

Optimized Smith-Waterman Implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer

Jeff Allred, Jack Coyne
William Lynch and Vincent Natoli
Stone Ridge Technology
2107 Laurel Bush Road
Bel Air, MD 21015
Email: jallred@stoneridgetechnology.com

Joseph Grecco
Intel Corporation
77 Reed Road
Hudson, MA 01749
Email: joe.grecco@intel.com

Joel Morrisette
Intel Corporation
5300 NE Elam Young Parkway
Hillsboro, OR 97124
Email: joel.morrisette@intel.com

Abstract—The Smith-Waterman algorithm is employed in the field of Bioinformatics to find optimal local alignments of two DNA or protein sequences. It is a classic example of a dynamic programming algorithm. Because it is highly parallel both spatially and temporally and because the fundamental data structure is compact, Smith-Waterman lends itself very well to operation on an FPGA. Here we demonstrate an implementation of this important algorithm in a novel FSB module using the Intel Accelerator Abstraction Layer (AAL), a newly released software middleware layer which abstracts away hardware idiosyncrasies and allows application developers to program to a common and predictable interface. We have modified SSEARCH35, an industry standard open-source implementation of the Smith-Waterman algorithm, to transparently introduce a hardware accelerated option to users. We demonstrate performance of nine billion cell updates per second and discuss further opportunities for performance improvement.

I. INTRODUCTION

In recent years a number of competing technologies for software acceleration have emerged. While it is still unclear what sustained role they will hold on future computing platforms it is clear that there are important applications that present an opportunity for acceleration on non-traditional platforms. Reconfigurable hardware such as FPGAs, for example, has been made available to users by several High Performance Computing (HPC) hardware vendors and used effectively to accelerate a wide variety of important algorithms. Among the most recent tools available for reconfigurable computing are socket replacement modules, FPGA platforms that substitute directly for a CPU in-socket and middleware software abstraction layers.

In this work we address the Smith-Waterman algorithm[3], a highly parallel Bioinformatic algorithm that lends itself well to implementation on FPGAs. The hardware platform targeted is the XtremeData XD2000i FSB module. Using AAL, we have integrated our hardware accelerated Smith-Waterman function into SSEARCH35, an open source industry standard. In section II we describe the target hardware platform in more detail. In section III we discuss the details of the Smith-Waterman algorithm. In section IV we discuss our particular

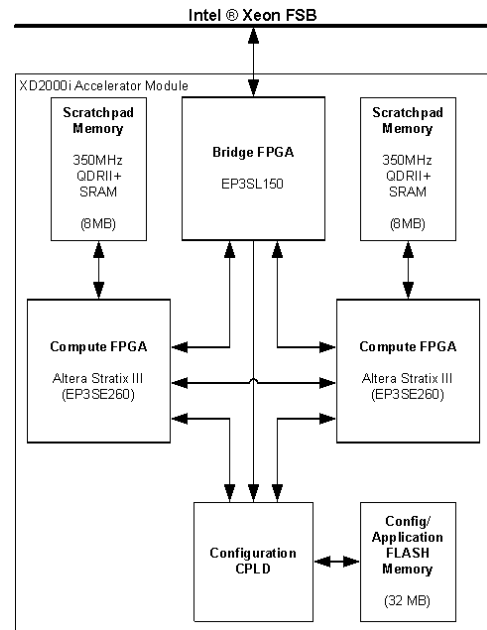


Fig. 1. Block Functional Diagram of Xtreme Data XD2000i Module

hardware design and implementation. Section V discusses the integration of our hardware algorithm into SSEARCH35, a standard Smith Waterman software application. Sections VI discusses our results and conclusions.

II. XTREMEDATA XD2000i FSB MODULE

The XtremeData, Inc. XD2000i hardware accelerator is a direct hardware replacement for the Intel[®] Xeon[®] micro-processor in dual-socket server and blade systems. Plugging directly into the Xeon Front Side Bus (FSB) via one of the system's CPU sockets, the XD2000i provides three Altera[®] EP3SE260 Field Programmable Gate Arrays (FPGAs) in a tiered topology, each providing 254,000 equivalent logic elements (LE). The "bridge" FPGA abstracts the FSB through an encrypted core, providing 8.5GB/s FSB access and two

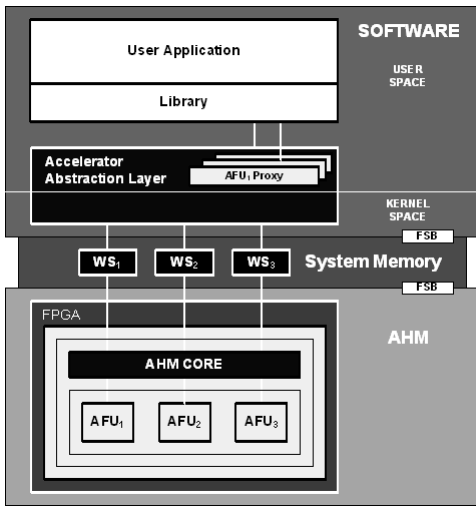


Fig. 2. Intel Accelerator Abstraction Layer software stack

local 9.6GB/s dedicated ports to the secondary "compute" FPGAs. The compute FPGAs are configured at runtime with user-defined logic to provide hardware-assisted acceleration as required by the host applications running on the Xeon® processor(s) in the system. Each compute FPGA is provided with 8MB of 350MHz QDRII+ RAM scratchpad memory. A block diagram of the FSB module appears in Fig 1.

III. INTEL ACCELERATOR ABSTRACTION LAYER (AAL)

To date, a coherent vendor supported middleware layer which eases the transition to accelerating platforms has been lacking. The Intel® QuickAssist Technology and Accelerator Abstraction Layer[1], [2], have been introduced to fill that gap. Intel® QuickAssist Technology is a comprehensive initiative that consists of a family of interrelated Intel® and industry standard technologies that enable optimized use and deployment of accelerators on Intel® platforms. QuickAssist facilitates the integration of hardware accelerators into Intel® platforms by defining a common usage paradigm through the Accelerator Abstraction Layer (AAL). AAL defines a common software framework (Service APIs) and primitives for accessing QuickAssist accelerators that have been discovered and registered via the Accelerator Abstraction Services (AAS). Each accelerator is accessed via its existing interconnect interfaces through Accelerator Interface Adaptors (AIAs), which provide a standardized and hardware-agnostic interface to the underlying Accelerator Function Units (AFUs) implemented in hardware. In this manner, AAL provides a single programming model that supports multiple simultaneous applications on multiple compute cores. Figure two presents a block diagram of the ALL software stack.

By presenting a uniform discovery mechanism and messaging interface, AAL provides a migration path across multiple families and generations of accelerator implementations with minimal additional development effort. In this way, AAL is particularly well suited to asynchronous and massively parallel applications like the kind found in Smith-Waterman, enabling

	A	G	C	T	A	C	G	T	...
T	0	0	0	0	0	0	0	0	
A	0	5	4	3					
C	0	4	3						
T	0	3							
T	0								
T	0								
A	0								
G	0								
...									

Fig. 3. Smith-Waterman Similarity Matrix

the developer to realize linear performance gains with the integration of additional accelerator resources.

AAL consists of a collection of kernel and user mode runtime software services, libraries and APIs that sit between applications and accelerator modules (e.g., such as FPGA-based accelerator modules) deployed on Intel® processor-based platforms (See Figure 2).

AALs platform runtime services provide accelerator virtualization which abstracts the accelerators physical implementation technology from the application. Accelerators may be implemented in a variety of ways and connected using any number of technologies. This abstraction enables applications to be portable from one family of accelerator to another. Because applications do not directly control accelerator resources, AAL is able to share the physical accelerators among multiple applications transparently. Using AALs ability to dynamically reprogram the shared accelerators transparently to the application, the accelerator resources are more efficiently utilized on the platform by re-provisioning them during operation.

AAL implements a light-weight Service Oriented Architecture. Accelerator resources are exposed to the application as service objects. These service objects are a collection of hardware and software components abstracted through a single object called the Accelerator Function Unit(AFU), The application requests an AFU by providing the AAL resource factory service a manifest that describes the required capabilities. The AAL runtime services will locate accelerator resources, configure them if needed and return the resources as an AFU object. Libraries are provided that implement APIs to the technology specific message formatting and communications layers between the application and the accelerator

AFUs are implemented as hardware (digital logic) embedded in an FPGA, as well as the host side software that exposes it. Since certain algorithms run better on Intel® Architecture(IA) and others on purpose built accelerators like FPGAs, an AFU will typically implement parts of its algorithm on the IA host and parts on the dedicated accelerator HW. AAL allows the AFU developer to implement their algorithm on the

compute engine that best serves their needs.

IV. SMITH-WATERMAN ALGORITHM

The problem of sequence similarity is recurrent in the fields of Genetics and Bioinformatics. When a new gene is discovered its role and function may be inferred by its similarity to known sequences. The general problem is then to gain a measure of similarity between two separate sequences of 'letters' drawn from an 'alphabet'. In genetics the focus is usually on sequences of nucleic acids drawn from the set of four possibilities Adenine ('A'), Guanine ('G'), Cytosine ('C') and Thymine ('T') or AGCT for DNA or sequences of amino acids that make up proteins drawn from an alphabet of 20 possible amino acids. A typical problem will search for the best match of a query string inside a much larger database string. Methods for solving this problem with various additional boundary conditions are well known and drawn from the class of dynamic programming methods. Typical problem constraints are to find the longest common sub sequence, a full global alignment or the best local alignment. Smith-Waterman is a dynamic programming method which finds the best local alignment, typically between a shorter query string inside a longer database string. This contrasts with, for example, Needleman-Wunsch[5] which finds the best alignment of the whole query inside the database.

The problem statement is to find the best local alignment of the string Q within the string D , where the length of Q is n and the length of D is m . The strings may be represented as $Q_1Q_2Q_3\dots Q_n$ and $D_1D_2D_3\dots D_m$.

Thus we wish to find a subsequence inside of Q which maximizes a metric of similarity with some portion of the string D . The Smith-Waterman algorithm and its derivatives such as that of Gotoh[6] and Hirschberg[7] solve this problem by constructing a similarity matrix, the elements of which each hold a score that is calculated recursively. The score of any individual cell in the matrix, V_{ij} is a function of the scores of its neighbors to the top, left and top-left diagonal, in addition to D_j and Q_i . The recursion relation is given by:

$$V_{ij} = \max \begin{cases} 0 \\ V_{i-1,j-1} + \sigma(Q_i, D_j) \\ V_{i-1,j} + \sigma(Q_i, -) \\ V_{i,j-1} + \sigma(-, D_j) \end{cases}$$

The operation of the algorithm is best illustrated by considering the similarity matrix shown by example in Figure 3. The matrix is formed with the string D as the first row at the top of the matrix and the string Q as the left-most column. The left-top corner is not used so the matrix is aligned in such a way that the diagonal elements connect equivalently positioned characters in Q and D .

The scoring matrix σ determines the relative weighting of matches, mismatches, deletions and insertions. A sample scoring matrix for DNA sequencing is:

$$\sigma = \begin{pmatrix} 5 & -1 \\ -1 & -4 \end{pmatrix}$$

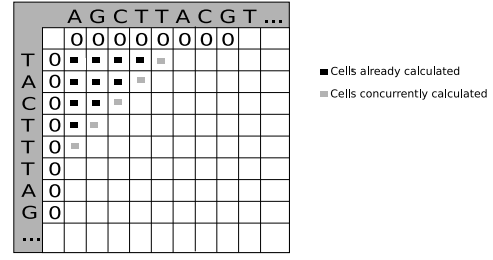


Fig. 4. Smith-Waterman Similarity Matrix. Black filled cells indicate scores already calculated. Gray cells indicate the concurrent computational front.

This scoring matrix increments the score by 5 for a match, decrements by 4 for a mismatch and penalizes insertions and deletions by 1. Figure 3 shows the score filled in for the first few cells in the upper left corner of a similarity matrix.

The basic Smith-Waterman algorithm described above penalizes the initiation and the extension of a gap equivalently. It is frequently the case that a larger penalty is applied to the initialization of a gap then to its extension. This affine gap model introduced by Gotoh[6] is accomplished by a relatively easy modification to the basic recursion relations by introducing two additional parameters, the left gap and the top gap. The new recursion relation for the affine gap model is given by:

$$V_{ij} = \max \begin{cases} 0 \\ L_{ij} \\ T_{ij} \\ V_{i-1,j-1} + \sigma(Q_i, D_j) \end{cases}$$

$$L_{ij} = \max \begin{cases} V_{i,j-1} - \alpha \\ L_{i,j-1} - \beta \end{cases}$$

$$T_{ij} = \max \begin{cases} V_{i-1,j} - \alpha \\ T_{i-1,j} - \beta \end{cases}$$

After completing the score calculation for the similarity matrix, the highest scoring cell is identified as the terminating point of the optimal alignment. To obtain the alignment itself one must backtrace from the high scoring cell to the point where the score becomes exactly zero. Backtracing requires each cell to store the information defining which of the three possible max values was chosen for the cell's score i.e. the information about which of the three possible neighbor cells originated its score value.

V. SMITH-WATERMAN FPGA IMPLEMENTATION

In this section we describe our approach to Smith-Waterman implementation in the FPGA. The design is first discussed in general terms and subsequently in terms of the specific implementation on the Altera Stratix III chips found on the XD2000i module.

The Smith-Waterman dynamic programming algorithm progresses a calculation front that is perpendicular to the diagonal of the sequence matrix (e.g. see Figure 4). It lends itself

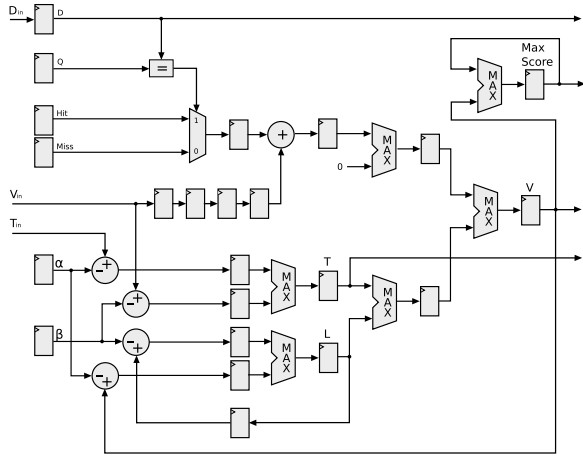


Fig. 5. Smith-Waterman Processing Element Block Diagram.

naturally to a systolic array with data streaming through element by element.

The algorithm calculates scores for every cell in a two-dimensional similarity matrix where each cell represents the intersection of a character in the query string, here placed as the vertical part of the array, and a character in the database string, here placed horizontally. The calculation of the score in any given cell is dependent on the scores of the cells that are neighbors in the left, up, and left/up diagonal direction. These data dependencies force an order of calculation for the cell scores.

A. Processing Element Design

An evaluation was made of the following three processing element organizations: (1) a processing element per cell, (2) a processing element for arbitrary sized, but much smaller arrays, and (3) a processing element for each row of the array. The processing element per cell suffers from a complex scheduling algorithm to assign processing elements to each individual cell, as well as complex element to element routing. The processing element for arbitrary sized arrays approach is a von Neumann architecture that relies too heavily upon block RAM resources. Using block RAM constrains the data paths to two independent paths only. It also creates a limitation on the number of processing elements based on the number of available block RAMs.

Each processing element of the third design approach represents one particular row of the matrix. In this organization, a processing element is responsible for the calculation of the scores in an entire row of the array. The calculation of a score is dependent on the scores above it (the row above) and to the left. This allows a march down the row, storing the result in a register for the next cell, and using the results from the processing element above. The results from above are used in order, so a FIFO allows easy transmission from one processing element to the next. Furthermore the processing element is a fixed algorithm and always requires the same number of clock cycles, the FIFO can be reduced to a register. In other

words the input and output rate of each processing element is fixed and equal, so there is no need for buffering of data. We avoid the dual port limitation described above due to this marching characteristic. This allows for the algorithm to be highly parallel and pipelined.

Organizing the machine with Row Based Processing Elements reduces the design from a two dimensional structure to a one dimensional structure. This eases scheduling and also makes it easier to flow from one FPGA to another. It creates a simple interface to the processor domain and it makes for simple place and route within a single fpga.

B. Datapath

Each processing element includes an input register that is fed from the output of the previous processing element. This input register includes a control field to specify the current action for the processing element. Based upon the control field the rest of this register contains the necessary data to perform the desired operation.

The predominant record type is a "Process Data" command. This record supplies the D, V, L and T from the previous row's machine. The D value is used to calculate whether a match occurred or not by comparing it to the Q value. When the current iteration is complete the D, V, L and T values will be passed on to the input register in the next processing element. There are three adders in the processing element. The Add Match/Miss Adder takes the diagonal from the previous row (via the V value) and adds either the match value or the miss value as calculated in the Q/D Match Logic. The Subtract gap penalty Adder calculates the score if a new gap is created by subtracting the New Gap Penalty from V_{left} and also from V_{up} . It must be subtracted in both the left and up direction to account for new gaps in either direction. The Extend Gap Penalty Adder behaves similarly, but subtracts the Extend Gap Penalty from L and then from T. This calculates the score potential from extending a previous gap in either direction. Finally, there are two MAX functions which compare the potential scores and return the larger of the scores. This datapath results in the last Max Function giving the score for the current cell. The current score is stored in V_{left} and put in the next processing elements V_{up} -value. The current gap based scores are put in L and the next processing elements T values.

The "Process Data" command takes four cycles to complete and is the longest of all the possible commands. All shorter commands were extended to four cycles to prevent machine starving and flooding. Although this seems inefficient to waste cycles by extending the machine pipeline, the "Process Data" command is the predominant command with all others occurring a negligible amount of the time. The benefit of extending the other commands to four pipeline stages is that with all commands having equal number of stages there is no need for a FIFO between stages. This dramatically reduces resource usage.

TABLE I
LOAD ALPHA/BETA COMMAND

Bits	Name	Description
[63:60]	Record Type : 0001	This field identifies the record as an Initialize Alpha and Beta Register Command
[59:52]	Reserved	Not used for this record type.
[51:32]	Alpha Value	This field holds the Alpha Scoring Value.
[31:20]	Reserved	Not used for this record type.
[19:0]	Beta Value	This field holds the Beta Scoring Value.

C. Control

The processing elements are controlled by self contained state machines. Every processing element can run independently of the others. We effectively created a four cycle machine for every possible input record to the processing element. Keeping every machine at four cycles allows us to be certain that the processing of the current input record will be complete by the time the next input record is ready from the previous machine. The control logic is then decoded based on the current input record type and the current clock cycle (1 of 4) of the machine. On each of the 4 states it decodes write enables based on the current state and the input record type.

The interface to the processing element is an inbound data path that feeds a FIFO residing in the core and an outbound data path that feeds a FIFO in the next adjacent core. The inbound and outbound data paths are identical and share the same encoded information. The data paths are 64 bits wide and have a write request signal for inputting data into the FIFO. Status flags are available to show when the FIFO is full and cannot hold any more data records. This data path is a direct map to the input register for each machine.

D. Smith-Waterman Word Encoding

The design includes five records that contain operational codes and data. These are described below and illustrated in Tables I-V.

1) *Initialize Alpha and Beta Registers*: This record supplies the scoring values Alpha and Beta. These values are used for scoring new gaps and gap extensions. The values are stored in the machine and the record is passed unchanged to the next core.

2) *Hit and Miss Registers*: This record supplies the scoring values Hit and Miss. These values are used for scoring matches and non-matches. The values are stored in the machine and the record is passed unchanged to the next core.

3) *Initialize T Register*: This record supplies query string value T to each machine. This value is stored locally and used for comparison to the database string value S that is sent with each Process Cell message. The value is stored in the machine and the record is passed unchanged to the next core.

TABLE II
LOAD HIT/MISS PENALTIES COMMAND

Bits	Name	Description
[63:60]	Record Type : 0011	This field identifies the record as an Initialize Hit and Miss Register Command
[59:52]	Reserved	Not used for this record type.
[51:32]	Hit Value	This field holds the Hit Scoring Value.
[31:20]	Reserved	Not used for this record type.
[19:0]	Miss Value	This field holds the Miss Scoring Value.

TABLE III
LOAD QUERY LETTER COMMAND

Bits	Name	Description
[63:60]	Record Type : 0010	This field identifies the record as an Initialize Q Register Command
[59:32]	Reserved	Not used for this record type.
[31:24]	Q Value	This field holds the Q value.
[23:5]	Reserved	Not used for this record type.
[4:0]	MachineNum	This field identifies the machine that these Q Values are directed at.

TABLE IV
LOAD INITIAL DIAGONAL SCORE COMMAND

Bits	Name	Description
[63:60]	Record Type : 1001	This field identifies the record as an Initialize V Diagonal Command.
[59:52]	Reserved	Not used for this record type.
[51:32]	V Value	This field holds the V Diagonal value.
[31:0]	Reserved	Not used for this record type.
[4:0]	MachineNum	This field identifies the machine that these Q Values are directed at.

4) *Initialize VDiagonal Register*: This record supplies the initial V Value in the upper left diagonal cell for the first row of the first machine. The value is stored in the machine and the record is passed unchanged to the next core.

5) *Process Cell*: This record instructs the core to process and score a single cell of the entire array. The record supplies the necessary data to calculate the cell score. That data consists of the score of the upper-left diagonal cell, the score of the upper cell, the current gap score of the upper cell, and the Q value, the search string value from the current column. The core will calculate the score for the current cell and then it

TABLE V
PROCESS CELL COMMAND

Bits	Name	Description
[63:60]	Record Type : 1000	This field identifies the record as an Process Cell Command.
[59:52]	Reserved	Not used for this record type.
[51:32]	V Up Value	This field holds the score of the cell directly above.
[31:30]	T Value	This field holds the columns T Value from the search string.
[29:20]	Reserved	Not used for this record type.
[19:0]	F Up Value	This field holds the current gap score of the cell directly above.

will create a new process cell record with the values it created and pass it to the next adjacent core.

VI. SMITH-WATERMAN SOFTWARE IMPLEMENTATION

Our goal in this project was to accelerate an application code that is already commonly used by practitioners of Bioinformatics. We chose the application SSEARCH35, a component of the FASTA package. SSEARCH35 was run with several differently sized query and database strings. In all cases, the vast majority of execution time was spent in the FLOCAL_ALIGN function which executes the Smith-Waterman algorithm. We studied the parameters of the function to determine the quantity and form of the data passed in and out. In cases where the database is very large which is almost all cases of interest, FLOCAL_ALIGN is called multiple times on successive horizontally oriented sections of the similarity matrix.

SSEARCH35 was designed around the idea that there would be multiple implementations of FLOCAL_ALIGN and defines a standard approach to add an additional implementation. The method involves editing three functions. They include a one-time startup call (init_work), a one-time shutdown call (close_work), and an execute call (do_work) that is called many times.

In the startup call, we allocate page locked memory for the execute call, initialize AAL, and obtain an exclusive lock on the hardware device. Likewise, in the shutdown call, we release the device lock, and release our memory back to the operating system.

The do_work() function is modified to call our hardware function replacement for FLOCAL_ALIGN. In our implementation, we can send an arbitrary length of the database, however, we can only send one character of the query string per processing engine. If, for example, the query string is 300 characters long, and there are only 100 processing engines, we would create three tiles to work over. In this case, each tile will compute the scores for 100 characters of the query string and the entire length of the database string. The information

that is returned from the previous tile is the exact starting data for the next tile, except for the characters in the query string.

VII. RESULTS AND CONCLUSION

We have ported the Smith-Waterman algorithm, a foundational approach to DNA sequencing in the field of Bioinformatics, to FPGA hardware. The algorithm's inherent spatial and temporal parallelism and its compact data structures lend it to effective acceleration in reconfigurable devices. The FPGA form factor used was the newly introduced XD2000i front side bus(FSB) FPGA module from XtremeDataInc. The XD2000i is a CPU socket compatible device that introduces two Altera Stratix III EP3SE260 user FPGAs. A third FPGA is present for communication with the FSB.

Our design is a linear systolic array of processing elements that operate along the rows of the similarity matrix. The current design fits 300 processing elements each of which requires 287 equivalent logic elements and 467 registers. The total design occupies 86% of the chip and operates at 200Mhz. It performs nine billion cell updates per second (BCUPS). Our application is found to be limited by computational intensity and not bandwidth considerations.

A number of design changes are currently being investigated to improve the performance further. These include operating on the second module FPGA (2x), unrolling the inner pipeline (4x), increasing the clock (1.5x) and reducing the score resolution from 20 bits to 8 bits (2.5x). These improvements can theoretically achieve a 30x improvement over our current implementation. A further factor of 1.6 is also possible from the difference in our achieved performance vs. theoretical maximum. These estimates put a theoretical performance maximum at about 450 BCUPS. A future publication will report on these optimizations.

We have also effectively used an early version of the Intel[®] Accelerator Abstraction Layer (AAL), a lightweight middleware application that provides software and hardware developers with a predictable abstraction layer through which they can harness a variety of accelerator options. The implementation of AAL in our user code was easily accomplished and will be well within the reach of those with application development experience. The introduction of AAL resulted in negligible performance degradation.

REFERENCES

- [1] Intel Corporation, White Paper, Enabling Consistent Platform-Level Services for Tightly Coupled Accelerators
- [2] Intel Corporation, *Intel Quick Assist Technology FSB-FPGA Accelerator Platform Software Developer's Manual*
- [3] T.F. Smith and M.S. Waterman, Identification of common molecular subsequences, *J. Mol. Biol.*, **147:195-197**, 1995.
- [4] Altera Corp. White Paper, *Implementation of the Smith-Waterman Algorithm on a reconfigurable supercomputing platform*, September 2007
- [5] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, *J. Mol. Biol.* **48:443-453**, 1970. September 2007
- [6] O. Gotoh, An improved algorithm for matching biological sequences, *J. Mol. Biol.*, **162:705-708**, 1982
- [7] D. Hirschberg, A linear space algorithm for computing maximal common subsequences, *Communication of ACM*, **18:341-343**, 1975