Technical Report
MECSE-23-2005

A survey of FPGA-based high performance computation in molecular biology and other domains

T. Ramdas and G. Egan

A survey of FPGA-based high performance computation in molecular biology and other domains

Tirath Ramdas, Gregory Egan

***Abstract*— Molecular biocomputation workflows traditionally involve days of compute time to align DNA/protein sequences. Custom computing machines (CCMs) provide a means to dramatically reduce alignment time, and FPGAs provide a practical means to implement such CCMs. Software implementation of some sequence alignment algorithms suffer quadratic time performance, however CCM implementations may be highly parallelized and consequently provide linear time performance. Similarly, CCMs may be used to accelerate workflows or operations in a wide range of domains, often dramatically outperforming large scale clusters. Programming and integration problems limit CCM usage, though progress has been made to overcome these problems. With continued development of tools, devices, and integration solutions, CCMs on FPGAs coupled to conventional systems present an effective architecture for high performance computing.**

***Index Terms*— Field programmable gate arrays, parallel architectures, pattern matching, programming.**

I. Introduction

THE classical scalar von Neumann architecture has endured due to its ability to provide adequate performance and programmability for virtually any application. Nevertheless in many cases application specific designs will undoubtedly provide far superior performance. Economic concerns have typically provided strong resistance against application specific solutions, however programmable logic devices – specifically Field Programmable Gate Arrays (FPGAs) – are diminishing the veracity of such arguments. Traditionally used merely as a prototyping technology for ASIC designs or glue logic for incompatible devices and protocols, increasing chip speed and increasing equivalent gate count have resulted in FPGAs being adopted for use as computational engines in their own right across a wide range of applications. In some applications, such as in the seismic and biocomputation domains, custom computational devices have dramatically outperformed large-scale clusters for specific applications [1][2][3].

While the success of FPGAs as custom computing machines (CCMs) may be largely attributed to economic factors – and even by itself this would not be a trivial contribution – reconfigurability does offer some distinct advantages over traditional ASIC solutions. With some applications, significant performance gains can be achieved by reconfiguration based on data that is only available at run-time [4][5][6]. In addition, it would be possible to swap hardware blocks in and out of an FPGA in a sense similar to context-switching [7]. It is also possible to fit multiple independent blocks within the same FPGA device, provided IO resources are appropriately allocated. Conversely an appropriately designed large block may be broken into stages and deployed in pieces sequentially on a smaller FPGA device, with interim computational results held in external memory; this approach is adopted in [8] for sequence alignment with small FPGA devices.

This paper surveys the state of the art in FPGA technology for high performance computing (HPC), with molecular biocomputation as a sample application domain. The nature and demands of some biocomputation algorithms are discussed, followed by a description of some existing biocomputation machines, especially Smith-Waterman accelerators. There is a critical need for better programming tools and methodologies for reconfigurable computing, as well as high performance integration for communication with a host machine and to a lesser extent peer computing units – these and other future development issues are discussed, followed by concluding remarks.

II. Computational Molecular Biology

With the volume of available genetic data doubling every six months, advances in conventional computing systems seem unable to keep pace. Software heuristic approaches, such as the Basic Local Alignment and Search Tool (BLAST), have contributed greatly to overcoming this growing performance deficit, however in the long-run even solutions such as these will not be sufficient. Consequently, CCMs are increasingly being coupled to conventional systems in order to accelerate biocomputing workflows.

Computational molecular biology covers sequence analysis as well as structure analysis, however this paper will focus primarily on sequence analysis, or more specifically sequence alignment, which is basically a form of pattern matching. Sequence alignment algorithms arguably represent the most important class of algorithms in molecular biocomputation. The general idea is that comparing a genetic or protein sequence against another sequence could reveal a level of homology – i.e. how closely related the two sequences are in an evolutionary sense – and could help determine the function of new sequences. Revealing similarities between

Manuscript received 30 May 2005.

T. Ramdas {Tirath.Ramdas@eng.monash.edu.au} is with the Department of Electrical and Computer Systems Engineering, Monash University.

G. Egan {Greg.Egan@eng.monash.edu.au} is the Director of the Center for Telecommunications and Information Engineering, Department of Electrical and Computer Systems Engineering, Monash University.

protein sequences may also expose similar structural traits, and therefore similar functionality in some regions of different proteins.

The fundamental sequence alignment algorithms are Needleman-Wunsch (NW) for global alignment and Smith-Waterman (SW) for local alignment, with the latter being more commonly used in biocomputation. While global alignment seeks to determine the best match between two sequences from one end to the other, local alignment seeks to determine the best alignment between subsequences of the two query sequences, and the latter turns out to be a more useful endeavor in molecular biology [9]. Due to the computational complexity of these algorithms, the BLAST tool, which is a heuristic based local alignment tool, has gained widespread acceptance by the biocomputing community despite the fact that BLAST provides lower sensitivity than SW. Another approach to sequence analysis is to use Hidden Markov Models (HMM), which offer a probabilistic approach to homology detection. A primer on sequence alignment is beyond the scope of this paper, however a general performance-oriented view of sequence alignment algorithms will be presented.

A. *Smith-Waterman and Needleman-Wunsch*

The NW and SW algorithms share many similarities – both algorithms consist of three steps: initialization, matrix fill, and traceback. Traceback reveals the optimal match alignment between two sequences (or segments of sequences for local alignment) based on each cell's score, which is calculated at matrix fill. NW and SW differ only in the traceback step. As SW is a more commonly used algorithm than NW and the two algorithms are so similar, the rest of this paper focuses on the SW algorithm, with comparisons drawn with NW occasionally.

A matrix is constructed with one sequence lined up against the rows of the matrix, and another against the columns, with the first row and column initialized with some value (usually zero) – i.e. if the sequences are of length M and N respectively, then the matrix for the alignment algorithm will have $(M + 1) \times (N + 1)$ dimensions. The matrix fill stage scores each cell in the matrix: this score is based on whether the two intersecting elements of each sequence are a match, and also on the score of the cell's neighbors to the left, above, and diagonally upper-left. Three seperate scores are calculated based on all three neighbors, and the maximum score is assigned to the cell. This is done for each cell in the matrix; doing so sequentially is therefore clearly an $M \times N$ operation. Even though the computation for each cell usually only consists of additions, subtractions, and comparisons of integers, the algorithm would nevertheless perform very poorly as the lengths of the query sequences become large.

At this point in the algorithm, a similarity score may be extracted from the matrix that quantifies the level of homology between the two sequences. To extract the optimum alignment from the matrix, a traceback is performed. This entails following the trail left behind from each cell to it's preceding cell in the matrix fill stage. One good way to achieve this is to note during the matrix fill stage which neighbor each cell had obtained it's maximum score from – i.e. if it obtained it's maximum score from the upper, left, or upper-left cell. Up till this point, the SW and NW algorithms are identical, however they diverge when it comes to the way this information is used. With NW the traceback starts at the last cell in the matrix and traces the maximal score path back to the first cell. With SW, traceback starts at the cell with the highest score in the matrix and ends at a cell when the similarity score drops below a certain threshold. The traceback stage is essentially not computationally intensive, and furthermore most of the work may be done during the matrix fill stage. One problem to consider is how to organize and store the traceback path data in an efficient manner, which is beyond the scope of this paper.

Parallelization of SW workflows may be achieved on a fine grained level or a coarse grained level. In practice, SW is often run with one query sequence against a sequence database, and in that scenario it is highly practical to distribute pairwise SW alignments of different queries to multiple nodes, for example in a commodity cluster. This coarse grained level of parallelism achieves linear speedup.

Fine-grained parallelization – or parallelization of the SW kernel in order to reduce $O(N^2)$ complexity – is complicated by data dependencies whereby each cell $c_{j,k}$ depends on the values of three neighboring cells $c_{j,k-1}$, $c_{j-1,k}$, and $c_{j-1,k-1}$ – with each of those cells in turn depending on the values of three neighboring cells, which effectively means that this dependancy extends to every other cell in the region $C_{\leq j, \leq k}$. This implies that it is possible to simultaneously compute $c_{1,3}$, $c_{2,2}$ and $c_{3,1}$, since these cells fall outside each other's data dependency regions.

In simple terms, assuming a square matrix (i.e. $N \times N$, with both query sequences being of length N) and only considering the upper left half of the matrix, the cells that may be computed in parallel (i.e. the cells for which all data dependancies are currently satisfied) are given by $c_{x-y+1,y}$ for $1 \leq y \leq x$ and $x \leq N$, which gives that x is the largest row value of the current set of computable cells. From this it may be concluded that the maximum number of cells that may be computed in parallel is N. This model is mirrored for the lower right half of the matrix. Expanding this model for non-square matrices introduces some non-linearities, but the parallelization implications remain the same. Given that it takes $x = N$ cycles to compute the upper left half of the matrix, the lower right half would consume $N-1$ cycles, since it would be redundant to recompute the cells where $x = N$. Therefore, a parallelized SW would require $2N-1$ operations, or generalizing for non-equal query sequences of length M and N, parallel SW requires $M + N - 1$ operations, i.e. an $O(N)$ implementation is possible.

However, parallel execution in this manner raises many practical problems, as discussed in [10]. These problems include:

1) Communication overheads – In order to satisfy data dependencies across all processing elements, the value of each cell once computed must be broadcasted to the corresponding neighbor processing elements. Although the communication overheads are somewhat limited due to the locality inherent in the algorithm from the

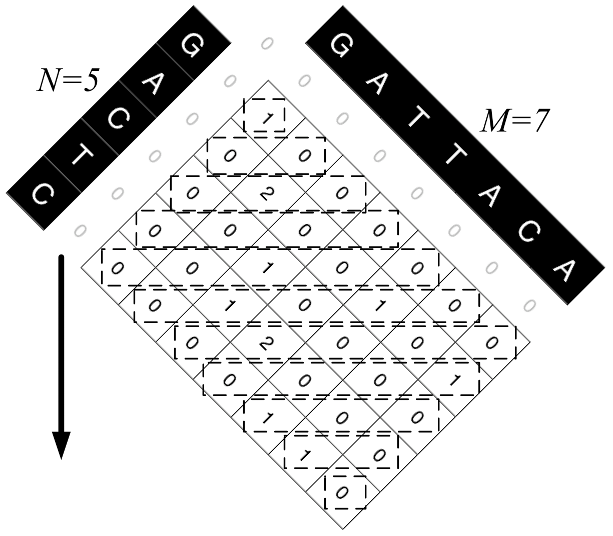

Fig. 1. A sample similarity matrix for two sequences, with bounding boxes indicating which cells are computed in parallel. The arrow indicates the direction in which the computation progresses, i.e. from top to bottom with the cells in the bounding boxes being simultaneously computed in each cycle. A total of 11 cycles are required for this computation, and a maximum of 5 cells may be computed in parallel.

perspective of each cell (i.e. each cell depends only on three adjacent cells), the fact that each cell in each cycle requires very simple computation but is dependent on data present on other cells places a lot of pressure on the communications infrastructure, and consequently processing bottlenecks would very often be due to communication limitations.

2) Load balancing – It would be possible to break the matrix down into smaller sub-matrices for computation across several nodes, however data dependancies must still be satisfied. Furthermore, as the number of available computations varies at each step efficient use of available processing elements would require that the number of cells allocated to each processing element vary at each cycle in the computation, and this would introduce some additional complexity into the parallelization.

Some actual implementations parallelize the algorithm in sub-optimal ways in order to simplify some aspects of the overall design, for example in [11] parallel processing occurs parallel to the query sequence as opposed to parallelizing along the diagonal of the matrix, though in any parallel implementation data dependancies must be satisfied.

Parallel SW may be implemented utilizing Single Instruction Multiple Data (SIMD) vector processing technologies present in contemporary CPUs such as Intel's Streaming SIMD Extensions (SSE), and up to six-fold speed-up over the base hardware (i.e. not exploiting SIMD facilities) may be achieved in this manner [11]. Large scale cluster computation is often adopted for sequence alignment, and a 120 node computational cluster, with a carefully crafted parallelization of SW, has been shown to yield a 90-fold speed-up [10]. While these levels of speed-up are valuable (especially in the case of utilizing SIMD technology, as such approaches usually require no additional expense), much higher speed-up factors may be obtained with CCMs (for example, a 330-fold speed up over desktop workstation performance was noted in [8]), and without the overheads associated with computational clusters (such as power requirements and system management/monitoring).

CCM's may be deployed on ASICs or FPGAs, and Smith-Waterman implementations have been demonstrated for both technologies. However, FPGA based solutions are more practical. There are many FPGA-based Smith-Waterman hardware accelerator designs available [3][5][6][8][12][13][14], and this paper will focus on aspects of these designs. Some SW implementations attempt to compute a maximum score, while some compute a minimum penalty – while the detailed design of each processing element may vary slightly, the overall architecture of each processing element and the manner in which the elements are integrated, and consequently the behavior of the system, is the same regardless of which approach is adopted.

B. BLAST

The BLAST algorithm makes the assumption that matched alignments will contain short stretches of perfect matches; therefore, once these stretches are identified it would be possible to extend away from these matches in search of a longer alignment. This approach works reasonably well when the query sequences exhibit a high degree of similarity, but BLAST struggles with dissimilar sequences. Nevertheless, FPGA based acceleration hardware exists for the BLAST algorithm as well [3][16][17]. It should be noted that while BLAST provides a computationally inexpensive alternative to sequence alignment over Smith-Waterman, the memory requirements are far greater. As processor performance improvements surpass memory performance improvements, this is likely to become a non-trivial problem.

C. Hidden Markov Models

As a software implementation, BLAST enjoys far better throughput than SW because BLAST does not attempt to be exhaustive; a heuristic approach is adopted that only fails with highly dissimilar query sequences. Similarly, the Hidden Markov Model approach provides a probabilistic – and non-exhaustive – tool for sequence analysis. A Hidden Markov Model may be thought of as a probabilistic state machine. Speech recognition is a very well-known application of Hidden Markov Models, and in fact FPGA based HMM accelerators for speech recognition have been implemented, such as in [19] where a 40 times speedup over a pure software implementation was achieved. The HMM approach to sequence analysis requires a set of unaligned sequences that are related as training data, from which a probabilistic model is built. This model may then be used to perform multiple alignments. The HMM approach is useful as it can categorize an entire family of sequences, however the effectiveness of this approach is often limited by the training data provided. Commercial FPGA based hardware accelerators for HMM based molecular biocomputing have also been provided by [3].

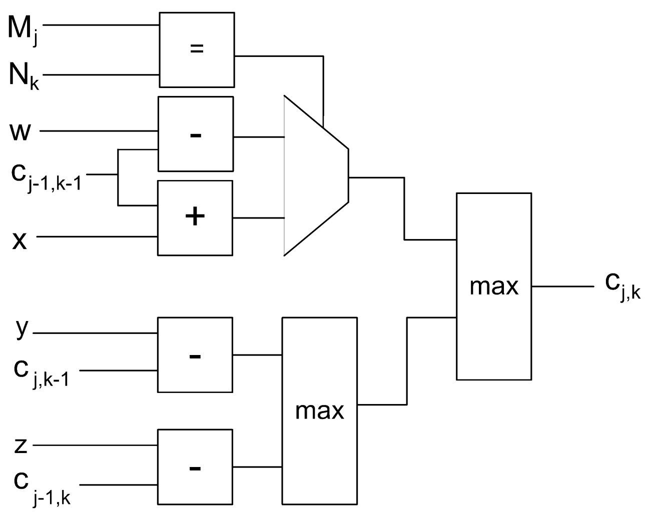

Fig. 2. A typical SW processing element, corresponding to a cell in the matrix. The inputs w, x, y and z are constant parameters of the SW algorithm. Only scoring is depicted in this diagram; in a full fledged implementation there needs to be a mechanism to indicate which input cell the score originated from, so that traceback may be performed.

III. Custom Computing Machines

As is the case with many matrix-based algorithms, systolic arrays are an elegant and efficient approach to instantiating the Smith-Waterman algorithm in hardware. When implemented as software, the Smith-Waterman algorithm is implemented by dynamic programming, and is therefore very computationally expensive - specifically it is an $O(N^2)$ algorithm. However, when implemented as hardware, massive parallelism achieves $O(N)$ performance.

Therefore, while custom hardware doesn't come close to the clock rates of current generation VLSI microprocessors, as sequence length N becomes appreciably large the performance of the custom hardware becomes vastly superior to a software based solution. This is the general tenet behind moving certain processes out of software and into hardware.

A. Smith-Waterman

When implemented as parallel hardware, the SW algorithm is usually implemented with a systolic array. A systolic array is made up of a set of processing units that are connected to a small number of neighbours in a mesh-like topology. In this case, as is usually the case with systolic arrays, the array is homogeneous – i.e. all the processing elements are identical – and according to the SW algorithm as previously described, each element is connected to three other elements, corresponding to the upper, left, and upper-left neighbors in the matrix. Each processing element needs to perform additions, subtractions, and comparisons of integers in order to determine the cell's maximum score. Individual implementations may vary, but basically the only hardware function units required in each processing element would be integer adders, comparators, and multiplexors.

One of the main challenges in the parallelization of SW is to do with communication requirements – the result of the computation of each cell needs to be communicated to it's three dependent neighbors. While this poses some problems in a commodity cluster, it is far less of a problem in a single-chip system like an FPGA-based implementation. Furthermore, the systolic array architecture suits this kind of computation very well.

However, while general purpose computers often have ample amounts of main storage, CCM boards typically do not have very large amounts of storage. If two very large and highly similar sequences are aligned, a large number of scores must be stored, and in addition very high similarity scores in many parts of the matrix are to be expected, and storage for a large range of values (from 0 to n) must be considered. The maximum score would be obtained if two identical sequences are aligned, in which case $n \propto N$. A basic fixed-length unsigned integer storage approach would require an $O(\log(n))$ word size. Furthermore, n also impacts the size of each processing element, which would have to add and compare scores with comparators and adders of $O(\log(n))$ bit width.

A more efficient storage approach is noted in [15]. Given the nature of the algorithm, it can be seen that the difference in scores of adjacent cells will not be very large (i.e. only low order bits will vary between neighbours) and therefore a modulo encoding scheme may be used. Such a scheme would work very well if the parameters of the algorithm – which determine the magnitude to which neighbouring scores may vary – are small values. Fortunately, this is typically the case for SW usage in molecular biology. By setting the SW parameters accordingly, it is possible to adopt a modulo 4 $O(1)$ encoding scheme, and therefore limit the word size for score storage and adding/comparing to a constant 2 bits for any sequence length N. Some implementations reduce this to a single bit, however these implementations are very rigid in terms of SW parameters. This approach is adopted in [5][6][8][12] and [14].

Some implementations, such as [5] and [6], utilize run-time-configuration to generate more efficient hardware. Some constants that are determined at run-time (i.e. parameters of the algorithm) are embedded into the logic in order to obtain customized constant adder circuits. In addition, by embedding one of the query sequences in the logic, customized comparators may be obtained. These efforts result in smaller and faster processing elements. Specifically, in terms of look-up-tables (LUTs) and flip-flop pairs, savings of approximately a factor of 5.5 have been attributed to run-time reconfiguration [6].

One of the most successful implementations [5] utilizes an FPGA board with a Xilinx XC2V6000-4 device and multiple memories attached. Ten SRAM chips are attached, in addition to 512MB PC133 SDRAM (accessible by the host computer), as well as some RAM that is dedicated for run-time-configuration data. The board communicates with the host PC via 64-bit 66MHz PCI, which is quite a restrictive interface, however the interface is only needed to transfer query data and FPGA configuration data to the CCM board, after which the computation is contained within the board. This configuration, with an implementation that exploits run-time-configuration, holds 7k processing elements and is capable of 1260 billion cell updates per second (CUPS). In practical terms, this solution is capable of aligning a 47MB database in less than 0.44s with linear scalability.

Other implementations have shown similarly impressive results. Implementations that use run-time reconfiguration are especially powerful, with the implementation presented in [6] providing 757 billion CUPS with a Xilinx XCV1000-6 FPGA, which holds 4k processing elements. An XC2V6000-5 FPGA on the same implementation would be capable of holding 11k processing elements and provide 3.2 trillion CUPS. Implementations that do not use run-time reconfiguration also provide very high performance, such as presented in [14] which uses a Xilinx XCV1000-6 to hold 4032 processing elements (where this number was specifically chosen for floorplanning purposes) to achieve 742 billion CUPS.

B. BLAST

BLAST consists of far more complex control paths compared to SW. Therefore, a simple and obvious CCM design to execute BLAST in it's entirety is not apparent. One attempt at a BLAST acceleration CCM [16] began by profiling the BLAST program (using the Unix gprof tool) to identify bottlenecks in the program, which may turn out to be good candidates for CCM based acceleration. This is a common first step in many cases where a CCM implementation to accelerate some problem is not obvious, and may be performed with other program profiling tools as well, such as Apple Computer's Shark tool which was used in [23].

A code segment of BLAST that accounted for 75% of the execution time was identified in [16]. This code segment performs lookup table indexing and referencing. Clearly, this is a far different computational kernel than sequential SW – where sequential SW performance is compute-bound, the performance of the kernel of the BLAST algorithm is IO-bound. The fact that the profiling tool indicated this kernel as being the bottleneck in the algorithm would therefore reflect limitations of the memory system rather than the processing engine, and implementing this kernel alone in a CCM with the rest of the algorithm being run on a host machine is unlikely to provide any reasonable performance gains. To further aggravate the situation, the communication with a host system in [16] was severely limited, as a 33MHz PCI interface was used.

Nevertheless, such a CCM was implemented. The kernel is implemented in hardware with two state machines. The first state machine – the input side state machine – reads query character data and generates LUT index addresses (with the LUT stored in off-chip SRAM), as well as writing some related data into a FIFO. The second state machine – the output side state machine – retrieves data from the LUT and pops data off the FIFO. It was found that the performance of the system was significantly slower than a pure software approach, and this performance deficiency was attributed to communication overheads between the CCM and the host system.

A more successful implementation is presented in [17], where it was was shown that 4 instances of the described CCM perform 10 times faster than a 128 node 667MHz CPU cluster (and a 40 times improvement is claimed as a price-performance ratio). This implementation makes similar observations regarding the bottleneck in the system as in [16], however far more effort is put into optimizing memory efficiency. State machines are implemented in [17] which are similar to [16], however in [17] the entire BLAST algorithm is implemented as a CCM. The fundemental difference between the two approaches [17] and [16] is that a very high throughput low-latency memory subsystem is provided in [17]. Each FPGA in the system has 4 independent DRAM channels, providing 12.8GBps aggregate peak memory bandwidth. In contrast, the resources made available to the CCM in [16] were relatively deprived, which resulted in an inordinate amount of communication with the host system, the expense of which was further compounded by a very meager interface.

A BLAST acceleration CCM is also commercially available [3], however details of this implementation are not publicly available. Nevertheless, performance numbers for the system have been released, and impressive claims have been made, such as a 216 hour job on an 8 CPU cluster taking 16 minutes on a DecypherBLAST engine.

C. Hidden Markov Models

The computationally expensive kernel in HMM sequence analysis lies in the Viterbi algorithm, a dynamic programming algorithm which scores and aligns sequences relative to a HMM of sequences in a database. A higher score would indicate that the query sequence is more probabilistically related to the other sequences in the database for the corresponding model. Biologically motivated HMM CCMs are not as common as SW or BLAST CCMs. Commercial implementations are available, such as from [3]. Very impressive performance claims have been made about the DeCypher accelerator; HMM analysis of approximately 1000 models takes approximately 9 hours on a particular computer system, and when the very same computation is run on the very same computer system but with two DeCypher cards attached the computation takes just 3 minutes.

One biologically motivated HMM acceleration CCM design is presented in [18]. The system consists of several parts, though the kernel of the system lies in the scoring block, wherein lies the Viterbi algorithm. The implementation of this stage is very similar to SW, i.e. it is basically a systolic array scoring machine, where scores in this context are actually probabilities. The algorithm requires multiplication of probabilities, which are expensive in terms of computation but also complicate implementation details, as multiplication of small probabilities produce very small numbers that can lead to underflow errors. To avoid these problems, the algorithm is performed in log space, which would replace multipliers with adders, and reduce the bit-width required to express the range of probabilities. In spite of this, optimizations like a modulo-encoding approach to bit-width limiting are not obvious, and therefore implementations may be expensive in terms of logic usage. Unfortunately, performance figures for the system implemented in [18] were not available.

D. Other Applications

Besides molecular biocomputation, FPGAs have been successfully used to accelerate other domains of computation. FPGAs are an excellent medium for the implementation of cryptogtaphy algorithms, as these algorithms are inherently parallel and consist of bit-level operations, therefore cryptography is one area that benefits greatly from custom computation [2]. Content addressable memories are used to deploy high capacity Intrusion Detection Systems [20]. Artificial neural networks, which are massively parallel by nature, have long been successfully implemented with FPGAs [21]. There are many implementations of Java virtual machines on FPGAs, but furthermore the parallelism and the freedom to use as much FPGA space as is available allows the implementation of multiple Java virtual machine execution paths in order to provide multiprocessor capability to Java threads [22]; the number of execution paths may be adjusted to fit the size of the FPGA device. This line of thinking may even be applied to VLIW processor designs [23]. Investigative use of FPGAs has been undertaken on structural mechanics computations [24]. Many diverse disciplines with computationally intensive workflows are either deploying FPGAs as an integral part of their work, or at least investigating the usage of such technology.

The approaches mentioned above involve very application specific designs, but a middle ground exists between highly specific architectures and mainstream general purpose processors. Very often an existing processor architecture is adopted for a certain application, but is augmented in some way to better suit the application, for example by adding a single bit conditional operation to the instruction set of a traditional microprocessor design [25]. Alternatively, microprocessors may be tightly coupled to external IP blocks. Another approach may be to implement one stage of an application as a custom design (e.g.: color thresholding) and another stage may be implemented as software running on a standard or modified RISC processor (e.g.: noise reduction code) with the two stages communicating through shared memory [25]. These design scenarios occur very frequently, and it is therefore not surprising that CPU softcores such as Altera's Nios and Xilinx's MicroBlaze are so commercially successful.

IV. Future Developments

The two largest stumbling blocks for wide spread adoption of FPGA solutions across a wide range of application domains are integration concerns, and development tools. Integration limitations have the potential to severely limit the impact CCMs could have on the overall performance of some applications. Also, in order for custom computation to be embraced by the larger computing community, new programming methodologies that are less hardware-centric and more algorithm-centric must emerge. Reconfigurable device technology itself will not remain static; increasing logic resources and speed will allow greater acceleration of demanding workloads. Specifically within the computational molecular biology domain, there is much work that may be done to improve performance and functionality of CCMs.

A. Integration

The performance gains in using a CCM may be lost if a large amount of time is required to transfer data between the CCM and a host machine and/or other peer computing units, as was the case in [16]. Therefore it is imperative that high bandwidth, low latency interconnects be used to integrate CCMs with host machines or other systems.

PCI, the de-facto commodity interface, is a relatively low bandwidth, high latency shared bus, and is therefore generally unsuitable for HPC acceleration. However, if a CCM is capable of many orders of magnitude performance increase over software running on a high-end CPU with minimal communication with a host processor and/or other external device other than for initial data loading – as is often the case in molecular biocomputation algorithms – even PCI solutions yield massive performance gains [3][5]. Nevertheless, better integration infrastructure is required to facilitate the use of CCMs for a broader range of applications. Advances in commodity integration standards, such as PCI-Express and PCI-X, will yield important improvements (particularly in bandwidth) but will be limited by economic and compatibility requirements and would therefore not provide the ideal solution. For high-end HPC systems proprietary interfaces capable of maximizing the utilization of CCMs are provided.

Silicon Graphics addresses this need [26] in the Altix line of HPC systems with a proprietary interconnect fabric – the NUMALink interconnect (12.8GBps) – designed for system-wide general purpose high performance scalability. Coupled with a reconfigurable computing module which attaches to the NUMALink fabric through a low latency Scalable Systems Port (6.4GB/s), this solution provides high performance (i.e. high bandwidth and low-latency) as well as scalability and flexibility.

Cray's XD-1 HPC systems [27] feature 6 FPGAs directly connected to AMD Opteron CPUs thru the 3.2GBps RapidArray interconnect; this very tight coupling should result in very low latency. 16MB caches are provided to FPGAs at 12.8GB/s. While this solution doesn't provide a tremendous amount of flexibility, very tight coupling (and the resultant low-latency) would allow for superlinear speedup of many applications including applications with high communication requirements with the host processor. Furthermore, run-time reconfiguration where a CCM is broken into a series of blocks and deployed consecutively on limited FPGA resources is highly practical in such a system, and API hooks are provided to support these activities at the application level.

B. Programming

While the idea of being able to take existing sequential C code meant for software compilation and compiling it instead for hardware deployment – often dubbed behavioral synthesis – is a very tempting idea, it remains to be seen if this can be done in a way that generates efficient hardware for a wide set of applications. Extracting optimal parallelism from software code is regarded as a computationally intractable problem, though it may still be possible to obtain a sub-optimal yet sufficient level of parallelism. Efforts to do so have enjoyed limited success, such as [23][28][29] and [30] which extract parallelism by means of an intermediary dataflow graph, however such an approach would only produce good quality results for certain sets of problems.

VHDL and Verilog have long been the two enduring hardware description languages – traditionally used predominantly for hardware documentation and simulation [31], but in recent years used increasingly for hardware synthesis as well. These HDLs afford the programmer very fine grained control over the hardware that is synthesized. However, just as software programmers moved away from assembly language and toward higher generation languages in order to boost productivity, so too shall hardware engineers move toward HDLs that are more descriptive of algorithms and procedural behavior rather than hardware function units [32], and this is true both for reconfigurable hardware users as well as ASIC designers.

The move to C-like HDLs would only be reasonable if this new breed of HDLs allows the production of designs that exhibit reasonably good quality of result (QoR). In order to achieve a good QoR, algorithm implementations must be parallelized, but furthermore designs must give due consideration to timing factors. Such concepts are relatively alien to C-like languages, but critical to the production of good hardware designs. Furthermore, software programming languages afford very little granularity in terms of variable and data path sizes – this limitation cannot be present in HDLs, since hardware designs can make use of arbitrary bus widths and register sizes.

It seems therefore that C-like languages may be adopted for their familiarity and algorithmic efficiency, but that constructs must be provided that aid the production of high QoR hardware designs. To this end languages such as Handel-C [33] and Impulse C [34] provide language (including preprocessor) constructs and semantics to explicitly enforce parallelism, build arbitrary width registers and datapaths, enforce pipelining, etc. SystemC [35] achieves the same ends by providing hardware constructs within class libraries. While this puts some extra burden on programmers to work such concepts into their designs, this is presently necessary for the construction of reasonable QoR hardware designs. Furthermore, just as software programmers often implement some critical kernel of their software in assembly language, very often (especially while next generation HDLs remain in their infant-to-adolescent years) designers will need to implement critical aspects of a hardware design in a traditional HDL.

Example Handel-C code segments may illustrate some constructs and semantics that were added to standard C in order to allow for efficient hardware synthesis. Parallelization may be explicitly declared using a '$par\{\}$' block:

```
par{
    a=b;
    b=a;
}
```

In traditional sequential programming, the effect of the inner block of the above code would be that both storage elements a and b will take the value of b. However, the above instructions execute in parallel, and consequently the effect is that the contents of a and b are swapped. This is completed in a single cycle because parallel hardware allows both operations to proceed simultaneously (with flip-flop writes typically occurring at the end of a clock cycle, thus avoiding a race condition).

Efficient hardware designs also involve arbitrary width data buses and arbitrary width register banks. Therefore, it is necessary to provide mechanisms to select subsets of data lines, and also to merge datalines. The append operator '@' allows for arbitrary buses to be constructed from smaller buses:

```
bigbus = bus1 @ bus2 @ otherbigbus[31:27];
```

In order to select a subset of datalines in a bus, the bit selection operator '$[x : y]$' may be used. In the above example, bits 31-27 of $otherbigbus$ are appended to all the bits on $bus2$, and this larger bus is then appended to all the bits in $bus1$, thus forming $bigbus$. These are just some of the more basic facilities that must be provided by any high-level HDL.

Where an application may be decomposed into a set of regular discrete functional units, a hardware implementation could be produced by simply dropping functional units into the design and linking them together appropriately. This approach may yield good results as each functional unit may be obtained from a highly optimized library of functional units – therefore each individual element in the design is itself a very high quality design. The task of linking each unit together to implement some algorithm is left to the designer. This is the approach taken with the Viva [36] development environment. A graphical drag-and-drop interface is provided to express dataflow thru various functional units, and allows rapid development of high performance custom solutions for many applications such as [13] and [24].

While next generation HDLs promise greater productivity for hardware designers, the present day reality may not reflect this claim yet. Choosing a design strategy for any project is dictated in large part by the availability of tools to get the job done. While C-like HDLs may grant a designer the ability to rapidly express a design, there are many more tools required to get a design onto an ASIC or FPGA. This includes tools for synthesis, place and route, timing analysis, visualization and debugging, stimulus generation and response-checking, simulation, and much more [31]. The electronics design automation (EDA) industry has suites of mature and tested tools for Verilog and/or VHDL designs, and these tools allow for the detection and removal of potential problems in designs very early in the design phase [31]. Also, EDA tools for traditional hardware design methodologies are mature enough that synthesis and other tools have reached a level of sophistication that provides a very high QoR for generated hardware. Nevertheless, where aspects of these tools need to be rebuilt to accommodate new HDLs, new and/or updated tools will likely emerge to support the new breed of HDLs. The potential productivity gains far outweigh the costs of adapting new design methodologies and tools.

Partial-reconfigurability is a feature present in many current generation FPGAs, and as previously discussed this allows for run-time-reconfiguration optimizations. It is conceivable that such approaches may allow for efficient computation of some computationally irregular programs, something that is traditionally regarded as unsuitable for CCM deployment due to the complex control paths, though whether or not this may be feasibly done for a sufficiently large subset of irregular problems remains to be seen. However, to some extent, awareness of run-time-reconfiguration capability may be built into HDLs to automatically exploit this capability. Such capability would also allow the deployment of large hardware blocks into limited configurable logic resources on smaller FPGAs that support run-time-reconfiguration.

C. Reconfigurable Devices

While FPGAs are commonly thought of as a "sea of gates", the devices are not necessarily homogenous in nature. It is common to include specialized circuitry within FPGAs to provide better performance for some applications. Specifically, hardware multipliers are often included so that FPGAs may be used for high performance DSP applications.

Floating point circuitry consumes too much space on today's FPGAs to be regarded as a reasonable solution, however as FPGAs continue to grow in size this limitation continues to diminish. Furthermore, it has been argued that FPGAs are improving in terms of floating-point performance at a greater pace than contemporary VLSI CPU designs [37].

D. CCMs in Molecular Biology

Specifically within the domain of CCMs for computational molecular biology, there is much room for further work. This includes enhancing existing SW implementations by providing more feature-filled solutions, and providing more alternatives for BLAST and especially HMM CCMs. This paper has only touched on the sequence analysis aspect of molecular biology; the structural analysis aspect, the most famous facet of which is protein folding, may also greatly benefit from CCM-based acceleration.

1) SW CCMs: There has been a lot of success in implementing basic SW CCMs that perform very well, and therefore it would be practical to extend these designs and build more sophisticated SW CCMs. Some possible extensions include a run-time variable module encoding scheme, that can adjust the widths of functional elements and storage words to match the requirements of the input parameters, instead of dictating a set of limited parameters that may be used. Also, instead of simplistically computing matches between characters of query sequences as identical match or no match, it would be useful to allow for a gradient of scores for characters that may be more or less matched, and similarly to allow wildcards. This would, however, complicate the implemented hardware substantially as a larger number of low-order bits would be required to encode scores if a more fine-grained level of scoring is to be allowed, which would consume far more logic and storage resources. Nevertheless, such a system may be sufficiently more useful in molecular biology to justify the hardware costs. As FPGA device sizes continue to grow, interconnect performance continues to improve, and storage densities increase, processing elements of rising complexity become less undesirable, provided the extra complexity provides extra functionality and/or performance.

2) Other sequence alignment: Thus far the bulk of CCMs in the molecular biology domain have been SW implementations. This is understandable, given that SW can be implemented as a CCM very cleanly and efficiently, and also because the performance gains over conventional general purpose systems are often very dramatic. However, other algorithms are also commonly used in molecular biocomputing. BLAST is frequently used as an alternative to SW for performance reasons; since a CCM implementation of SW yields linear time performance, this tempts us to conclude that BLAST is no longer required, since SW is a more rigorous algorithm than BLAST. However, it may be the case that while BLAST misses some matches that SW picks up, these distant matches may not be biologically relevant. Furthermore, a BLAST CCM may yield far greater sequence alignment capacity than a SW CCM – a thorough consideration of CCM approaches to both algorithms, especially with closely comparable benchmarks of typical molecular biology workflows, would be valuable. Hidden Markov Models provide some unique functionality in computational molecular biology that aren't immediately provided by BLAST or SW (such as multiple alignment) and therefore more effort into HMM CCMs may also prove to be worthwhile.

3) Structural analysis: Protein folding simulation is very computationally expensive, as it involves modeling interatomic interactions at various levels of sophistication – ranging from modeling each amino acid residue as a point particle in a lattice structure, to quantum mechanical methods that model electronic wave functions [38] – on a very fine time-scale. It has been crudely estimated that in order to simulate 100 microseconds of actual protein folding requires computation in the order of 10^{23} machine instructions – a computer capable of petaflops per second would take over 3 years to perform this computation [38]. Critical to the acceleration of such workflows is the level of parallelism that may be extracted. In [38] three broad approaches to parallelization are mentioned:

1) Each particle is bound to a hardware thread.
2) Each force term is bound to a hardware thread.
3) The problem space is divided spatially, with all the forces on particles within a volume unit bound to a hardware thread.

Each of these approaches has relative merits and applicability for different situations, as discussed in [38]. The success of the Folding@Home project suggests that such computations are parallelizable to a significant extent, as super-linear speedup has been achieved with the ensemble dynamics method for some cases, and near-linear speedup is achieved in others [39]. The kernel of such computations consists of summation of force terms, with each term relying on information from two to four neighboring particles for different kinds of interactions such as torsion, bending, etc. [38] – in a CCM implementation, each term may be computed in parallel, with data exchanged between neighboring processing elements. It should be determined if such a system could feasibly be deployed on a FPGA.

4) Performance comparison: While this paper has attempted to indicate the relative performance improvements yielded by various parallelization methods (i.e. SIMD vs cluster vs CCM), this has been done with trepidation because the performance numbers available in the literature are not directly comparable. Some work into thoroughly benchmarking different computation systems – i.e. single processor (or SMP) workstations utilizing SIMD functions vs large scale cluster (with each node also utilizing SIMD functions) vs CCM – would be worthwhile, for such an exercise would lead to better educated decisions regarding which system would suit a particular workload (and budget) best.

V. Conclusion

By utilizing run-time-reconfiguration based on run-time parameters and data, it is possible to construct simpler and more efficient hardware, and this may lead to tangible performance benefits because the hardware constructed will likely perform better in terms of propagation delay and other delays as opposed to slightly more generic hardware, and furthermore as the hardware produced will be smaller it would be possible to fit more hardware on any given FPGA device. In the case of SW CCMs, implementations that utilize run-time-reconfiguration perform significantly better than fixed implementations. This is because processing elements are simpler, and therefore run quicker. In addition, more processing elements may be deployed on finite FPGA logic resources.

There is the perception that FPGAs themselves are merely the current generation realization of reconfigurable computing – that "FPGAs are still only a first-generation embodiment of the big idea of a general-purpose, reconfigurable substrate for special-purpose computing" [32]. While it remains to be seen what shape the next generation of reconfigurable computing architectures will take, the current generation of reconfigurable computing architectures are evolving both in terms of device size and configuration (for instance, with the inclusion of on-chip dedicated circuitry) but more importantly tools and interconnects to better utilize reconfigurable computing are growing in availability and sophistication.

CCMs are a viable solution for high performance computing providing dramatic performance gains across many domains, with molecular biocomputation being one such domain. Where biocomputation workflows typically included days or weeks of HPC server time to align sequences, CCMs can provide relatively instantaneous access to alignment information.

Reconfigurable architectures combined with conventional CPU architectures provide a platform for computational efficiency and elegance combined with the practicality of a general purpose system, and such systems are likely to flourish as development and integration overheads for such solutions continue to decline.

References

[1] Seismic Processing with Tricons Tsunami Suite Accelerated by Starbridge Hypercomputing, *Starbridge: The Hypercomputing Company*, [online], http://www.starbridgesystems.com/resources/whitepapers2.html (Accessed: 10 May 2005).

[2] Mitrion Applied, *Mitrion – Unparalleled Computing*, [online], http://www.mitrion.com/applied.shtml (Accessed: 10 May 2005).

[3] DeCypher Performance, *TimeLogic Home* [online], http://www.timelogic.com/performance/index.html (Accessed: 10 May 2005).

[4] A. Rudra, "FPGA-based applications for software radio", RFDesign, pp. 24-35, May 2004.

[5] K. Puttegowda, W. Worek, N. Pappas, A. Danapani and P. Athanas, "A run-time reconfigurable system for gene-sequence searching", in *Proceedings of the 16th International Conference on VLSI Design*, 2003, pp. 561-566.

[6] S. Guccione, E. Keller, "Gene Matching Using Jbits", in *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, 2002, pp. 1168-1171.

[7] K. Puttegowda, "Context Switching Strategies in a Run-Time Reconfigurable System", M.S. Thesis, Virginia Polytechnic Institute and State University, Blacksburn, Virginia, USA, 2002.

[8] Y. Yamaguchi and T. Maruyama, "High Speed Homology Search with FPGAs", *IPSJ Transactions on High Performance Computing Systems*, vol. 43, pp. 196-205, 2002.

[9] R. Durbin, S. Eddy, A. Krogh and G. Mitchison, "Pairwise alignment", in *Biological sequence analysis: probabalistic models of proteins and nucleic acids*, Cambridge University Press, 1998, pp. 22-23.

[10] W. S. Martins, J. B. Del Cuvillo, F. J. Useche, K. B. Theobald and G. R. Gao, "A multithreaded parallel implementation of a dynamic programming algorithm for sequence analysis", in *Pacific Symposium on Biocomputing 2001*, 2001, pp 311-322.

[11] T. Rognes and E. Seeberg, "Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors", *Bioinformatics*, vol. 16, pp. 699-706, 2000.

[12] D. T. Hoang, "A Systolic Array for the Sequence Alignment Problem", Brown University, Providence, Rhode Island, Technical Report CS-92-22, 1992.

[13] A Reconfigurable Computing Model for Biological Research Application of Smith Waterman Analysis to Bacterial Genomes, *Starbridge: The Hypercomputing Company*, [online], http://www.starbridgesystems.com/resources/whitepapers1.html (Accessed: 10 May 2005).

[14] C. W. Yu, K. H. Kwong, K. H. Lee and P. H. W. Leong, "A Smith-Waterman Systolic Cell", in *Proceedings of the 10th International Workshop on Field Programmable Logic and Applications*, 2003, pp. 375-384.

[15] R. J. Lipton and D. Lopresti, "A Systolic Array for Rapid String Comparison", in *Proceedings of the Chapel Hill Conference on VLSI*, 1985, pp. 363-376.

[16] K. Muriki, K. D. Underwood and R. Sass, "RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation", in *Proceedings of the Fourth International Workshop on High Performance Computational Biology*, 2005.

[17] C. Cheng, "BLAST Implementation on BEE2", University of California, Berkeley, Technical Report, 2004.

[18] S. Gupta, "Hardware Acceleration of Hidden Markov Models for Bioinformatics Applications", M.S. Thesis, Boise State University, Idaho, 2004.

[19] S. J. Melnikoff, S. F. Quigley and M. J. Russell, "Implementing a Simple Continuous Speech Recognition System on an FPGA", in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[20] L. Bu, J. A. Chandy,"FPGA Based Network Intrusion Detection using Content Addressable Memories", in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 316-317.

[21] J. Zhu and P. Sutton, "FPGA Implementation of Neural Networks – a Survey of a Decade of Progress", in *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications*, 2003, pp. 1062-1066.

[22] J. Parnis and G. Lee, "Exploiting FPGA Concurrency to Enhance JVM Performance", in *Proceedings of the 27th conference on Australasian computer science*, 2004, pp. 223-232.

[23] A. K. Jones, R. Hoare and D. Kusic, "An FPGA-based VLIW Processor with Custom Hardware Execution", in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005, pp. 107-117.

[24] R. Singleterry, J. Sobieski and S. Brown, "Field-Programmable Gate Array Computer in Structural Analysis: An Initial Exploration", in *43rd American Institute of Aeronautics and Astronautics (AIAA) Structures, Structural Dynamics, and Materials Conference*, 2002.

[25] T. Ramdas, L. Ang and G. Egan, "FPGA Implementation of an Integer MIPS Processor in Handel C and it's Application to Human Face Detection", in *Proceedings of TENCON 2004*, 2004, pp. 36-39.

[26] Extraordinary Acceleration of Workflows with Reconfigurable Application-specific Computing from SGI, *White paper*, Silicon Graphics Inc., 2004.

[27] Application Acceleration with FPGA-Based Reconfigurable Computing, (Cray Inc. - The Supercomputer Company), [online] 2005,

http://www.cray.com/products/xd1/acceleration.html (Accessed: 10 May 2005).

[28] T. J. Callahan, “Automatic Compilation of C for Hybrid Reconfigurable Architectures”, Ph.D. dissertation, University of California, Berkeley, California, USA, 2002.

[29] M. Budiu, S. Copen. Goldstein, “Pegasus: An Efficient Intermediate Representation”, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Technical Report CMU-CS-02-107, 2002.

[30] Catapult C Synthesis, *Datasheet*, Mentor Graphics Corporation, 2005.

[31] J. L. Lee, “Back to the language roots”, (Embedded.com), [online] December 2004, http://www.embedded.com/showArticle.jhtml?articleID=55801140 (Accessed: 10 May 2005).

[32] I. Page, “Compiling software to gates”, (Embedded.com), [online] December 2004, http://www.embedded.com/showArticle.jhtml?articleID=55801142 (Accessed: 10 May 2005).

[33] Handel-C For Hardware Design, *White paper*, Celoxica Limited, August 2002.

[34] From C to FPGA, (C Programming Tools for FPGA Platforms), [online], http://www.impulsec.com/C_to_fpga.htm (Accessed: 10 May 2005).

[35] S. Swan, “An Introduction to System Level Modeling in SystemC 2.0”, *White paper*, Cadence Design Systems Inc., May 2001.

[36] Viva Software – A Graphical Programming Environment For FPGAs, (Starbridge: The Hypercomputing Company), [online] 2004, http://www.starbridgesystems.com/products/vivatour.html (Accessed: 10 May 2005).

[37] K. Underwood, “FPGAs vs. CPUs: Trends in Peak Floating-Point Performance”, in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, 2004, pp. 171-180.

[38] F. Allen et al., “Blue Gene: A vision for protein science using a petaflop supercomputer”, *IBM Systems Journal*, vol. 40, pp. 310-327, 2001.

[39] S. M. Larson, C. D. Snow, M. Shirts and V. S. Pande, “Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology”, to appear in *Computational Genomics*, R. Grant, editor, Horizon Press, 2002.