Scalable hardware accelerator for comparing DNA and protein sequences

Conference Paper · January 2006

DOI: 10.1145/1146847.1146880 · Source: DBLP

7 authors, including:

Dirk Stroobandt
Ghent University
198 PUBLICATIONS **1,943** CITATIONS

SEE PROFILE

Yves Van de Peer
Ghent University
641 PUBLICATIONS **37,891** CITATIONS

SEE PROFILE

Scalable Hardware Accelerator for Comparing DNA and Protein Sequences

Philippe Faes, Bram Minnaert, Mark Christiaens, Eric Bonnet, Yvan Saeys, Dirk Stroobandt, Yves Van de Peer

***Abstract*— Comparing genetic sequences is a well-known problem in bioinformatics. Newly determined sequences are being compared to known sequences stored in databases in order to investigate biological functions. In recent years the number of available sequences has increased exponentially. Because of this explosion a speedup in the comparison process is highly required. To meet this demand we implemented a dynamic programming algorithm for sequence alignment on reconfigurable hardware. The algorithm we implemented, Smith-Waterman-Gotoh (SWG) has not been implemented in hardware before. We show a speedup factor of 40 in a design that scales well with the size of the available hardware. We also demonstrate the limits of larger hardware for small problems, and project our design on the largest Field Programmable Gate Array (FPGA) available today.**

I. Introduction

In this paper we present a scalable accelerator for comparing protein sequences. Comparing protein sequences is a very computationally expensive operation that is often performed in the field of bioinformatics. A typical operation would be to compare one newly determined sequence with each sequence in a database, and calculate their similarities. One such database, the NCBI protein database [1], contains almost 3 million protein sequences with lengths ranging from 6 to 36805 amino-acids. The computational complexity of one comparison is of the order $O(N_1 N_2)$, the product of the length of the two sequences.

An algorithm that is often used for comparing sequences is the Smith-Waterman[2] algorithm, which was extended by Gotoh[3]. Our implementation of this Smith-Waterman-Gotoh (SWG) algorithm can compare two sequences of length 1024 in 50 ms on a modern desktop computer. We have been able to accelerate this operation with a factor 40, using a FPGA.

Previous hardware solutions [4], [5] simplify the Smith-Waterman algorithm by setting many of the parameters to a fixed value, by only allowing DNA comparisons, and by not implementing the Gotoh extension. We put only very weak restrictions on the length of the protein sequences, support the Gotoh extension and provide the full flexibility of arbitrary substitution matrices and insertion penalties.

Even with the current FPGAs, we can accelerate the algorithm by a factor 40.

We demonstrate the scalability by projecting our design onto an FPGA which is seven times bigger than the FPGA we used for our initial development.

II. Protein comparison

In this section we will present the basic principles of protein comparison, the Smith-Waterman algorithm and its extension presented by Gotoh.

A. Alignment

A well known problem in bioinformatics is the comparison of Desoxyribonucleic Acid (DNA) sequences and of protein sequences. The former are represented by strings of nucleotides, where possible nucleotides are A, C, G, and T, an alphabet of four symbols. The latter are represented by a string of amino-acids. The 20 possible amino-acids are also represented by roman capital letters, forming an alphabet of 20 symbols. We will focus on comparing protein sequences, but the algorithm and the acceleration we describe is identical when comparing DNA sequences. The calculation for DNA sequences is considerably easier, because of the shorter alphabet.

Comparing genetic sequences is based on the properties of mutations. Over time a sequence mutates and three elementary mutation operations are considered: nucleotides or amino acids can be inserted, deleted or substituted.

Mutations can be visualized by an *alignment*. For example a possible alignment between the protein sequences `AGTTAT` and `TTATATTT` is

```
--AG-TTAT
  |  || |
TTATATT-T.
```

The *edit distance*[1] of an alignment expresses the similarity of the two sequences. It is calculated as the sum of the matching scores of individual characters (positive for matches and usually negative for substitutions) and the penalties of the gaps. For the above example the edit distance is

$$M(\mathsf{A},\mathsf{A}) + M(\mathsf{G},\mathsf{T}) + 3M(\mathsf{T},\mathsf{T}) - 2G(1) - G(2), \qquad (1)$$

with M the so-called *substitution matrix*, which contains the matching scores and mismatch penalties and G, the *gap penalty function*.

B. Smith-Waterman Algorithm

The Smith-Waterman algorithm [2] uses dynamic programming to find the *optimal*, *local* alignment. This is the alignment of two subsequences which produces the maximum edit

[1]This is a misnomer in existing literature: the value we calculate does not have the axiomatic properties of a "distance". A better term would be *similarity*, since higher values represent better matches. We will stick with the conventional term *distance*.

INFOSCALE '06. Proceedings of the First International Conference on Scalable Information Systems, May 29-June 1 2006, Hong Kong

distance. The algorithm is based on a linear gap penalty ($G(n) = \gamma n$). A possible local alignment of the previous two protein sequences is

```
AG-TTAT
|  || |
ATATT-T,
```

another local alignment is

```
AGTTAT
|  |||
A--TTT.
```

Two genetic sequences s_1 and s_2 with characters $s^j_{1|2}$ ($j = 1..N_{1|2}$), the gap penalty function $G(n) = \gamma n$ and a substitution matrix M are given. T. F. Smith and M. S. Waterman showed that the optimal local alignment can be found by constructing four matrices: B (Best), H (Horizontal), V (Vertical) and D (Diagonal). In the first phase of the algorithm, these matrices are constructed by initializing these values for $i = 1..N_1$ and $j = 1..N_2$:

$$B(0,0) = 0, \quad (2)$$
$$B(i,0) = 0, \quad (3)$$
$$B(0,j) = 0, \quad (4)$$

and recursively calculating the other values:

$$D(i,j) = B(i-1,j-1) + M(s^i_1, s^j_2), \quad (5)$$
$$H(i,j) = B(i,j-1) - \gamma \quad (6)$$
$$V(i,j) = B(i-1,j) - \gamma \quad (7)$$
$$B(i,j) = \max(D(i,j), H(i,j), V(i,j), 0). \quad (8)$$

We can calculate $B(i,j)$ without referencing D, H or V by substituting formulae 5, 6 and 7 into formula 8:

$$B(i,j) = \max \begin{cases} (B(i-1,j-1) + M(s^i_1, s^j_2), \\ B(i,j-1) - \gamma, \\ B(i-1,j) - \gamma, \\ 0) \end{cases} \quad (9)$$

The highest element in the B matrix is the comparison distance of the two sequences, $C(s_1, s_2)$. The position of this element denotes the position of the end of the two subsequences.

The second phase of the algorithm is the trace-back phase. Assuming that $B(i^*, j^*)$ contains the highest element of the B matrix we search the maximum values of $D(i^*, j^*)$, $H(i^*, j^*)$ and $V(i^*, j^*)$ as in Eq. 8. Then we move diagonally, horizontally or vertically respectively until we meet a value 0 which indicates the start of the subsequences.

To clarify we present the B matrix in Fig. 1 for the running example, with $\gamma = 1$, $M(i,i) = 3$ and $M(i,j) = -1$ for $i \neq j$. It is clear that the value of each element $B(n,m)$ can be calculated as soon as the values for $B(i,j)$ for $i < n$ and $j < m$ have been calculated.

		T	T	A	T	A	T	T	T
	0	0	0	0	0	0	0	0	0
A	0	0	0	3	0	3	0	0	0
G	0	0	0	0	2	2	2	0	0
T	0	3	3	2	3	2	5	5	4
T	0	3	6	5	5	4	5	8	8
A	0	0	5	9	4	8	7	7	7
T	0	3	3	8	12	11	11	10	10

Fig. 1. B Matrix for the alignment of AGTTAT and TTATATTT.

C. Adaptation

A common use-case for the Smith-Waterman algorithm is to compare one (or more) sequences to all sequences in a large database, in order to find a small number of close matches. Later, a more detailed comparison can be performed on these matches. When this strategy is taken, we are not interested in the exact alignment, but merely in the edit cost.

Since we will not perform a back-track phase, the values of the D, H and V matrices need not be stored. We only need to store one row of the B matrix if we calculate its values in a row-by-row fashion. We store the preliminary *best* result in a separate variable, and update whenever a higher value in the B matrix is calculated. This greatly improves the scalability of the algorithm, since the memory requirements are no longer $O(N_1 \cdot N_2)$, but $O(N_1)$. Since $C(s_1, s_2) = C(s_2, s_1)$, we can change the order of the sequences when one sequence is longer than the other. This allows us to reduce that the memory requirements to $O(\min(N_1, N_2))$.

D. Gotoh Extension

Gotoh [3] proposed a modification to the Smith-Waterman algorithm to support gap penalties of the form $G(n) = \omega + (n-1)\epsilon$. We refer to the original paper for the details and suffice by stating that two more subtractions and two more max-operations are required for calculating an element of the B matrix. We have implemented full support for the Gotoh extension, even though it increases the requirements on the hardware. To our knowledge, no other hardware accelerators have been presented for the Smith-Waterman-Gotoh algorithm, or any other algorithm with the Gotoh extension.

III. Hardware Implementation

A. Overview

The core of our implementation consists of a scalable pipeline, a controller and some memories (Fig. 2). The nodes of the *pipeline* are used for evaluating Eq. 9 in a massively parallel way. Each node can evaluate the equation, and pass the result to the next node. The final node writes its results to a *scratch-pad memory*, so that is can later be retrieved by the first node. The pipeline is connected with the memories that contain the *protein sequences*. Each node has its own copy of the *substitution matrix*, so that we do not need a large global interconnection between each node and a central multi-port memory. Finally, each node is a four-stage pipeline in itself.

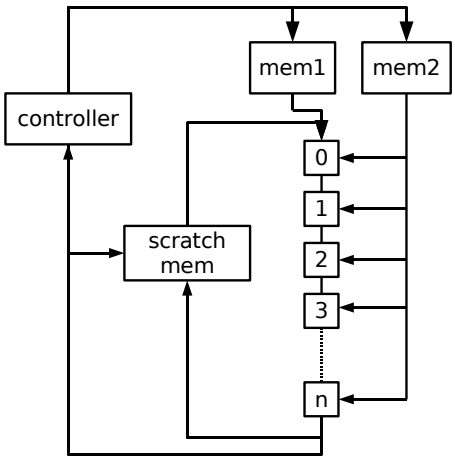

Fig. 2. Architecture of the Hardware

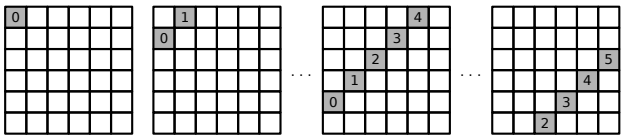

Fig. 3. Calculate Elements of B with a Large Pipeline

We will now explain the basic parts of our SWG implementation, and how they are controlled by the controller in more detail.

B. Pipeline and Scratch-pad Memory

For executing the Smith-Waterman algorithm, one column of the B matrix will be processed by a single node of the pipeline over a period of N_1 clock cycles. The resulting score will be passed on to the next node of the pipeline. Should we have enough cells in the pipeline, we could execute the algorithm by having each column calculated by one node, as depicted in Fig. 3.

In order to calculate $B(i, j)$, a node needs to know the values of $B(i, j-1)$, $B(i-1, j-1)$ and $B(i-1, j)$. This means, for a given timestep n, each node needs the value it calculated at time $n-1$ and the values its predecessor calculated at times $n-1$ and at $n-2$. All these values are stored and transferred in the pipeline as needed.

Because we want to support sequences larger than the size of the pipeline, we introduce a scratch-pad memory. The last node of the pipeline cannot pass its data to a successor node, but rather stores its results in a memory. The data in the memory are fed to the first node of the pipeline when the next set of columns are processed.

The size of the sequences is only limited by the memory available on the FPGA. Our current implementation allows two sequences of 1024 amino-acids, and uses only 35% of the available memory on the FPGA, most of which is used for duplicating the substitution matrix in each node of the pipeline. We are confident that we can easily adapt our design for much longer sequences (at least up to 16384). Moreover,

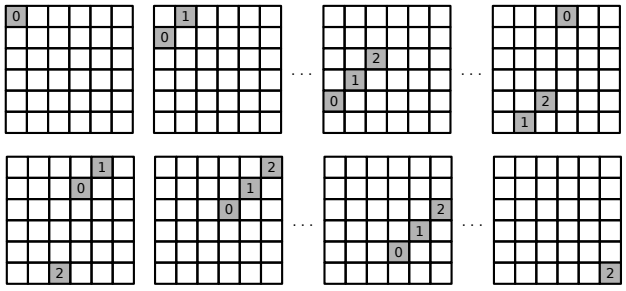

Fig. 4. Calculate Elements of B with a Pipeline and Scratch-pad Memory

the design can be adapted so that one of both sequences is streamed through the pipeline, and thus can be of arbitrary length. Only the size of the scratch-pad memory limits the size of one of both sequences. Since the size of this memory is directly proportional to the maximum value of N_1, we feel our design can scale very well to larger protein sequences.

C. Access to Substitution Matrix

In our implementation each cell of the pipeline processes an element of the dynamic programming array B in one clock cycle, and then moves to the element below the current. Calculating one element requires evaluating Eq. 9.[2] This includes a memory lookup of $M(s_1^i, s_2^j)$. In order to avoid complex interconnections between one central copy of M and all the cells of the pipeline, we place a copy of M in each cell. This increases memory requirements, but not to the extent that memory becomes a bottleneck. Our largest design uses almost all of the FPGA's logic elements, but only 35% of the available memory.

Before starting an alignment operation, we load the substitution matrix into the local memories by piping the values into the pipeline. This again facilitates the data interconnection and scalability, because each cell only has to communicate with the previous and the next cell in the pipeline.

D. Elements of s_1 and s_2

Passing the elements of a protein sequence to each node of the pipeline independently would cause non-scalable interconnections. We take advantage of the predictable nature of the pipeline. If a certain element of s_1 is used by a node, the same element will be used by the successor of that node in the next timestep. This means we can pipe the elements of s_1 into the pipeline at node 0. Every node will pass the element of s_1 on to the next node.

The elements of s_2 are somewhat more complicated. Every node will use the same element of s_2 for a certain period of time before it moves on to another column in the B matrix. However, the neighbouring nodes will never need the same element of s_2. This means we must pass the element directly to each node when the node needs it, i.e. when the node is at the first row of a new column. We notice that no two nodes can ever start a new column at the same time, so we can

[2]Remember we are ignoring the Gotoh extension for the sake of clarity.

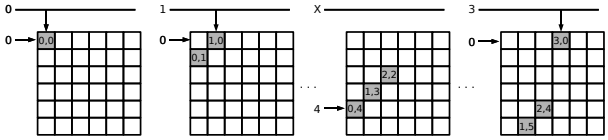

Fig. 5. Passing elements to the pipeline

time-multiplex a single interconnection channel for passing the elements of s_2 (Fig. 5). Whenever a node starts a new column, it will find the correct element s_2 on this s_2-channel. The node stores the element until it has finished its column. If no node starts a new column, the s_2-channel is unused.

E. Pipelining

The operations performed in one node are:

1) calculate the memory address where $M(s_1^i, s_2^j)$ can be loaded from the local memory,
2) load $M(s_1^i, s_2^j)$ from memory,
3) calculate $B(i, j)$ using Eq. 9,
4) check if $B(i, j)$ is a new maximum, if so set maximum to $B(i, j)$.

The results of these operations are passed on to the next node. Because the operations enumerated here are rather complex, they cannot be executed in a single clock cycle on an FPGA. We have constructed each node as a four-step pipeline. This does not degrade the over-all throughput, because each node can still finish one operation per clock cycle. It only means that four matrix elements will be in the pipeline of a single cell at the same time. When we revise Figures 3 and 4, we should think of the highlighted elements as those elements that are in the final stage of the pipeline of the indicated node. The other stages are already filled with elements that will come to the final stage in the next clock cycles.

F. Related work

Systolic arrays, somewhat similar to our pipeline, have been proposed for sequence matching by [5]. The original implementation, however requires that the length of the systolic array is twice the length of the largest sequence. Since protein sequences with thousands of amino-acids are known today, this requirement is not feasible on current FPGAs. Other authors [4] have proposed a systolic array that can compare a sequence of arbitrary length to a sequence of a fixed length (*in casu* 38). This is too large a restriction for most bioinformatics queries. By contrast, our current implementation can align two sequences of length 1024, and can easily be scaled to accommodate larger sequences.

G. Communication between Java and FPGA

We have developed a hardware accelerator using a transparent interface[6], [7] between the high level control (programmed in Java[8]) and the actual FPGA. First we developed a pure Java application for protein alignments, based on [9]. We determined which Java method takes up most computation time and developed a hardware accelerator for this method.

When we execute the unmodified Java program on our adapted Java Virtual Machine (JVM), the JVM will automatically intercept the alignment method. Instead of executing this method on the main processor, it has the alignment executed on the FPGA. All subsequent communication between the main processor and the FPGA is handled by the JVM.

The use of a transparent interface leaves the choice of hardware to the JVM, so that the programmer can concentrate on the algorithm. The transparent interface is also robust to multi-threading. It will be possible to fork different Java threads that each invoke one FPGA. Moreover, when a thread waits for an FPGA to finish its work, the Central Processing Unit (CPU) becomes available for a different Java thread. Our design has the potential of scaling well with multiple FPGAs and with multiple CPUs.

IV. Simulated Speed-up

We have simulated the computation of an alignment of two proteins of size 1024, for various pipeline sizes. Recall that the software implementation takes 50ms to execute. As can be seen from Fig. 6, the initial speedup is proportional to the amount of added hardware. For larger pipelines, the speedup decreases for two reasons.

Firstly, since the computation time decreases, the communication overhead can no longer be ignored. The minimal time required, for infinitely fast hardware is the time to copy the data to the FPGA over the PCI bus. This problem can be reduced by copying the substitution matrix and the query sequence[3] only once so that only the sequences from a database need to be copied over the bus. The bus can also be used more efficiently by using Direct Memory Access (DMA) transactions. Theoretically DMA burst transaction can reach a throughput of 133MB/s, while our current PCI bus simulation allows for a throughput of 6MB/s. We could theoretically accelerate the communication by a factor of up to 22. Fig. 6 also shows the execution times without communication overhead.

Secondly, when the size of the pipeline approaches the size of the protein sequence, the pipeline cannot be used optimally. The pipeline will be filled with bubbles (nodes that do no useful calculations) and it will delay the computation. This is demonstrated more clearly in Fig. 7 , where we align a protein of size 70 with one of size 112.

These results show that it may be useful to *downscale* the hardware when expecting many short alignments. Since we use an FPGA, it is feasible to reconfigure the hardware at run time, so that the hardware can be adapted to the expected use.

V. Synthesis Results and Measured Speed-ups

We have synthesized our design for the Altera Stratix 1s25 FPGA of our target system. The largest feasible design has a systolic array of 59 nodes. The highest possible clock frequencies range from 62.12 MHz for an array of one single node to 49.49 MHz for the array of 59 nodes. This confirms our claim that the clock frequency does not degrade when scaling the design up to larger systolic arrays.

[3]The query sequence is the sequence that is held constant while scanning over an entire database.

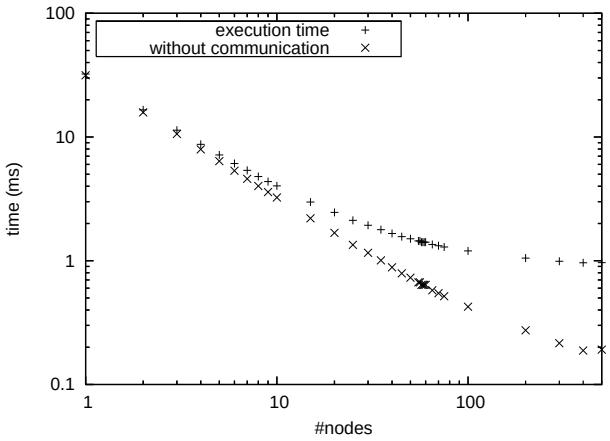

Fig. 6. Execution times at 33 MHz, $N_1 = N_2 = 1024$

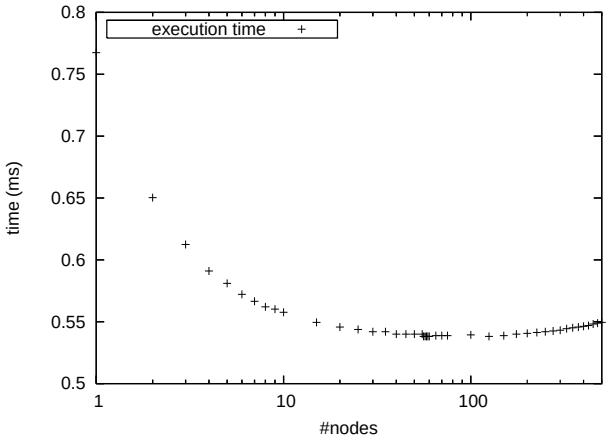

Fig. 7. Execution times at 33 MHz, $N_1 = 70$, $N_2 = 112$

If the FPGA were to operate at its maximum frequency, instead of the fixed frequency of the PCI bus, the execution times would be as depicted in Fig. 8. Because the clock frequency is relatively stable, the same trends can be detected as in Fig. 6, although the clock frequency introduces a lot of noise in the graph.

We have verified the simulation results on a desktop PC with an AMD Athlon MP 2600+ processor, and extended with an Stratix 1s25 FPGA on an Altera PCI Development Board, Stratix edition. The results of the simulation, and the actual execution on the JVM with the FPGA are summarized in Table I. The actual measured results are slightly faster than the predicted simulation results. The difference can be explained because, even though some execution time is spent in software, the actual PCI bus arbitration is not identical to the simulated bus.

TABLE I

EXECUTION TIME FOR ALIGNMENT OF TWO SEQUENCES OF LENGTH 1024

execution time (ms)	software	1 node	59 nodes
simulated, at 33 MHz	N.A.	32.32	1.413
simulated, at max clock freq.	N.A.	17.61	0.86
measured, FPGA at 33 MHz	50.07	32.15	1.28

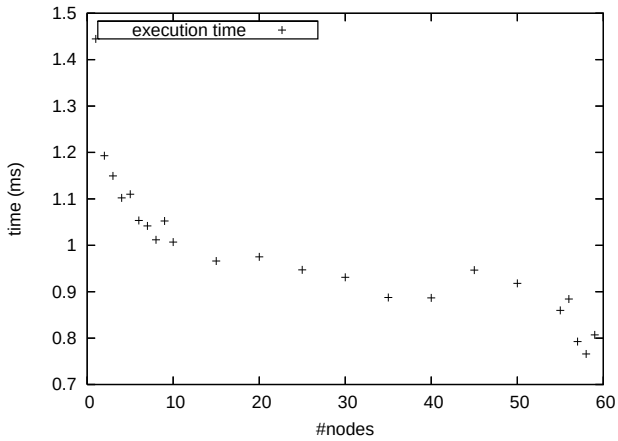

Fig. 8. Execution times at maximum clock frequency

VI. PROJECTED RESULTS

We synthesized our design for the largest FPGA available from Altera, the Stratix II 180 (EP2S180F1508C5) chip. We were able to fit a pipeline of 350 nodes into a single chip. The clock frequency ranges from 70.05 MHz for a design with only one node to 42.84 MHz for the design with 350 nodes. This is an acceptable variation in clock frequency over such a wide range of designs. It confirms again the benefit of local interconnections, with a very limited amount of global interconnections. We can conclude that our design can be scaled up to fit the largest FPGAs known today.

VII. ACKNOWLEDGMENTS

Bram Minnaert worked on this research as a master student. This research in part is supported by grant 020174 of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), by GOA project 12.51B.02 of Ghent University, by grant G.0021.03 of the Fund for Scientific Research – Flanders and by hardware donations from Altera. Philippe Faes is supported by a scholarship of the IWT-Vlaanderen.

REFERENCES

[1] "National center for biotechnology information." http://www.ncbi.nlm.nih.gov.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[3] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705–708, 1982.

[4] B. West, R. D. Chamberlain, and R. S. Indeck, "An FPGA-based search engine for unstructured database," in *Proc. of 2nd Workshop on Application Specific Processors (2003)*.

[5] R. J. Lipton and D. P. Lopresti, "A systolic array for rapid string comparison," in *1985 Chapel Hill Conference on VLSI* (H. Fuchs, ed.), (Rockville, MD), pp. 363–376, Computer Science Press, 1985.

[6] Ph. Faes, M. Christiaens, D. Buytaert, and D. Stroobandt, "FPGA-aware garbage collection in java," in *2005 International Conference on Field Programmable Logic and Applications (FPL)* (T. Rissa, S. Wilton, and P. Leong, eds.), (Tampere, Finland), pp. 675–680, IEEE, 1 2005.

[7] Ph. Faes, M. Christiaens, and D. Stroobandt, "Transparent communication between Java and reconfigurable hardware," in *Proceedings of the 16th IASTED International Conference Parallel and Distributed Computing and Systems* (T. Gonzalez, ed.), (Cambridge, MA, USA), pp. 380–385, ACTA Press, 11 2004.

[8] K. Arnold and J. Gosling, *The Java Programming Language*. Addison Wesley, 1996.
[9] J. Moustafa, “Open source Java implementation of Smith-Waterman.” http://jalinger.sourceforge.net.