# Accelerating HMMer on FPGAs Using Systolic Array Based Architecture

Yanteng Sun[1], Peng Li[2], Guochang Gu[1], Yuan Wen[1], Yuan Liu[2], Dong Liu[2]

[1]*College of Computer Science and Technology, Harbin Engineering University*
[2]*Intel China Research Center*
{*sunyanteng, guguochang, wenyuan*}*@hrbeu.edu.cn*, {*peng.p.li, yuan.y.liu, dong.liu*}*@intel.com*

## Abstract

*HMMer is a widely-used bioinformatics software package that uses profile HMMs (Hidden Markov Models) to model the primary structure consensus of a family of protein or nucleic acid sequences. However, with the rapid growth of both sequence and model databases, it is more and more time-consuming to run HMMer on traditional computer architecture. In this paper, the computation kernel of HMMer, P7Viterbi, is selected to be accelerated by FPGA. There is an infrequent feedback loop in P7Viterbi to update the value of beginning state (B state), which limits further parallelization. Previous work either ignored the feedback loop or serialized the process, leading to loss of either precision or efficiency. Our proposed syslolic array based architecture with a parallel data providing unit can exploit maximum parallelism of the full version of P7Viterbi. The proposed architecture speculatively runs with fully parallelism assuming that the feedback loop does not take place. If the rare feedback case actually occurs, a rollback mechanism is used to ensure correctness. Results show that by using Xilinx Virtex-5 110T FPGA, the proposed architecture with 20 PEs can achieve about a 56.8 times speedup compared with that of Intel Core2 Duo 2.33GHz CPU.*

## 1. Introduction

HMMer is a widely used open-source implementation of profile HMM (Hidden Markov Models), which searches for patterns in protein or nucleic acid sequences in bioinformatics. Profile HMMs [9] [10] are statistical models of multiple sequence alignments. They capture position-specific information about the degree of conservation in the multiple alignments, and the varying degree to which gaps and insertions are permitted. A profile HMM can be used to model a certain protein family. HMMer is composed of several programs that perform different tasks to facilitate protein sequence analysis. *Hmmpfam* and *hmmsearch* are the most frequently used and most time consuming ones. *Hmmpfam* searches an HMM database for matches to a query protein sequence. *Hmmsearch* searches a sequence protein database for matches to an HMM. With the rapid growing of sequences and model databases, it is more and more time-consuming to run HMMer on traditional computer architecture.

SIMD (Single Instruction Multiple Data) extensions on modern and MPI implementation on multiprocessor systems has been used to accelerate HMMer on conventional processers [4] [7] [12]. HMMer has also been accelerated on other special purpose processor such as GPU (Graphic Processing Unit) [3] and network processor[2].

With the development of modern FPGA (Field Programmable Gate Array) technology, more and more works have been done by implementing HMMer in parallel on FPGAs. Many solutions [1] [11] [15] are based on the simplified HMM without a rare feedback loop in HMM, while other implementations accelerates full Plan7 HMM [14] [13]. FPGA and MPI are combined into an MPI-HMMER-Boost to achieve a better speedup [8].

In this paper, we proposed a systolic-array-based reconfigurable architecture with a parallel data providing unit and an auto recalculation unit to exploit the maximum parallelism of the Plan7 HMM. The proposed architecture speculatively runs with fully parallelism assuming that the feedback loop does not take place. When the rare feedback case actually occurs, a rollback mechanism is used to ensure correctness.

The paper is organized as follows. In Section 2 we introduce the algorithm of *P7Viterbi* in HMM application. In Section 3 we analyze the data dependency of the algorithm and introduce our system architecture. In Section 4 we explore some special issues that occur when implementing the design into FPGAs. In Section 5 we present the performance results with the various resource constraints. In Section

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

6 we take a brief look at the related works in this field. Finally, in Section 7 we summarize our work.

## 2. Algorithm description

Through profiling, we found that over 90% run time of *hmmsearch* is spent on function *P7Viterbi*, which uses Viterbi algorithm to solve Plan7 HMMs. The architecture of a Plan7 HMM (length=4) is shown in Figure **1**. State B and E stand for the beginning and end of the model, while state M, I, and D denote the action of match, insertion, and deletion, respectively. State S, N, C, T, and J are special states which control the algorithm-dependent feature of the model: for example, how likely the model is to generate various sorts of local or multi-hit alignments. Note that there is a feedback loop from state E, state J to state B, however, it rarely changes the value of state B. In section 3.5, we will specify how to this rare feedback loop is supported in our architecture. The pseudo-code of *P7Viterbi* is shown in Algorithm 1. *L* and *H* are the length of the input sequence and the model. In *P7Viterbi*, Plan7 HMM is used against each element of the input sequence, with all states calculated.

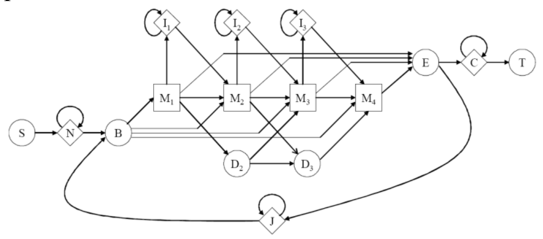

**Figure 1: Plan7 HMM of length 4 [14]**

**Algorithm 1：P7Viterbi**

```
for i=1 to L
    for k=1 to H
        calculate state M(i,k)
        calculate state D(i,k)
        calculate state I(i,k)
    end k
    calculate state N(i)
    for k=1 to H
        calculate state E(i)
    end k
    calculate state J(i)
    calculate state B(i)
    calculate state C(i)
end i
```

For convenience, we define a set of M, I, and D states with the same index (*i, k*) as a node, and a group of nodes and other states (S, N, B, J, E, C, and T) with the same index (*i*) as a layer. *P7Viterbi* calculates these states layer by layer so as to find the best match. In each layer, the value of states is calculated in order that shown in Figure **1**.

Algorithm 2 shows the details of each state calculation. *S* denotes the input sequence. *Tr(s1,s2)* is the transition score of moving from one state (s1) to another (s2). The emission score of a amino acid (*S[i]*) when being emitted at a state (s1) is in *e(s1,S[i])*. The *–INFTY* denotes the minimum value of the model.

**Algorithm 2: Calculation of states**

$$M(i,k)=\max\begin{cases} e(M_k,S[i])+\max\begin{cases} M(i-1,k-1)+tr(M_{k-1},M_k)\\ I(i-1,k-1)+tr(I_{k-1},M_k)\\ D(i-1,k-1)+tr(D_{k-1},M_k)\\ B(i-1)+tr(B,M_k)\end{cases}\\ -INFTY\end{cases}$$

$$D(i,k)=\max\begin{cases} D(i,k-1)+tr(D_{k-1},D_k)\\ M(i-1,k)+tr(M_k,I_k)\\ -INFTY\end{cases}$$

$$I(i,k)=\max\begin{cases} e(I_k,S[i])+\max\begin{cases} M(i-1,j)+tr(M_k,I_k)\\ I(i-1,k)+tr(I_k,I_k)\end{cases}\\ -INFTY\end{cases}$$

$$N(i)=\max\begin{cases} N(i-1)+tr(N,N)\\ -INFTY\end{cases}$$

$$E(i)=\max\begin{cases} M(i,k)+tr(M_k,E)\\ -INFTY\end{cases}$$

$$J(i)=\max\begin{cases} J(i-1)+tr(J,J)\\ E(i)+tr(E,J)\\ -INFTY\end{cases}$$

$$B(i)=\max\begin{cases} N(i)+tr(N,B)\\ J(i)+tr(J,B)\\ -INFTY\end{cases}$$

$$C(i)=\max\begin{cases} C(i-1)+tr(C,C)\\ E(i)+tr(E,C)\\ -INFTY\end{cases}$$

## 3. System architecture and design

In this section, we first analyze the data dependency in *P7Viterbi* algorithm (section 3.1). Based on the dependency, a systolic-array-based architecture is proposed (section 3.2) with a parallel data providing unit (section 3.3) and an auto recalculation unit (section 3.5). Section 3.6 describes the whole system architecture.

### 3.1. Data dependency analysis

Data dependency analysis is the first step to exploit full parallelism in application using reconfigurable architecture. Figure **2** shows the data dependency of one node with its neighbor nodes without the feedback loop. The "*" in the figure denotes any one of the M, I, and D states. From the figure, we can see that the input of one node comes from the preceding nodes of the same layer and from the nodes of the former layer. The input can be divided into 4 groups. Since the input sequence *S[i]* is different layer by layer, data from group *c* is provided separately from memory (details to be found in 3.3). Data dependency analysis shows that systolic-array-based architecture is a good choice for the application.

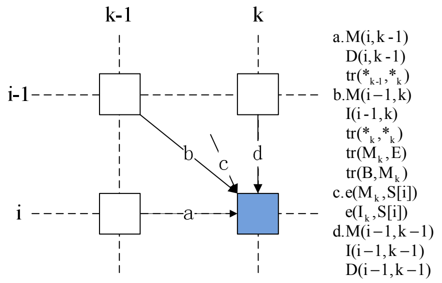
k-1
k
i-1
i
a
b
c
d
a.M(i,k-1) D(i,k-1) tr(*k-1,*k)
b.M(i-1,k) I(i-1,k) tr(*k,*k) tr(Mk,E) tr(B,Mk)
c.e(Mk,S[i]) e(Ik,S[i])
d.M(i-1,k-1) I(i-1,k-1) D(i-1,k-1)

**Figure 2: Data dependency without feedback loop**

## 3.2. Systolic-Array-Based architecture

Systolic arrays [4] [6] have been widely used to accelerate bioinformatics applications recently. A systolic array is composed of matrix-like rows of data processing unit, which transfers processed data only to its neighbors. The data processing units are of the same construct and perform the same operations on the data they receive. A traditional systolic array can not properly handle the feedback loop and emission scores described above. A parallel data providing unit (described in section 3.4) and auto recalculation unit (described in section 3.5) is used to handle these special cases.

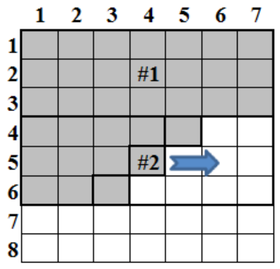

**Figure 3: Systolic array work flow**

A systolic-array-based architecture is used to accelerate *P7Viterbi* algorithm, whose work flow is shown in Figure **3**. Though data dependency analysis above, we found that no dependency between anti-diagonal nodes. So N processing elements (N=3 in the figure) are used to concurrently calculate the anti-diagonal nodes, from left to right, then from top to bottom.

The data path of the architecture is shown in Figure 4(a), in which *a. b, c,* and *d* represent the four groups of data mentioned above The data flow is shown in Figure 4(b). Data group *a* comes from the output of current PE last cycle, while data group *b* comes from the output of neighbor PE last cycle. Data group *c* is provided from parallel providing unit described next session. Data group *d* comes from the output of neighbor PE the cycle before last, which is buffered in our architecture.

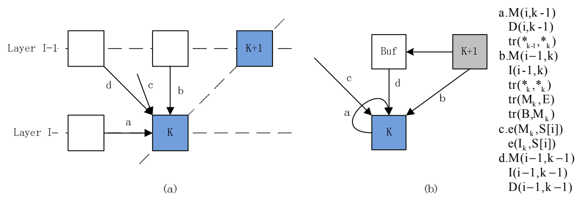
Layer I-1
Layer I
K+1
K
Buf
a
b
c
d
(a)
(b)
a.M(i,k-1) D(i,k-1) tr(*k-1,*k)
b.M(i-1,k) I(i-1,k) tr(*k,*k) tr(Mk,E) tr(B,Mk)
c.e(Mk,S[i]) e(Ik,S[i])
d.M(i-1,k-1) I(i-1,k-1) D(i-1,k-1)

**Figure 4: Systolic data flow**

## 3.3. Parallel data providing unit

Data group c mentioned above includes the emission scores needed independently by each layer, indexed by *S[i]* and *k*. Since *S[i]* must be one of the 24 different kinds of proteins, 24 blocks of RAM are used to store the emission score data. In Figure **5**, 3 PEs are calculating the result of the input sequence "114" in parallel. The fourth and fifth layer has the same input, so *e(s4,1)* and *e(s5,1)* stored in the same RAM are needed concurrently, which will cause structural conflict. The simplest way to solve this conflict is to duplicate data to N copies. However, this will exhaust the limited on-chip memory of FPGAs. We propose a parallel data providing unit to provide the necessary data within on-chip resource constrains.

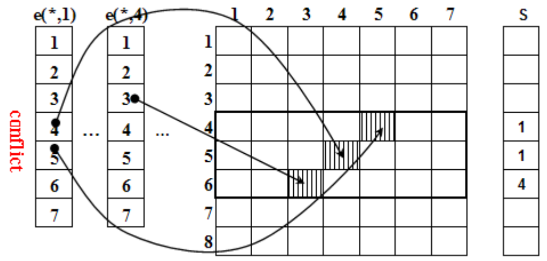

**Figure 5: Emission score input**

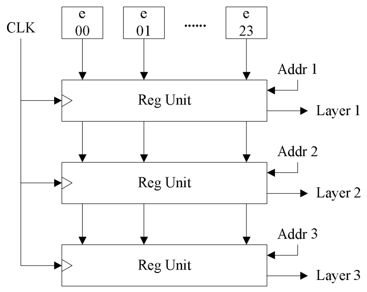

**Figure 6: 3-way Data Providing Unit**

Figure 6 shows our N-way (N=3 in the figure) parallel data providing unit. For every cycle, the first "Reg Unit" in the figure reads all input data indexed by *S[i]*, while the data buffered previous go through the register network step by step.

Using this parallel data providing unit, N copies of input data can be accessed concurrently with 1/24 area consumption compared with naïve implementation.

## 3.4. Calculation Partition

Figure **7Error! Reference source not found.** shows how a task with 8 input sequence elements is partitioned among 3 PEs. The block with vertical shadow stands for the PE that is working while the block with horizontal shadow stands for a PE that is disabled. In the last round of systolic operation, unnecessary PEs are disabled by control unit. This technique (programmable calculation) is also used in the auto recalculation (Section 3.5).

A Ping-Pong RAM is used to fill the gap between two divided systolic operations. The first systolic operation's data are provided by the RAM of each variable, and the outputs are buffered by the Ping-Pong RAM. When starting another round of systolic operation, the Ping-Pong RAM provides the inputs needed by the systolic operation and buffers the outputs that are the inputs of the next.

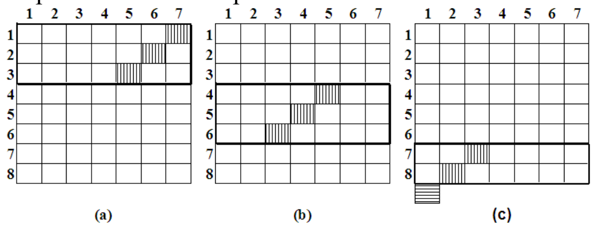

**Figure 7: Calculation partition**

## 3.5. Auto Recalculation unit

In *P7viterbi* algorithm, the value of state B is either from state N or from state J. State J is calculated from state E, which is dependent from the whole layer. This is against the assumption we just made above on no dependency between anti-diagonal nodes. Fortunately, the value of state B seldom comes from state J. So we speculatively assume that there is no feedback loop from state J to state B and use systolic-array-based architecture mentioned above to calculate all the states. After calculating the whole layer, values are checked to see whether the feedback loop actually occured. If it did occur on rare cases, an auto recalculation unit is used to recalculation the correct values again.

Our design can update the B state after a systolic operation, which also means that only the last layer of a systolic operation can update the B state. In others words, there is a feedback loop between two systolic operations but not between two layers. Then the problem changes to what should we do when the feedback loop take effect just inside a systolic operation. An auto recalculation unit to handle the feedback loop when it indeed needs to update the B state inside a systolic operation, and our design will detect whether this takes place. If that happened, a recalculation is launched with the layer that updates the B state as the last layer of this systolic operation.

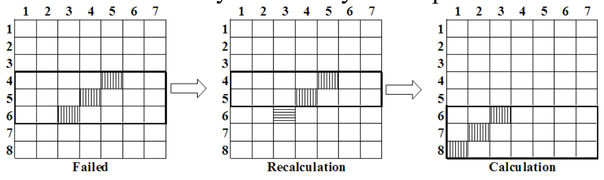

**Figure 8 : Auto recalculation**

As shown in Figure 8, if the fifth layer needs to update the B state, then the systolic operation that calculates this group should be redone, because the calculation of sixth layer uses an old value of B state but not the latest one. Therefore, we need to calculate row 4 and 5 at one time and update the B state, then we need to start another systolic operation with the newest value. In this recalculation case, the working number of PEs should also be set to 2. As we can see, the programmable calculation we described in Section 3.4 also fits the need of the auto recalculation unit of the architecture.

Parallel data providing unit to work with the recalculation means the data that have been read should be provided again. Suppose there are *N* PEs, and the current systolic operation is doing on layer *l1* to *l2*, and the nth layer updates the B state, then the next *N-n* layers need to be computed at the new value, and the recalculation needs to be performed on layer *l1* to *n* and not on layer *l1* to *l2*. The next operation works on row *n* to *n+N* and then on the next group. A trace-back mechanism that can re-provide all of the data with the correct range of data is needed, according to the number of layers a systolic operation needs to work on.

Figure 9 shows the flow chart of recalculation if the PE number is *N_PE*. The *id_out* is the control parameter that is sent to the programmable calculation unit. Every PE has an id that ranges from 1 to *N_PE*. If the *id_out* is smaller than its id, then it is disabled; if it is not smaller, it deals with the data it receives. If there is no need to recalculate, the *id_out* should be set to *N_PE*, and the Ping-Pong RAM should be switched. If recalculation is needed, the PEs that need new value of the B state should be disabled, by setting the *id_out* to the id of the PE that triggers the feedback loop; and the address should be rolled back.

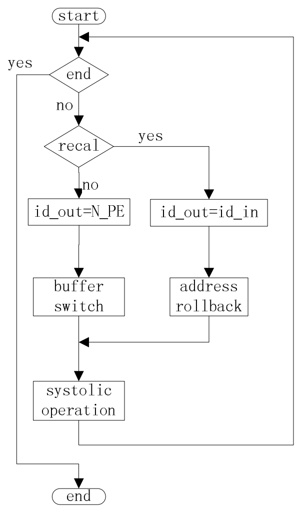

**Figure 9: Recalculation flow chart**

Figure 10 shows the architecture of the auto recalculation unit. The "ei unit" and "em unit" deal with the emission score of I state and M state respectively. The "state unit" deals with the data of states of Plan7 HMM; its main component is Ping-Pong RAM. The "special unit" deals with the data of special states that are not related to outer loop (layers), such as B, E, T, etc.

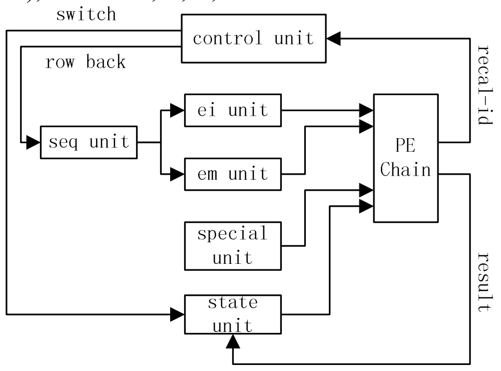

**Figure 10: Recalculation architecture**

## 3.6. System architecture

The system architecture is shown in Figure 11. IO transceivers provide data transfer from the host PC to the design, via the IO interface of the FPGA board, such as PCI Express, FSB, etc. PEH (Processing Element Header) is necessary for loading data to the systolic array from RAMs and Ping-Pong RAMs. The calculation partition (Section 3.4) is done by setting the right mode of the PEH. Arbiter logic is used to decide whether recalculation is needed. The Parallel data providing unit provides data access to emission score of I and M states as the algorithm describes.

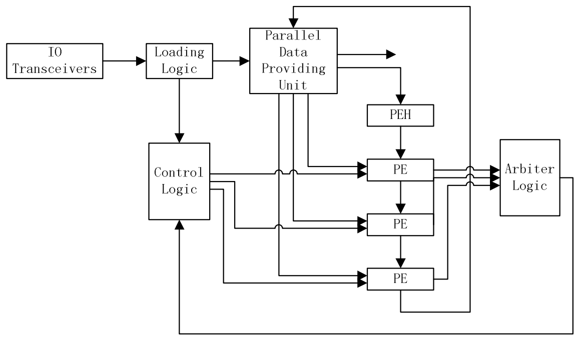

**Figure 11: System architecture**

## 4. FPGA implementation

FPGA chips have many different kinds of resources, so a parametric design is portable on FPGAs. Some measures also need to be taken to achieve a higher working frequency. There are some special instructions when implementing the design.

### 4.1. Parallelizing basic operations

As we have described, it takes several add operations and a multi-input maximization, which are done serially by the CPU, to calculate the value of one state. The sum of the values can be implemented in parallel on FPGA by generating redundant accumulators, but only if there are no data dependencies. We take the calculation of state D for example. There are two pairs of inputs shown in the figure, $D(i,k-1)$, $tr(D_{k-1},D_k)$, $M(i-1,k)$ and $tr(M_k,I_k)$. The result must be no less than the minimum value, so *–INFTY* is involved in the comparison. In this example, the accumulations can be processed separately. As shown in Figure **12**(a), the maximization of $N$ inputs can be implemented by connecting $N-1$ comparators. The comparison of two 32-bit values is very time-consuming, and it takes 2 comparisons and 2 selections, so it's not efficient to compare values in that way. *Compare pair* is an acceptable means of comparing values in parallel, and this method is used in the cache replacement unit of the CPU. As shown in Figure 12(b), each pair of three inputs is compared and the result is sent to a selector that can decode the result and choose the maximum of the three inputs. In this way, three values can be calculated in the time it takes to make one comparison. The tradeoff of this method is that more comparators are needed than are needed when the calculations are done serially. Considering that a 32-bit comparator takes only 32 LUTs (Lookup Tables) in Virtex-5 FPGAs, it is an acceptable way to speed up the design.

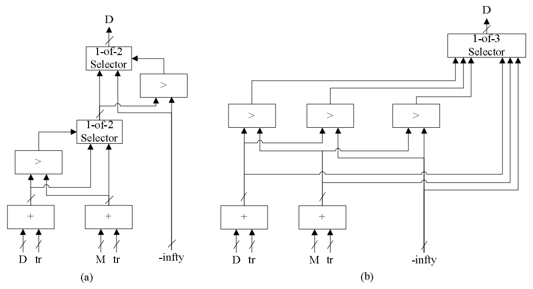

**Figure** 12**: A comparison of D State**

### 4.2. Pipelining

The compare pair reduces the time delay of the PE, but the time delay of the main data path decides the working frequency of the system. Pipelining is an effective method to cut down on the time delay. A module is designed for each state. The module (module*X*) is named by the state (*X*) it calculates, and there are eight modules in a node, as described in Algorithm 1. Using FPGA's inherent concurrency, moduleM, moduleD, moduleI, and moduleN can work in parallel. The main path of the design is from moduleM to moduleB. ModuleM has a time delay of 8.243ns, moduleE 2.531ns, and moduleN 6.211ns. ModuleJ, moduleB and moduleC are of the same structure, and the time delay is 6.441ns. As shown in Figure 13, we change moduleM, moduleN, moduleE, and moduleB into timing units. ModuleM, moduleD, moduleI and moduleN belong to stage 1, moduleB belongs to stage 3, and the remainders belong to stage 2. Now, the main path is stage 2, and the time delay is less than 10ns.

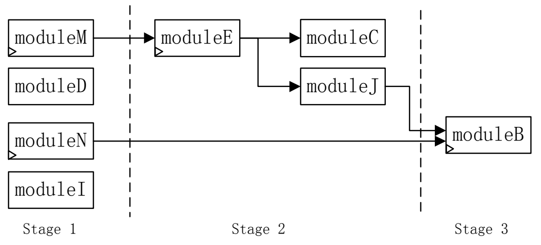

**Figure** 13**: Pipeline**

### 4.3. Parameterization

In today's chipsets such as Virtex-5, it is acceptable to generate 10 to 20 PEs. With the development of FPGA technology, we can generate more PEs to achieve a greater speed-up. Parameterizing the design makes it easy to set the number of PEs that we need. Our method is based on two magic statements: "generate" and "generic". The PE chain, control unit, and data providing unit are parameterized. Once this is done, the design can be mapped to various FPGA chipsets by modifying the parameter of the PE number in the top-level design file.

## 5. Performance evaluation

We choose Xilinx Virtex-5 110T as our experimental chipset. The design is coded in VHDL, synthesized and implemented by Xilinx ISE9.2i. The code of HMMer is taken from 456.HMMer of Spec CPU2006, which is in C language and is equal to HMMer-2.3. The code is compiled by Visual Studio .NET 2005 with an "O2" compiler flag. The Pentium® platform's CPU is the Pentium 4 3.2GHz, the memory is DDR 1G, and the hard disk is 160G SCSI (80G*2). The Intel® Core™ 2 platform's CPU is the Intel® Core™ 2 Duo 2.33GHz, the memory is DDR2 4G, and the hard disk is 160G S-ATA. They all run Windows XP SP2 with PAE (Physical Address Extension) opened.

We run HMMer at Core2 Duo with single thread. HMMer have a "HMMER_THREADS" flag for runing in parallel on multi CPUs/Cores platform. This technic can run *P7Viterbi* in parallel, but can not accelerate the excutation of a single round, so we run HMMer at single thread on Core2 Duo for compare. Table 1 shows the registers and LUTs used by different numbers of PEs; for the FPGA we used, we can utilize 25 PEs at most. The minimum period of our design is 7.68 nanoseconds, and our design can run at a maximum frequency of 130MHz after place and route post-simulation. CUPS (Cell Updates per Second) is a commonly used performance measure in computational biology. One calculation of M, I, and D states called one CPUS. The maximum CPUS performance of our design is 130MHz * 25PEs = 3.2GCUPS. The performance of *hmmsearch* is around from 24 MCUPS [15] to 55 MCUPS [1] on a Pentium4 platform. In our tests, Pentium4 is 29.8 MCUPS, and Core2 Duo is 35.5 MCUPS.

**Table** 1**: Resource utilization (5vlx110tff1136-3)**

| Number of PEs | Slice Registers | Slice LUTs |
|---|---|---|
| 5 | 19% | 19% |
| 10 | 37% | 38% |
| 15 | 55% | 56% |
| 20 | 72% | 74% |
| 25 | 90% | 93% |

No matter the software or FPGAs, the time that the *P7Viterbi* needs is related to the length of the query sequence. In our design, if the value of state B doesn't change, the time is $C \times T \times [L/N]$; otherwise, it is $C \times T \times (n + [(1 - \sum_{i=0}^{n} b_i)/N])$. "[*x*]" means taking the nearest integer greater than *x*. *C* is the number of system clocks one systolic operation needs. In this design, $C = 2 \times N + M + 12$. *N* is the number of PEs. *M* is the length of the Plan7 HMM, and *n* is the times the value of state B changed by state J. The *bi* stands for the *ith* change that happens at the *bith* PE of the PE chain. For the moment, we only measure the core executive time of FPGA design, the download time of Plan7 HMM and query sequence, and the cost of starting PCI-E transfer is not considered. In compared with that, in the software timing, we only count the executive time of that two loops described in Algorithm 1, function calling of *P7Viterbi* and memory transfer is not considered either. A full test will be launched in our further work.

The reference database provided by CPU2006 has 122564 query inputs. We run HMMer at two different platforms, and we time the *P7Viterbi* function by taking the average of 5 times of sampling.

Table **2** shows the acceleration ratio of two different scales of FPGA implementations vs. two kinds of software platforms. The input that achieves the minimum speed up is the shortest one, because the length of its query sequence is less than the number of PEs, however, it takes one systolic operation. The more PEs we have the longer one systolic operation takes, so the minimum value goes down a little with an increased number of PEs. The maximum speed-up is proportional to the number of PEs. The acceleration ratio is as much as 110 times that of the normal Pentium 4 platform; even with the newest Intel Core 2 Duo platform, the speed-up can be as much as more than 90 times.

**Table 2: Acceleration ratio**

| Number of PEs | Software Platform | Average | Minimum | Maximum |
|---|---|---|---|---|
| 10 | Core 2 Duo | 30.830 | 8.287 | 48.633 |
| 10 | Pentium 4 | 36.381 | 8.098 | 60.131 |
| 20 | Core 2 Duo | 56.837 | 7.816 | 89.802 |
| 20 | Pentium 4 | 67.098 | 7.638 | 110.072 |

Figure 14 and Figure 15 show a graph of the acceleration ratio. The X axis is the input sequence length, and the Y axis is the FPGA acceleration ratio. The last systolic operation does not run at full speed, if the remaining layers are less than the PEs. When the query sequence is small the tradeoff is obvious, so the acceleration ratio at the beginning of the figure is a little less than the average but it increases with the length of input.

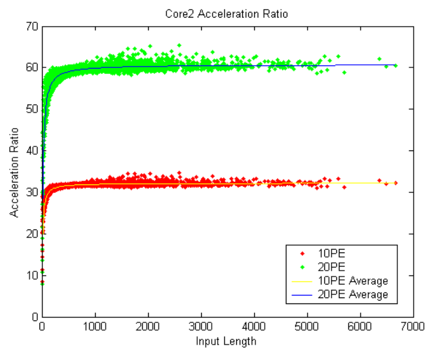

**Figure 14: Intel® Core™ 2 platform acceleration ratio**

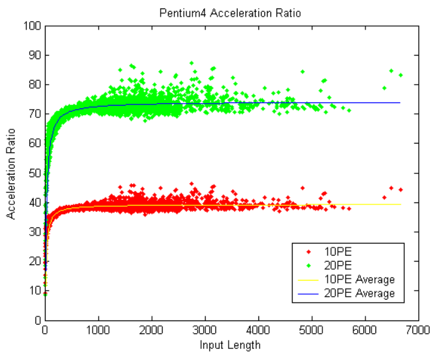

**Figure 15: Pentium 4 platform acceleration ratio**

## 6. Related work

Let us take a brief look at the work that has been done on accelerating an HMMer application by using FPGAs. Plan7 HMM has a feedback loop which makes it possible to describe a string of multiple motif instances in one protein. Many researchers eliminate the feedback loop [1] [15] [11] so as to facilitate the FPGA implementations, which leads to the lost of multi-hit alignments. Jacob, et al. also use a systolic array to parallelize the design [1], they got a throughput of 10647 MCUPS, and with the best cast a 190x speedup is achieved. The early work of Oliver, et al. [15] also uses a simplified Plan7 HMM, and it gets a maximum speed-up of 220 with 5.3GCUPS. The work centers main of the PE, and needs to make some introduction about the transmission and emission probs. Maddimsetty, et al. use a systolic array too, and it proposes two alternative ways to recover the sensitivity [11]. The performance of their work is likely of 5-20 GCUPS.

There are also some works that speed up the full Plan7 HMM. Oliver, et al. analyzes the feedback loop (J state) and find out it hinder computing one query in parallel [14]. Instead, they parallelize the design at coarse-grained level that is running several independent query sequences at one time, which also accelerates the task of search in overall running time.

The drawback to this method is that an individual query cannot be accelerated, and the coarse-grained parallelism makes the throughput of the IO system of the host computer heavier with the growing number of PEs involved. They get a 700 MCUPS on XC3S1500, but the emission storage is not introduced. Derrien, et al. [13] have the same motivation with us in that they want to accelerate an individual query, but they work out their problem in a more complex way. They get a 330 MCUPS to 660MCUPS performance. They use a polyhedral model to achieve linear space-time mappings. They then use an architecture template to make them portable, and employ idle sates to deal with Plan7 HMM of different lengths. Our approach is simpler: our design can dynamically decide the operation of processing elements (PEs) and parallelize the calculations that don't involve the feedback loop; the states number of Plan7 HMM can be assigned at start up. The work in [8] combines both MPI [7] and FPGA [14] strategies together, and the proposed multi-grained acceleration achieves a reasonable speed-up.

## 7. Conclusion

In this paper we proposed a systolic-array-based implementation of Plan7 HMM on FPGAs with a parallel data providing unit and an auto recalculation unit. With the Virtex-5 chipset we can accelerate the Plan7 HMM by 56.8 times with 20 PEs compared to an Intel Core 2 Duo 2.33GHz platform.

## 8. Acknowledgments

This research is supported by Intel-University Collaboration Program. We would also like to thank the Xilinx University Program (XUP) for the FPGA development kit and technical supports.

## 9. References

[1] Arpith C. Jacob, Joseph M. Lancaster, Jeremy D. Buhler, Roger D. Chamberlain. "Preliminary results in accelerating profile HMM search on FPGAs." *IPDPS* 2007: 1-8.

[2] B. Wun, J. Buhler, and P. Crowley. "Exploiting coarsegrained parallelism to accelerate protein motif finding with a network processor". In PACT '05: *Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[3] D.Reiter Horn, Mike Houston, Pat Hanrahan, "ClawHMMER: A Streaming HMMer-Search Implementation." *SC'05: The International Conference on High Performance Computing, Networking and Storage*, 2005.

[4] E. Lindahl. Altivec-accelerated HMM algorithms. http://lindahl.sbc.su.se/.

[5] H. T. Kung, C. E. Leiserson. "Systolic arrays for VLSI." In *Proceedings SIAM Sparse Matrix Proc.*, 1979, pp. 256-282.

[6] H. T. Kung, C. E. Leiserson. "Algorithms for VLSI Processor Arrays." *Introduction to VLSI Systems (C. A. Mead and L. A. Conway, eds.)*, chapter 8.3, pp: 271-292, Addison-Wesley, 1980.

[7] J.P. Walters, B. Qudah, V. Chaudhary, "Accelerating the HMMER Sequence Analysis Suite Using Conventional Processors." *AINA'06: Proc. Of the 20th International Conference and Advanced Information Networking and Applications – Volume 1 (AINA'06)*, pp: 289-294, IEEE Computer Society, 2006.

[8] J.P. Walters, X. Meng, V. Chaudhary, T. Oliver, Y.Y. Leow, D. Nathan, B. Schmidt, J. Landman, "MPI-HMMER-Boost: Distributed FPGA Acceleration." *The Journal of VLSI Signal Processing*, Vol. 48, No. 3. (2007), pp. 223-238.

[9] Krogh, M. Brown, I. S. Mian, K. Sjolander, D. Haussler. "Hidden Markov models in computational biology: applications to protein modeling." *Journal of Molecular Biology*, 235:1501–31, 1994.

[10] L. R. Rabiner, B. H. Juang. "An Introduction to Hidden Markov Models." *IEEE ASSP Magazine*, 3(1):4-16, January 1986.

[11] Rahul P. Maddimsetty, Jeremy Buhler, Roger D. Chamberlain, Mark A. Franklin, Brandon Harris. "Accelerator Design for Protein Sequence HMM Search." *Proc. of 20th ACM Int'l Conference on Supercomputing*, June 2006.

[12] R. Darole, J.P. Walters, V. Chaudhary, "Improving MPI-HMMER's Scalability with Parallel I/O." *IPDPS 2009*.

[13] S. Derrien, P. Quinton, "Parallelizing HMMER for Hardware Acceleration on FPGAs." *ASAP*, 2007.

[14] T. Oliver, Y.Y. Leow, B. Schmidt: "Integrating FPGA Acceleration into HMMer.", Parallel Computing, Vol. 34, No. 11, pp.681-691, 2008.

[15] Timothy F. Oliver, Bertil Schmidt, Yanto Jakop, Douglas L. Maskell: Accelerating the Viterbi Algorithm for Profile Hidden Markov Models Using Reconfigurable Hardware. International Conference on Computational Science (1) 2006: 522-529