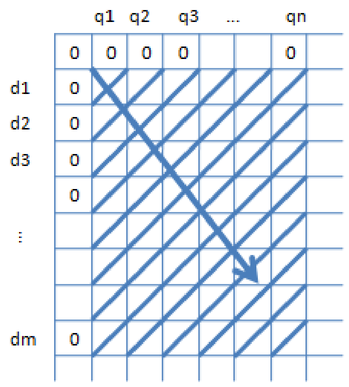

Figure 1. Smith-Waterman matrix computation.

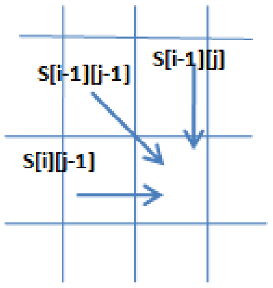

Figure 2. Calculation of score for one cell.

diagonal of the final score matrix is obtained.

The Smith-Waterman algorithm computes the similarity between two sequences of RNA, DNA or protein. One is the query sequence and another is a known sequence from database. Suppose the length of query sequence Q is n and length of database sequence D is m. We can arrange the two sequences in a matrix where rows are represented by database sequence and columns are represented by query sequence.(Figure 4a). The resulting matrix is of dimension $(m+1)(n+1)$. Now following computation is carried out in each cell having $1 <= i <= m$ and $1 <= j <= n$ to obtain score corresponding to that cell:

$$S[i][j] = Max \begin{cases} S[i-1][j-1] + \delta[i][j] \\ S[i-1][j]\text{- gap cost} \\ S[i][j-1]\text{- gap cost} \\ 0 \end{cases}$$

In the above equation, $\delta[i][j]$ corresponds to the similarity value obtained by comparing amino acids from query sequence and database sequence mapping onto that cell. For our experiments we get this value by consulting PAM-250 similarity matrix. The 'gap cost' denotes the penalty of inserting a gap while constructing the matched sub-sequence.

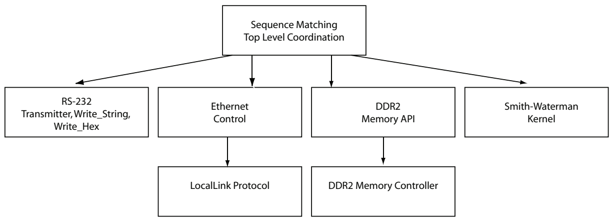

Figure 3. Top level architecture of the BEE3 Smith-Waterman co-processor

It should be noted that each element in a diagonal can be computed parallel. However computation can only proceed one diagonal at a time since each diagonal is dependent on results obtained in previous diagonal. With this configuration, a set of n processing elements (PE) is sufficient. Each database character is shifted through processing elements in each clock cycle and corresponding values for matrix elements are calculated in that cycle. The architectural details of the hand-coded hardware implementation is given in the next section.

After each cell has been assigned a score, a traceback stage starts tracing back a path from the cell having highest score to the cell having 0 score. However the traceback phase is completely sequential in nature and hence we have mapped it into software.

4.2. Architecture description of kernel in C#

The top-level architecture of the Smith-Waterman co-processor is shown in Figure 3. The co-processor system is the BEE3 FPGA system which comprises 4 Xilinx XC5VLX155T FPGAs each with two channels to external memory with each channel supporting 8GB of memory (giving a total of 64GB). The host PC sends query and data-base sequences using a raw Ethernet protocol which is implemented with a special device driver on the host Windows Vista PC and by a specially design circuit on the FPGA which can unpack sequence payloads directly into external DDR2 memory.

The design for the **Kiwi** flow uses a 'horizontal' array of processing elements that are run in parallel, each with its own C# thread. For flexibility in design exploration, we decided to implement a number of basic algorithm cells in each processing element. We programmed this using two parameters which we fix to constant values in the C# source code, but can adjust before running a **kiwic** compilation. Firstly, instead of handling just one query base, an element handles a substring of the query of width w bases. Although we can stream any length of database, m, through the hardware, long query sequences would need too many elements if each element could handle only one symbol. By adjusting $width$ we can scale up the query length n that is handled by each of p elements using $n = pw$. Using the second parameter we can experiment with the time/space folding inside each processing element because each process `unwind_factor` base symbols before pausing for a clock cycle. Hence we can have large area elements or smaller elements that do their work using a greater number of clock cycles.

Each element has a pair of channels that are sourced from the element to the left. A channel is a one-place queue implemented as a C# class that has Read and Write methods. Each element handles a fixed part of the query sequence and so is loaded with the appropriate slices of the PAM matrix to handle those symbols before processing of the database symbols starts. Between each element, one channel passes a symbol from the db sequence to the element on the right and the other passes its score, also to the element on the right. At the left-hand side, the next base element for the input dataset and a constant score of zero are passed in. At the right-most column the data is discarded. These special channel behaviors at the left and right were implemented by subclassing the channel class in three different ways: one provides a constant value, one never goes busy when written and one reads data from a database symbol staging buffer, ultimately sourced over the Ethernet.

Score data from the above and the diagonal above left is retained in the current element from the previous cycle of operation. Overall diagonal operation is emergent: it can be observed that any particular element cannot proceed until its left neighbor has processed the database symbol, and since symbols propagate from the left, elements to the right start processing later, but once busy, they remain busy every clock cycle until the database ends, when they go idle in the reverse order. They block in their channel read calls. The maximum score ever encountered by an element is just recorded in an extra public field and these fields are scanned at the end of the database by another C# function that indexes over the element array.

The C# class definition for a single element is as follows. We used C# short variable to hold the score values. Since the maximum value in the PAM array is 17, query strings of length up to 1927 can be processed without any risk of overflow.

```
public class SwElement
{ int width, unit;
  public int max;
  public int [] prev, here;
  public byte [,] slices; // Local part of the PAM array
  public Kiwi.Channel<short> left_score, right_score;
  public Kiwi.Channel<byte> left_data, right_data;
  public Thread thread;
  short diag_left_left = 0;

  public SwElement(int u, int h)
   { width = h; unit = u;
     here = new int[width];
     prev = new int[width];
     slices = new byte[width, 20];
   }

  public short run()
  { max = 0;
    byte dbval = left_data.Read();
    short topScore = left_score.Read();
    right_data.Write(dbval);
    for (int qpos = 0; qpos < width; qpos++)
        prev[qpos] = here[qpos];
    for (int qpos = 0; qpos < width; qpos++)
    { if ((qpos % unwind_factor)== 0) Kiwi.Pause();
      int above = prev[qpos];
      int left = qpos==0 ? topScore: here[qpos-1];
      int diag = (qpos == 0) ? diag_left_left:
              prev[qpos - 1];
      int score = slices[qpos, dbval];
      int nv = Math.Max(0, Math.Max(left - 10,
                    Math.Max(above - 10, diag + score)));
      if (nv > max) max = nv;
      here[qpos] = nv;
      if (qpos == width-1) right_score.Write((short)nv);
    }
    diag_left_left = topScore;
    return max;
  }

  // We need a way of exiting the elements
  // when run as a S/W program.
  public void SWOperator()
  { while(!elements_exit) run();
  }
}
```

A horizontal array of `P.UNITS` instances is created by the following code. The configurable parameters of the design were all placed in a class of constants, not shown, called P.

```
static SwElement [] Elements;

public static short run(seq query, seq db)
{ int width=(query.seq.Length+P.UNITS-1)/P.UNITS;
  Elements = new SwElement [P.UNITS];
  for (int unit=0; unit<P.UNITS;unit++)
{ Elements[unit] = new SwElement(unit, width);
  Elements[unit].left_data = (unit==0) ? new
  Kiwi.Channel<byte>() : Elements[unit−1].right_data;
  Elements[unit].left_score = (unit==0) ? new
  Kiwi.constChannel<short>(0): Elements[unit−1].right_score;
  Elements[unit].right_data = (unit==P.UNITS−1) ?
  new Kiwi.sinkChannel<byte>(): new Kiwi.Channel<byte>();
  Elements[unit].right_score = (unit==P.UNITS−1)?
  new Kiwi.sinkChannel<short>(): new Kiwi.Channel<short>();

  // Send the relevant part of the PAM matrix.
  for (int h=0;h<width;h++)
  { int sidx = h+unit*width;
    int idx = sidx>= query.seq.Length ? 0:
             symbol_to_index(query.seq[sidx]);
    for (int i=0; i<20; i++) Elements[unit].slices[h, i] =
              (byte)(pam250[idx, i]);
  }
  // Create a thread for each element
  Elements[unit].thread = new Thread(delegate()
                    { Elements[unit].SWOperator(); });
  Elements[unit].thread.Start();
}


for (int dbpos = 0; dbpos < db.seq.Length; dbpos++)
{ byte dbval = (byte)(symbol_to_index(db.seq[dbpos]));
  Elements[0].left_data.Write(dbval);
}

short max = 0;
for (int u = 0; u<P.UNITS; u++)
        max = Math.Max(Elements[u].max, max);
return max;
}
```

4.3. Architecture description of the hand-coded hardware kernel

The hand-coded implementation was designed to mirror the overall operation of the software version, using the same level of parallelism: both designs used nine threads. It would have been possible to define a larger number of smaller cells, to increase the parallelism, but our main aim was to determine if there was a performance or efficiency gap between what is generated using the **Kiwi** approach and what can be done by an RTL expert who implements precisely the same algorithm.

4.4. Results

Table 2 presents the main difference between the hand-coded and **Kiwi** synthesized designs. This is the length of the RTL code. As per Table 1, the hand-coded version used 8 instances of a cell module that if in-lined to resemble the flat design generated by **kiwic** would lead to a file approximately 977 lines long.

Design	RTL Length	State	CUPs/Clock
Hand	396 lines	59877 bits	8/19 = 0.42
Kiwi	27421 lines	68666 bits	8/20 = 0.40

Table 2. Comparison with hand-coded design.

With an unwind factor of 1, meaning that $w = 16$ clock cycles should be used for each db element, the measured results in simulation were 19 and 20 clock cycles per db element for the two designs. The overheads arise owing to the way data was passed along channels from one element to the next.

Table 3 shows the device utilization and clock frequencies achieved for the two designs when elaborated for two contemporary FPGA devices using Synplicity Synplify Premier.

Although the CUPs/Clock metric for this design is quite low, at 0.4, compared with previous FPGA implementations (§4.5), this could be increased by simply instantiating further cells. For instance, we used only 15 percent of the Altera array and hence CUPs/Clock could be easily raised to 2.6. However, this would still be a comparatively low figure, compared with (§4.5). A figure closer to 100 would be reached using a larger number of smaller cells owing to the additional parallelism. However, we cannot yet report that result because of heap size problems when **kiwic** is applied to that larger designs.

Running the C# program under .net or mono resulted in output that was nearly identical to the output from running the version created by **kiwic** using the Verilog simulator. The main difference was a permutation of printing order for `$display` statements executed in a common clock cycle. It is interesting to note that the mono version, running on a 2 GHz AMD Turion core, performed at 117 kCUP/s, but clearly this style of execution is intended only for debugging and development, rather than efficiency. An un-threaded version would certainly run faster when the mono interpreter only uses one core of the workstation, as was the case with this result.

Writing the code to convert from ASCII base characters and initializing the slices was much easier and natural in C# than in Verilog RTL.

Here is an example of the console output, generated by the `Console.WriteLine` calls in the C# program that are converted to Verilog `display` calls:

```
Starting processing of database
    length 894 at time 689100
Updating max to 6 at 0
Updating max to 11 at 1
Updating max to 20 at 2
Updating max to 26 at 3
```

Design	FPGA PART	Device	Utilization	Levels	Clock	CUP/s
Hand coded	Altera Stratix III	EP3SL340	5536 ALMs	28	138 MHz	58×10^6
Hand coded	Xilinx Virtex V	XC5VLX155T	5215 LUTs	25	101 MHz	42×10^6
Kiwi	Altera Stratix III	EP3SL340	20925 ALMs	37	83 MHz	33×10^6
Kiwi	Xilinx Virtex V	XC5VLX155T	55306 LUTs	86	46 MHz	18×10^6

Table 3. FPGA Performance Results (figures from Synplicity Premier).

```
Updating max to 30 at 5
Updating max to 36 at 6
Updating max to 42 at 7
```

4.5. Comparison with Previous Results

Oliver et. al. [17] exploit dynamic reconfiguration to optimize the kernel computation performed by the processing element by customizing it to the query string. They report a performance of up to 8.0 GCUP/s for a query length in the range 841 – 1004 using a RC2000 FPGA mezzanine PCI-board with a Virtex-II XC2V6000 from Celoxica. In related work [18] they report a 170 speedup for linear gap penalties and 125 for affine gap penalties.

One project used the XtremeData Inc. XD1000 board which includes an Altera StratixII FPGA and a dual Operaton processor to achieve a speedup of about 180X for a sequence length of 65536 [23].

An early implementation of a systolic Smith-Waterman cell was reported by Kwong et. al. [22] which worked on the Pilchard platform which uses a Xilinx Virtex XCV1000E-6 FPGA (1228 slices) and an SDRAM interface instead of a conventional PCI bus to communicate with the host system.

Li et. al. [13] used an unclocked implementation of the cell computations rather than the typical synchronous implementation which results in each major diagonal being processed at once. This approach yielded a 160-fold performance improvement over software versions of the algorithm.

5. Future Work

It appears that **Kiwic** causes clock cycles to be wasted as data is transferred through the one-place channels, although, interestingly, almost exactly the same amount of overhead occurred in the hand-crafted code as in that generated by our compiler. The compiler is designed so that two threads can operate on a single shared-variable an unlimited number of times per clock cycle, so the reason for this overhead in the synthesized design needs further investigation.

As we use more of the FPGA for longer sequences n or lower width w, we will get more parallel speed up. There are four FPGAs on the BEE3 board, so our utilization of the total hardware resource in this experiment is actually four percent.

kiwic needs to be modified to use less memory or to better support an incremental or modular compilation arrangement.

The C-to-gates backend we used was an old piece of code that does not embody the latest developments in auto-parallelization and scheduling. The code it generates does not seem to mesh well with today's FPGA tools. A next step might be to investigate generating Bluespec System Verilog instead of conventional Verilog.

Although Synplicity Synplify Premier outperformed the FPGA tools provided by the FPGA vendors, it took 45 minutes to place and route the design. This must be added to the **kiwic** compilation time of 15 minutes to give the turn around time for a result. Of course, with the **Kiwi** approach, a certain level of design validation can be carried out quickly by running the C# design on the mono or .net virtual machines, but it would be nicer to be able to move to the FPGA execution platform in a more timely way. Producing a faster version of **kiwic** should not be a major problem and this could then emit its internal VM code directly for interpreting on a '*default standard*' programming of the FPGA resources, thereby avoiding the place and route time.

6. Conclusions

In this work we investigated whether well-known parallel constructs used for software programming could be exploited as a basis for hardware acceleration of a scientific application. The techniques were the single-place buffer and threading. We encouraged the user to use a greater number of threads than might be natural, since we exploit this parallelism for spatial layout across the FPGA, yet he still had a working program that could be run under .net or mono. Another source of parallelism is that extracted from within each single thread, which was provided by a conventional C-to-gates backend flow.

Our main design parameter was the upper bound of the query sequence. We also implemented two secondary parameters to support architectural exploration and sensitivity analysis. However, we did not report on adjusting the secondary parameters since we clearly need to leverage recent work on C-to-gates and perhaps develop a C-to-Bluespec flow which will give us better basic silicon use efficiency.

By entering the same design in two different ways, we discovered that it was much easier to write and debug the application in C# than in synthesisable RTL, but that the spatial efficiency of the C# result was poor. However, we expect to improve on this situation.

In the future we will demonsrate streaming data to a parallel bank of FPGA booards over their Ethernet network interfaces. We will extend our Kiwi development framework accordingly, so that it remains straightfoward to migrate code from workstations or server blades to FPGA without rewriting the software or changing the structure of the interconnection network.

Our results demonstrate good preliminary results to support a flow that allows software engineers to model an algorithm using standard concurrency constructs and then synthesize them to good digital circuits which perform significantly faster than their software counterparts but are not as fast as hand crafted designs. We believe that there is a growing demand for software engineers and systems biologist to exploit FPGA-based co-processors to accelerate compute intensive kernels. The software-based flow that we present here is a step towards a system which makes the power of reconfigurable computing accessible to a much wider audience that is already having to learn parallel and concurrent programming due to the arrival of multi-core processors.

References

[1] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protien database search programs. *Nucleic Acids Research*, 25(17), 1997.

[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alginment search tool. *Journal of Molecular Biology*, 215(3), 1990.

[3] Janthan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. *7th IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[4] Francesco Bruschi and Fabrizio Ferrandi. Synthesis of complex control structures from behavioral systemc models. *Design, Automation and Test in Europe*, 2003.

[5] B. A. Buyukkurt, Z. Guo, and W. Najjar. Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs. *Int. Workshop On Applied Reconfigurable Computing*, March 2006.

[6] M. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high hevel language. *8th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[7] Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, 14, April 1997.

[8] Sumit Gupta, Nikil D. Dutt, Rajesh K. Gupta, and Alex Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. *International Conference on VLSI Design*, January 2003.

[9] IMEC. CleanC analysis tools. *Web page* http://www.imec.be/CleanC/, 2008.

[10] Celoxica Inc. Handel-C language overview. *Web page* http://www.celoxica.com, 2004.

[11] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, Brandon Harris, and Roger Chamberlain. Mercury BLASTP: Accelerating Protein Sequence Alignment. *Transactions on Reconfigurable Technology and Systems*, 1(2), 2008.

[12] Monia S. Lam and Robert P. Wilson. Limits of control flow on parallelism. *The 19th Annual International Symposium on Computer Architecture*, May 1992.

[13] Issac TS Li, Warren Shum, and Kevin Truong. 160–fold acceleration of the smith-waterman algorithm using a Field Programmable Gate Aarray (FPGA). *BMC Bioinformatics*, 8(185), 2007.

[14] D.J. Lipman and W.R. Pearson. Rapid and sensitive protien similarity searches. *Science*, 227(4693), 1985.

[15] W. A. Najjar, A. P. W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8), 2003.

[16] Rishiyur Nikhil. Bluespec SystemVerilog: Efficient, correct RTL from high-level specifications. *Formal Methods and Models for Co-Design (MEMOCODE)*, 2004.

[17] Timothy Oliver and Bertil Schmidt. High performance biosequence database scanning on reconfigurable platforms. *International Parallel and Distributed Processing Symposium*, 2004.

[18] Timothy Oliver, Bertil Schmidt, and Douglas Maskell. Hyper Customized Processors for Bio-Sequence Database Scanning on FPGASs. *International Symposiumm on Field-Programmable Gate Arrays*, 2005.

[19] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1), 1981.

[20] Andres Takach, Bryan Bower, and Thomas Bollaert. C based hardware design for wireless applications. *Design, Automation and Test in Europe*, 2005.

[21] Y. D. Yankova, G.K. Kuzmanov, K.L.M. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis. DWARV: Delftworkbench automated reconfigurable VHDL generator. *17th International Conference on Field Programmable Logic and Applications*, August 2007.

[22] C. W. Yu, K. H. Kwong, K. H. Lee, and P. H. W. Leong. A Smith-Waterman Systolic Cell. *Proceedings of the 13th Int. Workshop n Field Programmable Logic and Applications (FPL'03), Springer, LNCS 2778*, 2003.

[23] Peiheng Zhang, Guangming Tan, and Guang R. Gao. Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications*, 2007.