

Hardware Acceleration of HMMER on FPGAs

Steven Derrien · Patrice Quinton

Received: 7 November 2007 / Revised: 2 June 2008 / Accepted: 16 August 2008 / Published online: 2 October 2008
© 2008 Springer Science + Business Media, LLC. Manufactured in The United States

Abstract We propose a new parallelization scheme for the `hmmsearch` function of the HMMER software, in order to target FPGA technology. `hmmsearch` is a very compute intensive software for biological sequence alignment, based on profile hidden Markov models. We derive a flexible, generic, scalable hardware parallel architecture which can accelerate the core of `hmmsearch` by nearly two orders of magnitude, without modifying the original algorithm of this software. Our derivation is based on the expression of the algorithm as a set of recurrence equations, and we show in a systematic way how a very efficient parallel version of the algorithm can be found by combining scheduling, projection, partitioning, pipelining and precision analysis. We present the performance of the implementation of this parallel algorithm on a FPGA platform.

Keywords HMM · Bioinformatics · Reconfigurable computing

1 Introduction

Over the last few years, FPGA based accelerators haven't proved to be a very attractive solution for implementing many of the most compute intensive bio-computing algorithms. Among others, FPGA implementations of Smith and Waterman [16], BLAST [1, 11], ClustalW [17] or even Weighted Finite Automaton [8] algorithms have exhibited impressive speed-up

factors, making them a very viable alternative to expensive supercomputing infrastructures such as vector computers or PC clusters.

Among other approaches, profile based hidden Markov models (HMM) have been recently used by biologists to predict the structure and function of a protein directly from its representation as an amino-acid sequence [10]. The approach consists in building and providing probabilistic models of protein sequences that share similar structures or functions. As of today, there are several software implementations of this model, the `hmmsearch` package being one of the most widely used.

Profile HMM can be used in different contexts. For example, when trying to discover the biological function of a given amino-acid sequence, the user matches this sequence against a database of profiles such as the *Pfam* database [2] so as to find the best corresponding profile. This type of operation is handled by the `hmmpfam` utility of `hmmsearch`. Another use is *gene annotation*, where the user wants to match a whole sequence database against one or several profile HMM databases, as done by the `hmmsearch` utility. However, given the computational complexity of the algorithm, such *intensive comparisons* lead to prohibitive execution time (in the order of days or weeks): it is therefore natural to look for means of accelerating the execution of this software.

We propose in this work to accelerate using reconfigurable hardware the most time consuming routine of the `hmmsearch` software [5] of HMMER, namely the `P7Viterbi` procedure which computes the score between a HMM and an observed sequence. We do so by using linear space–time mappings based on the so-called *polyhedral model*. This leads to a flexible parallel

S. Derrien (✉) · P. Quinton
IRISA, Rennes, France
e-mail: sderrien@irisa.fr

architecture template which handles the feedback loop present in the Plan7 HMM model used by HMMER.

This paper is organized as follows. Section 2 briefly introduces the principles of the profile HMM based algorithm and surveys related work on accelerating this algorithm both in hardware and software. Section 3 presents the principles of our parallelization schemes, and Section 4 details the space–time mapping refinements which lead to the final hardware architecture. Section 5 studies the impact of arithmetic precision on the algorithm quality of results and on its hardware implementation performance. Section 6 gives experimental results in terms of resource usage and performance improvement. Section 7 concludes and presents future work directions.

2 Background

2.1 Principles

Figure 1 depicts the Plan7 HMM used in HMMER. It is a probabilistic model of a family of sequences (see [10] for more details) which contains three types of states: match states (square nodes), insertion states (diamond nodes) and deletion states (circle nodes). Running the `hmmsearch` tool consists in matching a single profile HMM against a large number of input sequences, and in finding the sequences having high similarity with this HMM.

The most time consuming part in this program is the `P7Viterbi` kernel, which is described in Table 7 of Appendix. This kernel computes a similarity score between the profile HMM and the sequence at hand using dynamic programming.

Matching a single HMM against a protein database is a very time consuming process which is repeated many times during intensive comparisons. Profiling shows that the `P7Viterbi` kernel accounts for more than

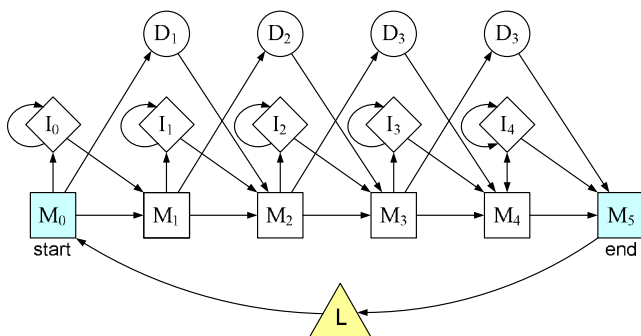


Figure 1 Structure of a Plan7 HMM.

97 % of the execution time. It is therefore a perfect candidate for hardware acceleration.

2.2 Related Work

There have been several attempts to accelerate HMMER using SIMD features of modern CPUs [12], parallel machines [20], GPUs [9] and even Network Processors [21].

Some authors have proposed to speed-up HMMER by using specialized hardware. For example [13] have studied different hardware-software partitioning schemes and their impact on performance and quality of results. They suggest decoupling the scoring step (based on the `P7Viterbi` algorithm) from the *traceback* step, since this later one is only useful on a very small subset of high scoring sequences. Their partitioning hence consists of executing the `P7Viterbi` kernel in hardware, leaving the *traceback* to a software host system.

In order to simplify its parallelization, they suggest modifying the original `P7Viterbi` algorithm by removing the outer loop carried dependency (see Section 4.1) which forbids the use of loop level parallelism. As acknowledged by the authors, this modification has a severe impact on the results that are obtained through this approach. In particular, this *feedback-free* modified algorithm actually correspond to a scoring function that does not account for low scoring motif that occur several time in the protein motif ; as a consequence the *feedback-free* algorithm induces a lot of false-negative matches (in other words, some potentially interesting hits are *missed* by the algorithm).

They propose to reduce the impact of this simplification by (1) reducing the threshold to reduce the filtering sensitivity and (2) unroll the HMM model so as to enable accounting for multiple occurring low score motifs. The later solution comes at the expense of a more computationally expensive algorithm (twice the original algorithm complexity). Since they do not propose an actual implementation, the remaining of their paper focuses on providing early performance estimates using a simple performance model based on a previously existing Smith Waterman algorithm hardware implementation.

On the other hand, [15] propose an actual FPGA implementation of an profile HMM application and claim speed-up factors up to two orders of magnitude. However they do also consider a *feed-back* free implementation of the model. They later extended their approach [18] to handle the `P7Viterbi` feedback loop. However, their design faces the issue that all PEs may access the whole look-up table of `P7Viterbi`, which

3.2 Managing Arbitrary Sized Problems

We observe that several calls to P7Viterbi using the same HMM profile are done by the `hmmsearch` tool and we thus propose to merge all the sequences into a single macro sequence. Each sequence is delimited by a special character, the role of which is to reset the matching scores to their initial $-\infty$ value and to indicate that a new Viterbi algorithm instance is to take place.

On the other hand, even though the length of the HMM profile M remains constant for a given execution of `hmmsearch`, it can be different for each `hmmsearch` execution instance (`hmmsearch` motif length vary between 50 and 650 (see Fig. 3, with an average size of 200). We must therefore design an architecture capable of handling arbitrary sized HMM motifs. This can be done by simply inserting idle states in the HMM motif. The role of these idle states is to propagate the scores of the last non-idle state until it reaches the feedback loop. In the rest of the paper, we will therefore consider that constant M denotes the maximum allowed model size of our architecture.

3.3 Linear Space–Time Mappings

Given a system of recurrent equations similar to the one presented in Eq. 1, we want to derive a space–time mapping that is a linear transformation which gives, for any indexed variable in the SARE:

- A *logical execution* time instant, in the form of a linear function of the index (which we call *schedule*), written as

$$s(i_0, \dots, i_m) = s_0i_0 + \dots + s_m i_m \quad .$$

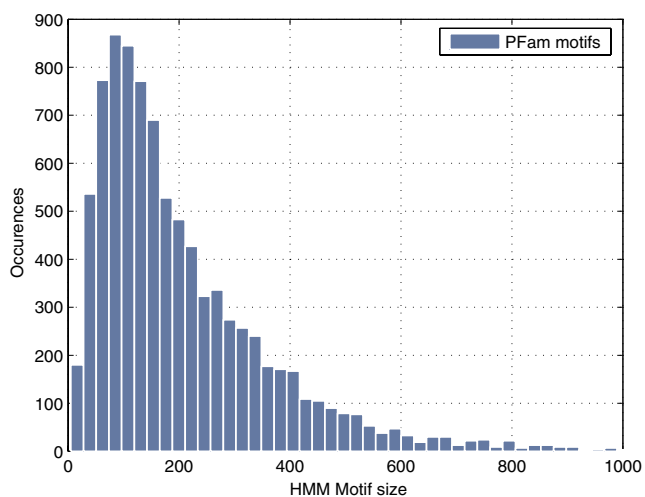


Figure 3 HMM model size distribution in the Pfam database.

- A *physical location*, i.e. coordinates in a processor space. This location is also expressed in the form of a linear function of the index (that we call *allocation function*). In our case we are only interested in linear arrays. We therefore write

$$p(i_0, \dots, i_m) = \alpha_0i_0 + \dots + \alpha_m i_m \quad .$$

Of course, this space–time mapping must satisfy several conditions.

- First, the chosen schedule must enforce all data dependencies present in the SARE. For $u, v \in \mathcal{D}$ with $u \delta v$, the schedule function must guarantee that $s(v) > s(u)$.
- Then, the space–time mapping must be conflict-free: there must be no u and v in \mathcal{D} such that $v \neq u$, $s(u) = s(v)$ and $p(u) = p(v)$.

Proving that a schedule is conflict-free when each PE executes computations in a one dimensional domain is relatively straightforward (see [19]), but this is more involved for higher dimensional domains. To solve this problem, we use [3] results of *juggling schedules*, which can be summarized as follows. Given a rectangular domain \mathcal{P} in \mathbb{Z}^{m+1} defined as :

$$\mathcal{P} = \{i_0, \dots, i_m \mid 0 \leq i_0 < N_0, \dots, 0 \leq i_m < N_m\} \quad ,$$

a schedule is said to be *juggling* if it has the following properties:

- It is *conflict-free*, i.e. there are no two distinct iterations (i_0, \dots, i_m) and (i'_0, \dots, i'_m) in \mathcal{P} which are scheduled at the same time instant.
- It is *dense*, i.e. the number of schedule steps separating the execution of iterations (i_0, i_1, \dots, i_m) and $(i_0 + 1, i_1, \dots, i_m)$ is $\prod_{k=1}^m N_k$.

Darte and al. have shown that juggling schedules are of the form below (up to a permutation of index i_k) with $\text{pgcd}(\lambda_k, N_k) = 1$:

$$s(i_0, \dots, i_m) = \left(\prod_{k=1}^m N_k \right) i_0 + \lambda_1 \left(\prod_{k=2}^m N_k \right) i_1 + \dots + \lambda_m i_m \quad (3)$$

Notice that we may have used well established results on *multi-dimensional schedules* [6, 7] to find conflict-free schedules, but this would lead to more complex proofs in our case where the domains are hyperparallelepipeds.

In the next section, we evaluate and refine several space–time mappings in order to obtain the best possible implementation.

4 Design Space Exploration

In this section, we propose to explore part of the design space for our hardware accelerator. Our approach is incremental, in the sense that we start from a initial (i.e. naive) parallelization scheme, which we refine until we obtain a well-suited solution.

4.1 A Naive Linear Schedule

In this subsection, we apply the parallelization methods introduced in Section 3 to the P7Viterbi kernel (with little success). From the SARE given in Eq. 1, it follows that the variables are indexed by two index i and k , and that the general form of a schedule is therefore $s(i, k) = s_0i + s_1k$. The data dependencies of this SARE are:

- $(i-1, k-1) \delta(i, k)$ therefore, we must ensure $s(i, k) > s(i-1, k-1)$, which in turn implies $s_0 + s_1 \geq 1$.
- $(i-1, k) \delta(i, k)$ which leads to $s_0 \geq 1$.
- $(i, k-1) \delta(i, k)$ which leads to $s_1 \geq 1$.
- $(i-1, M) \delta(i, k)$ which leads to $s_0 \geq M$.

Using integer linear programming we can derive the best (i.e. the fastest) schedule which is $s(i, k) = Mi + k$. Let us combine this scheduling function with the allocation function $p(i, k) = k$. This allocation leads us to a linear array of M processors, each of them executing iterations along axis i . From this schedule, we can see that M cycles separate the execution of iteration $(i-1, k)$ from the execution of (i, k) . In other words, on a given processor $p(i, k_0) = k_0$, the schedule is such that we perform a useful computation only every M cycles. This results in a highly inefficient space–time mapping.

Indeed, we can show that the schedule $s(i, k) = Mi + k$ could be realized on a single PE architecture. Using Darte et al. result on *juggling schedules*, we know that any schedule in the form $s(i, k) = Mi + \lambda k$ juggles for $0 \leq k \leq M$ if $pgcd(\lambda, M) = 1$, which is our case here.

Clearly, such a schedule has little practical interest, apart from convincing us that there is no chance of taking advantage of loop level parallelism in this loop nest.

4.2 Adding a Dimension to the SARE

Although loop level parallel schedules cannot be found in the P7Viterbi routine, we observe that running the hmmsearch tool consists in completely independent matching of a single HMM against a large number of input sequences: these matching are completely independent and can therefore be run in parallel.

We model this additional parallelism by adding to the SARE a new index which identifies instances of the kernel running in parallel. Call j this additional index. The new iteration domain \mathcal{D}' is then:

$$\mathcal{D}' = \{i, j, k \mid 0 \leq i < L, 0 \leq j < N, 0 \leq k \leq M\} \quad (4)$$

where N stands for the number of sequences that are matched in parallel during each realization of the modified SARE, which is shown below:

$$\begin{cases} x_{i,j,k} = f_1(y_{i-1,j,M}, hmm_k, \\ \quad \quad \quad x_{i-1,j,k}, TAB(seq_{i,j}, k), x_{i,j,k-1}, x_{i-1,j,k-1}) \\ y_{i,j,M} = f_2(hmm_k, x_{i,j,M-1}) \end{cases} \quad (5)$$

Denote

$$s(i, j, k) = s_0i + s_1k + s_2j \quad (6)$$

the scheduling function of this new SARE. The intrinsic data-dependencies remain unchanged by the transformation (all P7Viterbi instance are independent), therefore we can write the constraints on the scheduling function as:

$$\begin{aligned} (i-1, k-1) \delta(i, k) &\Rightarrow s_0 + s_1 \geq 1, \\ (i-1, k) \delta(i, k) &\Rightarrow s_0 \geq 1, \\ (i, k-1) \delta(i, k) &\Rightarrow s_1 \geq 1, \\ (i-1, M) \delta(i, k) &\Rightarrow s_0 \geq M. \end{aligned}$$

This lead to many possible schedules, among which only two really deserve attention: a *wavefront* and an *interlaced* schedule.

4.3 Wavefront Space–Time Mapping

In this approach we use the space–time mapping given by

$$\begin{aligned} s(i, j, k) &= Mi + k, \\ p(i, j, k) &= j, \end{aligned} \quad (7)$$

which obviously enforces data dependencies and is conflict-free. It is illustrated (for $M = 4, N = 4$ and $L = 5$) in Fig. 4, in which dashed arrows represent the iteration execution order on a single PE, while solid lines show which iterations are actually executed in parallel.

The wavefront space–time mapping is a very natural parallelization scheme in which each PE executes a distinct P7Viterbi kernel instance, and the chosen value for parameter N directly controls the number of PEs in the architecture. Although these P7Viterbi instances share the same HMM parameters values, we must account for the fact that variable TAB in Eq. 5 is used as a lookup table with $seq_{i,j}$ and k as indices.

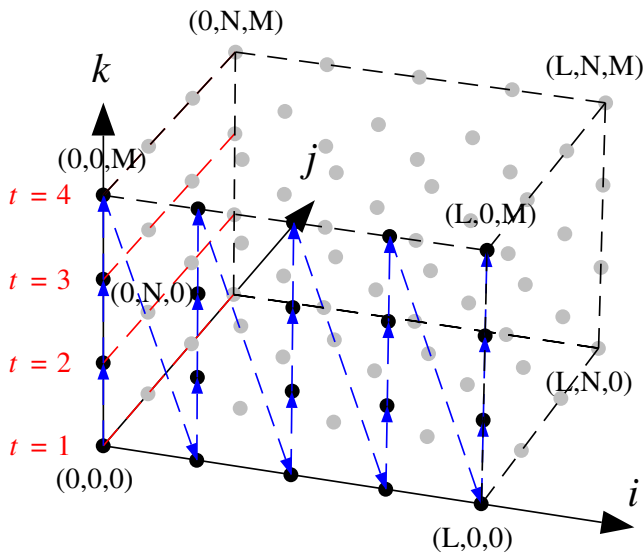


Figure 4 Wave-front space–time mapping.

At a given time instant t , all PEs share the same value of i, k , therefore they must access the same subset $TAB[*][k]$. Moreover, each PE accesses the whole table across time. As a consequence, all PEs must either hold a copy of the TAB variable, or they must access a shared copy of it. In the full `P7Viterbi` kernel the size of TAB is quite large: for an average HMM model size of 250, each processor requires two $5,000 \times 32$ lookup tables: this would severely limit the number of PEs that can be implemented.

One solution to this problem is to use a crossbar-like interconnect structure to distribute the table content (see Section 2.2). Instead, we propose another space–time mapping which allows the TAB variable to be distributed among the PEs, thereby reducing the architecture memory footprint to its minimum, while avoiding the need for a complex crossbar interconnect structure (as opposed to [18]).

4.4 Interlaced Space–Time Mapping

This improved space–time mapping is given in Eq. 8, and is illustrated in Fig. 5:

$$s(i, j, k) = Mi + j + k, \tag{8}$$

$$p(i, j, k) = k.$$

The reader can check that this mapping enforces data-dependencies and is conflict free if and only if $M = N$. This results in a mapping in which $N = M$ PEs are running in parallel. Since PE_p only executes iterations for which $k = p$, the hardware cost of the lookup table implementation is much lower: PE_p only accesses the

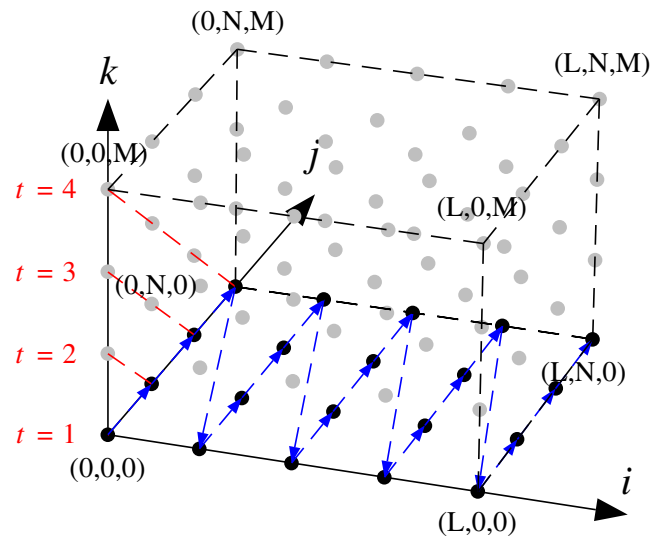


Figure 5 Interlaced space–time mapping.

subset $TAB[*][p]$. It is thus possible to distribute the content of the lookup table among the PEs, each PE holding $1/M^{th}$ of it. The memory cost of a single PE is reduced by a factor of M compared to the approach of Section 4.3.

On the other hand, we have no control over the resource usage of our architecture, since the number of PEs must be equal to the HMM model size M . This is again a severe limitation, since it is very unlikely that such architecture can be implemented on a FPGA even for moderate size model ($M \approx 50$).

4.5 Managing Resource Constraints

To overcome this new problem, we propose to use partitioning (also called *tiling*) of the iteration space. The two main partitioning techniques are LSPG (locally parallel globally sequential) partitioning [14] and LPGS (Locally Parallel Globally Sequential) [3]. LPGS is of no interest here because of the feed-back loop: it would not allow partitioning along dimension k in order to reduce the number of processors in the architecture. To the contrary, LSGP, as we shall see, can be used.

The LSGP partitioning transformation consists of two steps:

- first consider the processors of the initial space–time mapping as being *virtual*, and tile the virtual processor space.
- second, map each tile of the virtual processor space to a single *physical* processor which executes in turn the calculations associated with all virtual processors of the tile.

As far as the SARE is concerned, tiling the virtual processor domain amounts to replacing the processor space index k by two new indexes (v, p) defined by the equality $k = \sigma p + v$. Parameter σ is called the *tile width*, and we have $0 \leq v < \sigma$ and $0 \leq p < M'$, with $M' = \lceil \frac{M}{\sigma} \rceil$ and $0 \leq \sigma p + v < M$. The resulting domain becomes:

$$\mathcal{D}'' = \{i, j, v, p \mid 1 \leq i < L, 1 \leq j < N, 1 \leq v < \sigma, 1 \leq p < M'\}. \tag{9}$$

To cope with this transformation, the SARE is rewritten by using indices (i, j, v, p) instead of (i, j, k) and by modifying the data-dependencies accordingly.

For example, data dependency between $(i - 1, j, k)$ and (i, j, k) now holds between points $(i - 1, j, v, p)$ and (i, j, v, p) and is thus still uniform. When a dependency spans the k index, it leads to more complex situations. For example, consider the dependency between $(i, j, k - 1)$ and (i, j, k) . Depending on whether the values of k and $k - 1$ belong to the same tile (i.e. lead to the same value of p), the original dependency is then transformed into two distinct data dependencies:

$$\begin{cases} (i, j, v - 1, p) \delta (i, j, v, p) & \text{when } v > 1, \\ (i, j, \sigma, p - 1) \delta (i, j, 1, p) & \text{when } v = 1. \end{cases} \tag{10}$$

The partitioned SARE thus becomes:

$$\begin{cases} z_{i,j,v,p} = \begin{cases} x_{i,j,v-1,p} & \text{when } v > 1 \\ x_{i,j,\sigma,p-1} & \text{when } v = 1 \end{cases} \\ w_{i,j,v,p} = \begin{cases} x_{i-1,j,v-1,p} & \text{when } v > 1 \\ x_{i-1,j,\sigma,p-1} & \text{when } v = 1 \end{cases} \\ x_{i,j,v,p} = f_1(y_{i-1,j,M}, hmm_{\sigma p+v}, x_{i-1,j,v,p}, \\ \quad \text{TAB}(seq_{i,j}, v + \sigma p), z_{i,j,v,p}, w_{i,j,v,p}) \\ y_{i,j,\sigma,M'} = f_2(hmm_M, x_{i,j,\sigma,M'}) \end{cases} \tag{11}$$

We now can use the following space–time mapping:

$$\begin{aligned} s(i, j, v, p) &= \sigma M' i + \sigma j + v + \sigma p, \\ p(i, j, v, p) &= p. \end{aligned} \tag{12}$$

We can show that this mapping juggles for the domain \mathcal{D}'' if we choose $N = M'$. It is illustrated in Fig. 6 where the black dots correspond to the iteration sub-space allocated to the first PE. Here $\sigma = 2$, and therefore $\lceil \frac{M}{\sigma} \rceil = 2$ iterations are executed at a given time step.

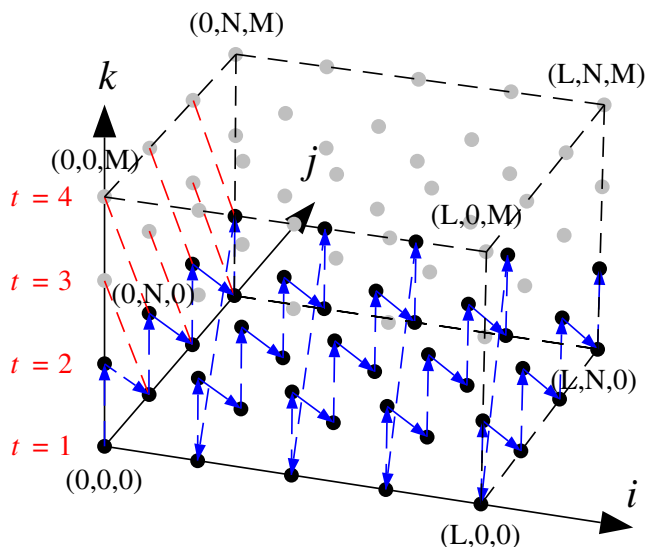


Figure 6 Partitioned space–time mapping.

4.6 Pipelining the PE Datapath

Although the architecture of Section 4.5 allows for a resource constrained implementation, it still requires that a whole loop iteration be executed within a single clock cycle. Given that the *complete P7Viterbi* loop body contains more than 20 arithmetic operations among which 7 lie in the critical path, the final maximum clock frequency of our design is likely to be disappointing if we do not pipeline the data path.

One solution is to modify the scheduling so that the pipelining of the loop body execution becomes possible (see [4]). This is achieved by applying another tiling transformation along axis j .

Let us write $j = \lambda jj + l$, with jj and l being the two new indexes. The new SARE is given in Eq. 14, and its variables are defined over the domain:

$$\mathcal{D}''' = \{i, jj, l, v, p \mid 1 \leq i < L, 1 \leq \lambda jj + l < N, 1 \leq l \leq \lambda, 1 \leq v < \sigma, 1 \leq p < M'\} \tag{13}$$

by the following recurrence equations:

$$\begin{cases} z_{i,jj,l,v,p} = \begin{cases} x_{i,jj,l,v-1,p} & \text{when } v > 1 \\ x_{i,jj,l,\sigma,p-1} & \text{when } v = 1 \end{cases} \\ w_{i,jj,l,v,p} = \begin{cases} x_{i-1,jj,l,v-1,p} & \text{when } v > 1 \\ x_{i-1,jj,l,\sigma,p-1} & \text{when } v = 1 \end{cases} \\ x_{i,jj,l,v,p} = f_1(y_{i-1,j}, hmm_{\sigma p+v}, x_{i-1,jj,l,v,p}, \\ \quad \text{TAB}(seq_{i,jj,l}, v + \sigma p), z_{i,jj,l,v,p}, w_{i,jj,l,v,p}) \\ y_{i,jj,l,\sigma,M'} = f_2(hmm_M, x_{i,jj,l,\sigma,M'}) \end{cases} \tag{14}$$

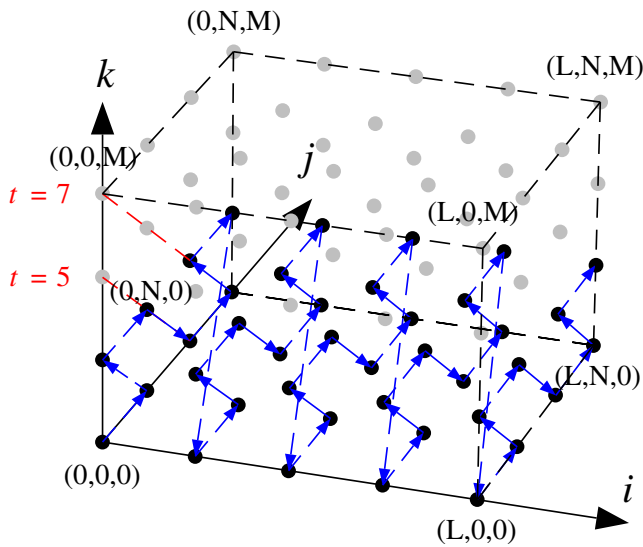


Figure 7 Pipelined space–time mapping.

Using the following space–time mapping:

$$s(i, jj, l, v, p) = \sigma \lambda M' i + \sigma (\lambda jj + j) + \lambda v + \lambda \sigma p, \\ p(i, j, v, p) = k. \tag{15}$$

we can show that this mapping juggles iff $0 \leq jj < N'$ and $0 \leq l < \lambda$ when N' and λ satisfy $\lambda N' = M'$. This space–time mapping is illustrated in Fig. 7 (with $\sigma = 2$ and $\lambda = 2$).

The execution of two dependent iterations is now separated by at least λ cycles (for dependency $(i, jj, l, \sigma, p - 1) \delta (i, jj, l, v, p)$), and by at most $\lambda \sigma M' + 1$ cycles (for dependency $(i - 1, jj, l, \sigma, p - 1) \delta (i, jj, l, v, p)$). By carefully selecting λ , it is therefore possible to find the best tradeoff between resource cost (each PE must implement a $\lambda M + 1$ deep delay-line) and the operating frequency (the data-path corresponding to the loop body can be implemented with λ pipeline stages).

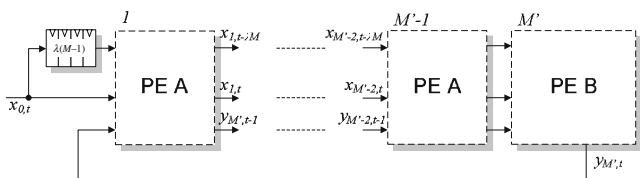


Figure 8 The final processor array architecture for the P7Viterbi routine.

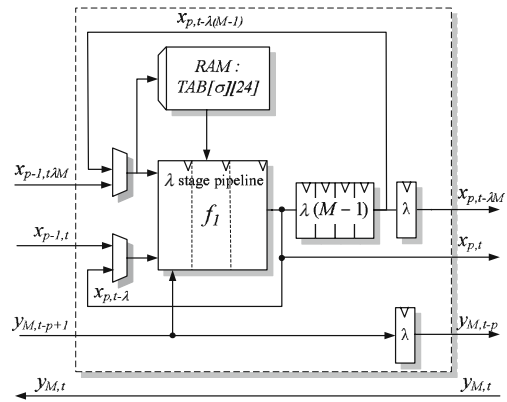


Figure 9 PE A internal structure.

4.7 Resulting Architecture

Using the mapping of Section 4.6, we obtain an architecture which consists of a linear array of M' processors as illustrated in Fig. 8. All PEs in the array have local interconnections except for the last one (PE $_{M'}$) which broadcasts part of its results (this broadcast corresponds to the HMM feedback loop).

From the schedule, we can derive the depth of the delay lines associated to each data dependency in a PE. This leads to two distinct PE internal architectures as illustrated in Figs. 9 and 10, and where functions f_1 and f_2 are implemented as λ stage pipelined datapath. These PEs are active every cycle, except for PE $_{M'}$ in which the variable $y_{i,M}$ is only updated (and broadcasted) every σ cycles.

Table 1 summarizes the characteristics of the various space–time mappings for the full P7Viterbi kernel as a function of the HMM model size (M), the design

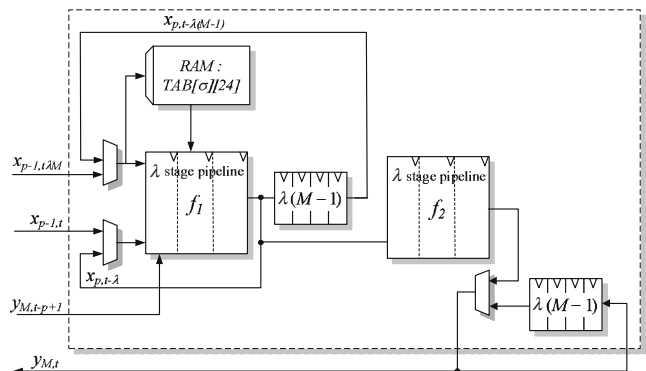


Figure 10 PE B internal structure.

Table 1 A summary of the characteristics for the proposed space–time mappings

Design	#PE	Memory per PE (in bytes)	#Pipeline stages
Wave front ($\sigma = 1, \lambda = 1$)	N	$O(MA)$	1
Interlaced ($\sigma = 1, \lambda = 1$)	M	$O(M + A)$	1
Partitioned ($\lambda = 1$)	$\lceil M/\sigma \rceil$	$O(M + \sigma A)$	1
Pipelined	$\lceil M/\sigma \rceil$	$O(\lambda M + \lambda \sigma A)$	λ

M is the length of the profile HMM, N is the number of sequences, A is the number of amino-acids ($A = 25$), σ is the size of the partition tile, and λ is the number of pipeline stages.

parameters (σ, λ, N) and the number A of distinct bases in the Amino Acid set (here 25).

In all cases, the PE memory cost grows linearly with parameter M (the size of the HMM), and with the number of instances of P7Viterbi running in parallel on each PE (we must store as many matrix columns in a PE as there are P7Viterbi instances alive in this PE).

5 Managing Performance/Precision Trade-Off

The original P7Viterbi routine operates on 32 bit integers. These integers values correspond to the images of probabilities that underwent a *log* encoding. This encoding allows a strength-reduction optimization: floating-point products are replaced by integer additions which are faster on most CPUs.

Even though 32 bit integer arithmetic has a very reasonable area cost on the FPGA, we would like to study whether it is possible to encode the algorithm variables on smaller bitwidths so as to reduce the area footprint of the loop body hardware implementation, and therefore allow more (and faster) processing elements to be implemented on the FPGA. In [15], the problem of finding the best encoding is mentioned but is not studied in depth. The authors propose to use 24 bit arithmetic which they claim as being sufficient without any more details. In the following, we show that further reducing the bitwidth is possible, and does not

impact the validity of results (i.e no false negative can appear).

5.1 Determining Actual Precision Requirements

The original HMMER implementation was implemented using 32 bit integers since it is the natural encoding for CPUs. However, as it is often the case, execution profiling shows that most of the data could be encoded on shorter bitwidth without any loss of precision.

Table 2 summarizes the *observed* bitwidth for each of the HMM model parameters. These results show that bitwidth reduction of 30% (and up to 60%) are possible without precision loss.

In addition to the HMM parameters, we can also study the dynamic range for the scores of the Viterbi dynamic programming matrix. From the algorithm, we know that these scores have a static lower bound which is given by a constant representing $-\infty$. Its value in the HMMER package is set to an arbitrary 10 digit value, however it is possible to choose a higher value, provided it does not modify the matrix values. To obtain both these upper and lower bounds for the matrix scores, we used a set of 10^6 P7Viterbi execution traces corresponding to the match of the *SwissProt* sequence database against a significant subset of the *Pfam* database. The results obtained in this way are presented in Table 3, and show that using 20 bits encoding preserves the original algorithm accuracy.

5.2 Measuring the Impact of Quantization

While the approach used in previous paragraph is quite effective (we obtain a reduction of 30%), we can further reduce bitwidth by using *quantization*. However, we must guarantee that this quantization will not cause false negatives due to the loss of accuracy in data and computations.

Let us write v the original 20 bit integer value, and $Q_n(v)$ its quantized counterpart that we define as:

$$v_q = Q_n(v) = \left\lceil \frac{v}{2^n} \right\rceil . \tag{16}$$

Table 2 Observed bitwidth for HMM parameters in the Pfam database.

Parameter	msc [] []	isc [] []	esc [] [] , bsc [] []	tsc [] []
Range	$[-1.510^4, 10^4]$	$[-10^3, 500]$	$[-1.510^6, 0]$	$[-1.510^4, 5.10^3]$
Bitwidth	15	11	20	14

Table 3 Observed bitwidth for dynamic programming matrix.

Variable	mmx [] []	imx [] []	dmx [] []	xmx [] []
Range	$[-10^5, 5.10^3]$	$[-10^5, 5.10^3]$	$[-10^5, 5.10^3]$	$[-10^5, 5.10^3]$
Bitwidth	20	20	20	20

The inverse quantization can in turn be written as :

$$Q_n^{-1}(v_q) = 2^n v_q . \quad (17)$$

This quantization scheme satisfies $Q^{-1}(Q(v)) \geq v$ for any v . Thus we can observe that for all pair v_1, v_2 of values, we have:

$$\begin{aligned} Q_n^{-1}(\max(Q_n(v_1), Q_n(v_2))) &\geq \max(v_1, v_2), \\ Q_n^{-1}(Q_n(v_1) + Q_n(v_2)) &\geq v_1 + v_2 . \end{aligned} \quad (18)$$

We also know that the `P7Viterbi` routine is entirely based on these two operations, It follows that the similarity score sc_Q obtained after running the algorithm in reduced precision is such that $Q_n^{-1}(Q(sc)) \geq sc$, where sc is the score obtained by running the algorithm in full precision.

As a consequence, a sequence that would have caused a match in 21 bit integer arithmetic will necessarily cause a match even though intermediate computations use quantized data. We can hence reduce the precision without the risk of generating false negatives.

5.3 The Performance/Selectivity Trade Off

Reducing bitwidth through quantization helps increasing the performance of our hardware implementation (better functional density, lower clock period). On the other hand, even though it does not induces *false negatives*, this loss of precision increases the number of *false positives* and decreases the *reject rate* of the accelerator (we define the reject rate as the probability for a sequence to be filtered out by the accelerator).

Figure 11 shows how this loss of precision impacts the reject rate of the scoring Viterbi routine for various Quantization schemes (we used HMMER defaults scoring threshold values).

We can observe that, for the default cut-off value of HMMER, the quantization schemes with bitwidth reductions greater than 8 bits strongly impacts the reject rate (only a sequence out of two is filtered out). On the other hand, the reject rate remains relatively unaffected for bitwidth reductions below 6 bits. We can therefore implement the algorithm using 15 bit

wordlength arithmetic, without significantly degrading the *averagereject-rate*.

6 Experimental Results and Performance Analysis

In this section we provide a quantitative analysis of the characteristics of FPGA implementations of our accelerator along with some performance estimates.

6.1 Implementing the Accelerator on FPGA

Section 4 has shown that the hardware resource usage (in terms of logic cells and memory) of our hardware accelerator is highly dependent on the size M of the HMM model at hand, and on design parameters such as the pipeline level λ , on the partitioning factor σ and on the chosen bitwidth W .

In order to have a more quantitative view of this resource cost, we implemented on a Xilinx Spartan3-4000 FPGA several configurations of the *complete* `P7Viterbi` kernel (including control), each one with a different set of parameters. All these configurations have been chosen so as to maximise the number of processing elements that could fit the target FPGA device for given set of values for M and λ , in all cases, the chosen quantization is Q_6 which correspond to a datapath operating on $W = 15$ bit wordlength. To do so we designed a parametrized generation tool in

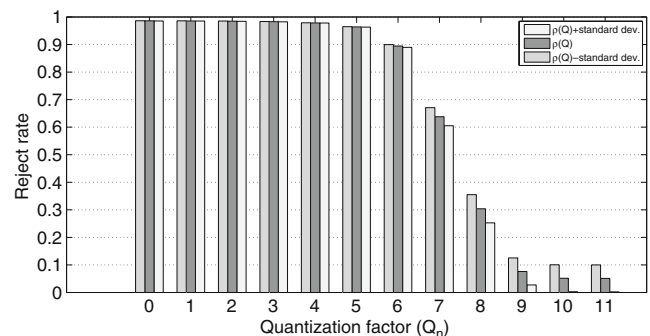


Figure 11 Average reject rate (along with standard deviation) of the Viterbi algorithm for Quantizations Q_0 to Q_{11} .

Table 4 Resource usage and performance (for a Xilinx Spartan3-4000) of the best wavefront schedule implementations for a set of given motif sizes.

M	N_{PE}	BRAM	Slices	f_{max}	GCUP/s	Speedup
60	24	96/100%	20,784/72%	40	960	38
120	16	96/100%	15,532/55%	40	640	25.5
190	9	90/94%	9,750/30%	40	360	14.5
250	9	90/94%	11,690/40%	40	360	14.5
380	4	80/85%	6,760/20%	40	160	6.5
500	4	80/85%	7,660/25%	40	160	6.5
600	4	88/92%	6,668/24%	40	160	6.5

Java which generates synthesizable VHDL (parameters include σ, λ, M and bitwidth W).

The results corresponding to all these configurations (after Place and Route) are summarized in Table 5. They show that for larger values of M , the limiting factor is the number of embedded memory blocks available on the target FPGA device (BRAM column), as a consequence, the number of processing elements that can fit in the device is limited.

For a fair comparison, we also implemented the architecture corresponding to the *wave-front schedule*. This architecture was simply obtained by generating an instance of our architecture in which we set $\sigma = M$ and $\lambda = 1$ and which was then replicated N_{PE} times. These results are given in Table 4, and show that using our *partitioned pipelined schedule* allows for a larger number of processing elements to fit on the device, especially for larger values of M (i.e. larger HMM sizes).

6.2 Performance Analysis

Performance figure for `hmmsearch` are generally measured in Million Cells Update per Second (MCUP/s), where a cell update corresponds to an iteration of the `P7Viterbi` algorithm. In our implementation, each PE performs one iteration per cycle, with clock frequencies ranging from 40 to 60 MHz. This is to be compared to a reported software performance of 24 MCUP/s on a Intel P4 CPU (see [15]). The last two

columns of Tables 4 and 5 give peak performance estimates for our accelerator (we assume that the hardware is clocked at its maximum clock frequency) and corresponding speedup over the software realization.

However, these performance gain should be balanced by the fact that even though it is possible to use a HMM profile of size M with an architecture which accommodates a maximum model size of M_{max} , only a fraction M/M_{max} of the computations actually contributes to the matching score. For example for $M = 50$ and $M_{max} = 64$, the sustained performance splis to only 78% that of peak performance.

This situation favors the use of reconfigurability: for a given value of M , we can choose among a library of preexisting bitstream the configuration which offers optimal performance for that very specific value of M . Of course, such a strategy induces a hardware reconfiguration overhead, however, given current reconfiguration latencies (125 ms for a Spartan3-1000 through USB-2) its impact on overall performance is very limited. On the other hand, the fact that M_{max} is constrained by $M_{max} = \sigma N_{PE}$ strongly restricts the number of possible configurations, and often prevents to find a perfect match for M_{max} . To illustrate this side effect, we estimated the optimal sustained performance that could be achieved by our accelerator for values of M in the range [50, 640], for a clock speed of 33 MHz. The results, given in Fig. 12, indicate that sustained performance is significantly impacted by this phenomena, especially for smaller values of M .

Table 5 Resource usage and performance (for a Xilinx Spartan3-4000) for various motif sizes, and for a Q_6 (15 bits) quantification scheme.

M	σ	N_{PE}	λ	BRAM	Slices	f_{max}	GCUP/s	Speedup
60	2	30	3	60/63%	16,453/59%	60	1.8	71
120	4	30	3	90/93%	16,534/59%	55	1.7	68
190	6	32	2	96/100%	18,640/67%	55	1.8	70
250	8	32	2	96/100%	18,127/65%	55	1.8	70
380	11	32	1	96/100%	20,682/74%	40	1.28	51
500	16	32	1	96/100%	20,171/72%	40	1.28	51
600	38	16	1	96/100%	12,844/46%	40	0.64	25

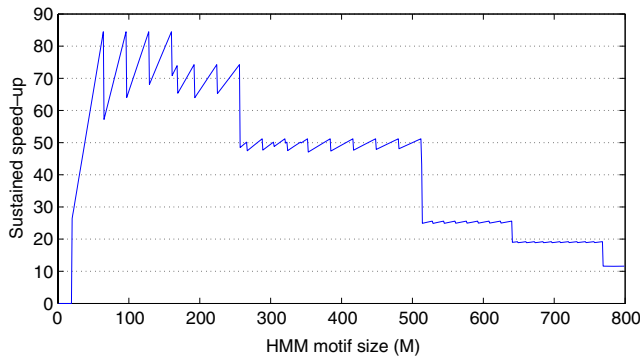


Figure 12 Sustained speedup as a function of M on a Spartan3-4000 FPGA.

represents the most time consuming kernel of the `hmmsearch` application. This parallelization scheme is based on the polyhedral model and allowed us to derive a simple yet flexible parallel architecture for accelerating the `hmmsearch` program. In order to further improve the efficiency of this architecture, we also provided an in depth analysis of the impact of data quantization on the algorithm Quality of Results (QoR). The corresponding hardware was then mapped to FPGA technology to obtain realistic performance estimates, which show that speedup between 10 and 80 can be achieved on a mid-end FPGA depending on the HMM model size at hand.

7 Conclusion

In this paper, we have presented an original parallelization scheme for the `P7Viterbi` algorithm, which

Appendix

Table 6 System of Uniform Recurrent Equations (SURE) for the `P7Viterbi` loop nest. M_{\max} is the maximum HMM model size, M is the size of the HMM model at hand, and L is the sequence length. \mathcal{D} is the domain $\{(i, k) \mid 1 \leq i \leq L, 1 \leq k \leq M\}$.

$s_{i,k}$	$= \begin{cases} dsq_i & \text{if } k = 1 \\ s_{i,k-1} & \text{otherwise} \end{cases}$	$(i, k) \in \mathcal{D}$
$tmp0_{i,k}$	$= \max_{-\infty} (xmb_{i-1,M} + BSC_k, mmx_{k-1,i} + TMM_k)$	$(i, k) \in \mathcal{D}$
$tmp1_{i,k}$	$= \max_{-\infty} (imx_{k-1,i-1} + TIM_k, dmx_{k-1,i-1} + TDM_k)$	$(i, k) \in \mathcal{D}$
$tmp2_{i,k}$	$= \begin{cases} MSC[s_{i,k}][k] & \text{elseif } MSC[s_{i,k}][k] = -\infty \\ \max(mmx_{i,k}^0, mmx_{i,k}^1) + MSC[s_{i,k}][k] & \text{elseif } MSC[s_{i,k}][k] \neq -\infty \end{cases}$	$(i, k) \in \mathcal{D}$
$mmx_{i,k}$	$= \begin{cases} mmx_{i,k} & \text{if } k > M \\ mmx_{i,k-1} & \text{if } k \geq M \end{cases}$	$(i, k) \in \mathcal{D}$
$dmx_{i,k}$	$= \max(-\infty, mmx_{k-1,i} + TMD_k, dmx_{k-1,i} + TDD_k)$	$(i, k) \in \mathcal{D}$
$imx_{i,k}^0$	$= \max(-\infty, mmx_{k,i-1}^b + TMI_k, dmx_{k,i-1} + TII_k)$	$(i, k) \in \mathcal{D}$
$imx_{i,k}^1$	$= \begin{cases} ISC[s_{i,k}][k] & \text{if } ISC[s_{i,k}][k] = -\infty \\ imx_{i,k}^0 + isc[s_{i,k}][k] & \text{if } ISC[s_{i,k}][k] \neq -\infty \end{cases}$	$(i, k) \in \mathcal{D}$
$imx_{i,k}^2$	$= \begin{cases} imx_{i,k}^1 & k > M \\ imx_{i,k-1} & k \leq M \\ -\infty & s_{i,k} = EOS \end{cases}$	$(i, k) \in \mathcal{D}$
$xme_{i,k}$	$= \max(mmx_{k,i}^b + esc_k, xme_{k-1,i})$	$(i, k) \in \mathcal{D}$
xmn_i	$= \max(xmn_{i-1} + XTN_LOOP, xme_{k-1,i})$	$1 \leq i \leq L$
xmj_i	$= \max(-\infty, xmj_{i-1} + XTJ_LOOP, xme_{i,M} + XTE_LOOP)$	$1 \leq i \leq L$
xmb_i	$= \max(-\infty, xmn_i + XTN_MOVE, xmj_i + XTJ_MOVE)$	$1 \leq i \leq L$
xmc_i	$= \max(-\infty, xmc_{i-1} + XTC_LOOP, xme_{i,M} + XTE_LOOP)$	$1 \leq i \leq L$

Table 7 Original source code for P7Viterbi routine.

```

1  int P7Viterbi(unsigned char *dsq) {
2      int i,k;
3      for (k = 1; k <= M; k++) {
4          mmx[0][k]=-INFTY;
5          imx[0][k]=-INFTY;
6          dmx[0][k]=-INFTY;
7      }
8      for (i = 1; i <= L; i++) {
9          for (k = 1; k <= M; k++) {
10             mmx[i][k] = max(-INFTY,XMB[i-1] + BSC[k],
11                             mmx[i-1][k-1] + TMM[k-1],
12                             imx[i-1][k-1] + TIM[k-1],
13                             dmx[i-1][k-1] + TDM[k-1]);
14             if (MSC[dsq[i]][k]) != -INFTY) {
15                 mmx[i][k] = mmx[i][k] + MSC[dsq[i]][k];
16             } else {
17                 mmx[i][k] = -INFTY;
18             }
19
20             xme[i][k] = max(-INFTY,mmx[i][k] + ESC[k],
21                             xme[i][k-1]);
22
23             dmx[i][k] = max(-INFTY,mmx[i][k-1] + TMD[k-1],
24                             dmx[i][k-1] + TDD[k-1]);
25             if (k < M) {
26                 imx[i][k] = max(-INFTY,mmx[i-1][k] + TMI[k],
27                                 imx[i-1][k] + TII[k]);
28
29                 if (ISC[dsq[i]][k] != -INFTY)
30                     imx[i][k] += imx[i][k]+ISC[dsq[i]][k];
31             } else {
32                 imx[i][k] = -INFTY;
33             }
34         }
35     }
36     xmn[i]= max(-INFTY,xmn[i-1] + XTN_LOOP);
37     xmj[i]= max(-INFTY,xme[i][M]+ XTE_LOOP,
38                 xmj[i-1] + XTJ_LOOP);
39     xmb[i]= max(-INFTY,xmn[i] + XTN_MOVE,
40                 xmj[i] + XTJ_MOVE);
41     xmc[i]= max(-INFTY,xmc[i-1] + XTC_LOOP,
42                 xme[i] + XTE_MOVE, )
43 }
44 return xmc[L] + XTC_MOVE;
45 }

```

References

- Altschul, S., Madden, T., Schffer, A., Zhang, J., Zhang, Z., Miller, W., et al. (1997). Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25, 3899–3402.
- Bateman, A., Birney, E., Cerruti, L., Durbin, R., Eddy, S. R., et al. (2002). The Pfam protein families database. *Nucleic Acids Research*, 30(1), 276–280.
- Darte, A., Schreiber, R., Rau, B. R., & Vivien, F. (2002). Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems*, 7(1), 159–172.
- Derrien, S., Rajopadhye, S., & Sur-Kolay, S. (2001). Combining instruction and loop level parallelism for array synthesis on FPGAs. In *International symposium on system synthesis (ISSS'01)*, Montreal.
- Eddy, S. (2004). *Sequence analysis using profile hidden Markov models*. Technical report, Washington University at Saint Louis.
- Feautrier, P. (1992). Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6), 389–420.
- Guillou, A. C., Quinton, P., & Risset, T. (2003). Hardware synthesis for multi-dimensional time. In *14th IEEE international conference on application-specific systems, architectures and processors (ASAP'03)*.
- Guyetant, S., Giraud, M., L'Hours, L., Derrien, S., Rubini, S., Lavenier, D., et al. (2005). Cluster of re-configurable nodes for scanning large genomic banks. *Parallel Computing*, 31(1), 73–96.
- Horn, D. R., Houston, M., & Hanrahan, P. (2005). Claw-HMMER: A streaming HMMer-search implementation. In *SC'05 : Proceedings of the 2005 ACM/IEEE conference on supercomputing*.
- Krogh, A., Brown, M., Mian, I. S., Sjölander, K., & Haussler, D. (1994). Hidden markov models in computational biology: Applications to protein modeling. *Journal Molecular Biology*, 235, 1501–1531.
- Lavenier, D., Georges, G., & Liu, X. (2007). A reconfigurable index FLASH memory tailored to seed-based genomic sequence comparison algorithms. *Journal of VLSI Signal Processing System*, 48(3), 255–269.
- Lindahl, E. (2005). HMMer altivec implementation. <http://lindahl.sbc.su.se/software/altivec/altivec-hmmmer-version-2.html>.
- Maddimsetty, R. P., Buhler, J., Chamberlain, R. D., Franklin, M. A., Harris, B. (2006). Accelerator design for protein sequence HMM search'. In *Proceedings of the ACM international conference on supercomputing*. Cairns, Australia: ACM.
- Moldovan, D. I., & Fortes, J. A. B. (1986). Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, 35(1), 1–12.
- Oliver, T., Schmidt, B., Jakop, Y., & Maskell, D. L. (2006). Accelerating the viterbi algorithm for profile hidden markov models using reconfigurable hardware. In *International conference on computational science*.
- Oliver, T., Schmidt, B., & Maskell, D. (2005a). Hyper customized processors for bio-sequence database scanning on FPGAs. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays* (pp. 229–237). New York, NY, USA: ACM.
- Oliver, T., Schmidt, B., Nathan, D., Clemens, R., & Maskell, D. (2005b). Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW. *Bioinformatics*, 21(16), 3431–3432.
- Oliver, T., Yeow, L. Y., & Schmidt, B. (2007). High performance database searching with HMMer on FPGAs. In *HiCOMB 2007, sixth IEEE international workshop on high performance computational biology*.
- Quinton, P. (1984). Automatic synthesis of systolic arrays from recurrent uniform equations. In *International conference on computer architecture*.
- Walters, J. P., Qudah, B., & Chaudhary, V. (2006). Accelerating the HMMER sequence analysis suite using conventional processors. In *AINA '06: Proceedings of the 20th international conference on advanced information networking and applications—Volume 1 (AINA'06)*.
- Wun, B., Buhler, J., & Crowley, P. (2005). Exploiting coarse-grained parallelism to accelerate protein motif finding with a network processor. In *PACT '05: Proceedings of the 14th international conference on parallel architectures and compilation techniques*.



Steven Derrien obtained his PhD from University of Rennes 1 in 2003, and is now assistant professor at University of Rennes 1. He is also a member of the Cairn research group at IRISA. his research interest include high level synthesis, loop parallelization, and reconfigurable systems design.



Patrice Quinton obtained a degree of Engineer in Computer Science of ENSIMAG (Grenoble, France), in 1972, and a These d'Etat in Mathematics of the University of Rennes (France) in 1980. He has been Directeur de Recherches of the CNRS, and head of the VLSI Parallel Architectures group of IRISA in Rennes between 1982 and 1997, and since then, he is professor of the University of Rennes 1. Patrice Quinton is currently deputy director of the brittany branch of Ecole Normale Supérieure of Cachan and member of the Cairn research group at IRISA, Rennes. His interests include parallel architectures, VLSI, systolic arrays, computer aided design and sensor networks. Patrice Quinton is co-author of one book, and author and co-author of about one hundred journal papers, international conference communications or book chapters.