HLS Tools for FPGA : faster development with better performances

Alexandre Cornu, Steven Derrien, Dominique Lavenier

HLS Tools for FPGA: Faster Development with Better Performance

Alexandre Cornu[1,2], Steven Derrien[2,3], and Dominique Lavenier[1,4,5]

[1] IRISA, Rennes, France
[2] INRIA, Rennes, France
[3] Université de Rennes 1, Rennes, France
[4] CNRS
[5] ENS Cachan Bretagne

Abstract. Designing FPGA-based accelerators is a difficult and time-consuming task which can be softened by the emergence of new generations of High Level Synthesis Tools. This paper describes how the ImpulseC C-to-hardware compiler tool has been used to develop efficient hardware for a known genomic sequence alignment algorithms and reports HLL designs performance outperforming traditional hand written optimized HDL implementations.

Keywords: ImpulseC, High Level Synthesis Tool, Hardware Accelerator, FPGA.

1 Introduction

FPGA density is increasing exponentially in such a way that the number of gates is approximately doubling every two years. Consequently, very complex designs can consequently be integrated into a single FPGA component, which can now be considered as high computing power accelerators. However, pushing processing into such devices leads to important development time and design reliability issues. Recently, many efforts have been done to help FPGA-targeted application designers to deal with such huge amount of resources. In particular, Electronic System Level tools provide higher levels of abstraction than traditional HDL design flow. Several High Level Languages (HLL) for modeling complex systems, and corresponding High Level Synthesis (HLS) Tools to translate HLL-designs into HDL synthesizable projects are now available.

Most of them are based on a subset of C/C++ [1] generally extended with specific types or I/O capabilities. This paper focuses on ImpulseC [3] and its associated design flow proposed by Impulse Accelerated Technologies. It also gives feedback in the context of high performance hardware accelerator design.

Experimentations have been conducted on a specific family of algorithms coming from bioinformatics and which are known to have highly efficient parallelization on FPGA. More specifically, in this work, genomic sequence comparison algorithms are considered. As a matter of fact, many efforts have been done to parallelize these algorithms, providing many optimized implementations which can be used as references [6,9,7,5].

A. Koch et al. (Eds.): ARC 2011, LNCS 6578, pp. 67–78, 2011.

In this paper, we detail how ImpulseC has been used to quickly implement parallel systolic architectures. Even if here, only a single application is considered (genomic sequence comparison), the methodology we followed can obviously be extended to many other algorithms with efficient implementation on systolic array, or more generally, implementation on massively parallel architectures. To achieve reasonable performance, standard optimizations such as loop pipelining or process splitting need however to be done. The use of HLL provides an easy way to perform these optimizations, together with a fast development cycle time. However, to obtain high performance, designers have to bypass ImpulseC restrictions and perform complex program transformations.

The experiments presented in this paper show that an HLL-designed accelerator can outperform optimized HDL designs. This can be first achieved by rapidly exploring several architectural variations without great efforts compared to HDL specifications. Second, this is also achieved by the use of high level code transformations allowing the designer to generate code which better fit to HLS tool input.

The paper is organized as follows: the first section briefly describes the HLS tool we have used : ImpulseC and gives some background on the parallelization scheme used for our target genomic sequence comparison algorithm. Section 3 presents our design strategy. Performances are finally detailed in section 4, in terms of code transformation efficiency, hardware accelerator performance, and design process.

2 HLS Tool, Algorithm and Parallelization

2.1 ImpulseC

ImpulseC is a high level language based on ANSI C. It has a few restrictions, mainly on structure and pointer usage. On the other hand, it includes libraries to define constructor functions, bit-accurate data types and communication functions.

Two levels of parallelism are available: (1) coarse grain parallelism, by implementing several ImpulseC processes that can communicate through streams, signals or registers; (2) fine grain operator parallelism, within one process or one process loop, through the use of instruction pipelining and data flow parallelism.

Each process can be set as hardware process, meaning it will be hard-wired, or as software process, meaning its sequential code will be executed on a processor. Implementation of streams between hardware and software processes are managed by specific PSP (Platform Support Package). Here, two different PSP have been used : the Nios2 softcore and the XD2000i development platform.

The development environment (IDE) is called CoDevelopper. Designer can perform software simulation, generate HDL for a specific platform through the use of ImpulseC compiler, analyze ImpulseC compiler report through the Stage Master Explorer tool or generate HDL simulation testbench with the CoValidator tool.

2.2 Algorithm

The goal of genomic sequence comparison algorithms is to identify similarities between genomic sequences for discovering functional, structural or evolutionary relationships. Similarities are detected by computing sequence alignments which generally represent the main time consuming part. A score is attached to each alignment and represents the number of matches between 2 sequences. The exact method for computing alignments between 2 sequences is based on dynamic programming (DP) algorithms. These algorithms have been abundantly described in the literature and we just give the main idea. Basically, DP algorithms consist in computing a matrix of size $N \times M$ where N and M are respectively the sizes of the genomic sequences. The following recurrent equation explains how to fill the matrix:

$$S(i,j) = max \begin{cases} S(i-1,j-1) + sub(x_i, y_j) \\ S(i-1,j) - gap\_penalty() \\ S(i,j-1) - gap\_penalty() \end{cases}$$

$1 \leq i \leq N, 1 \leq j \leq M$

x_i and y_j are respectively elements of sequences $X\{x_1 \dots x_N\}$ and $Y\{y_1 \dots y_M\}$; $sub()$ refers to a substitution cost. The final score is given by $S(N, M)$. From this basic equation many variations can be elaborated to compute different types of alignments. But, the main structure remains and the same parallelizing scheme can be applied to all.

2.3 Parallelization

Data dependencies are illustrated in figure 1. Note that computation of cells that belong to the same diagonal are independent and can be computed in the same time. This very regular and fine grained parallelization leads to a systolic array architecture: each Processing Element (PE) is affected to a sequence character X; then sequence Y crosses the array, character by character; after M steps, all $N \times M$ values have been computed. Again, a large literature is available on this topic. Interested readers can refer to the work of Lavenier[2] for a complete overview of systolic architectures for genomic sequence comparison.

Using systolic architectures requires two separate phases: a initialization step and a computational step. The first step pushes a query sequence into the systolic array in such a way that one PE store one character. The second step pushes several sequences from a bank, character by character, to the systolic array.

3 Design Methodology and Architecture Description

The overall architecture is based on very simple two-processes structure (Figure 2(1)): A first ImpulseC process (Server) generates sequences; the second process (Master) computes alignments. Both processes exchange data through

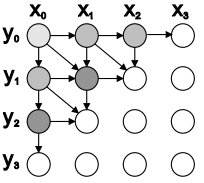

Fig. 1. Data Dependency for Similarity Matrix Computation

two streams: one carrying sequences, and the other sending back the scores related to the alignments.

The Server process has been first hardwired (release 1), then implemented on the Nios2 softcore processor (release 2, 3) and, latter on, to the XtremeData Xeon processor (cf. subsection 4.1).

The initial Master process has been split into two entities: a systolic array and a controller entity (still called Master), which has been iteratively split into many simple but optimized processes. Finally, as implementing too many small independent processes was leading to a waste of resources, PEs have been clustered. This architecture evolution is detailed in the following subsections.

3.1 Systolic Array Design

Processing Element (PE) have been designed to constitute a 1-D systolic array. Each PE is described as an independent ImpulseC process and performs computation of one column of the matrix (one cell per PE per loop iteration). The Master process feed the first PE of the array and get results from the last one (Figure 2(2)). A PE has 3 pairs of stream interfaces, allowing it to read values from previous PE and forwarding values to the next one. Streams $score_i$, $score_o$, are carrying scores from previous PE and to next PE. Both sequence stream pairs x_i, x_o and y_i, y_o are carrying sequences, character by character. For sake of clarity, two different streams are shown. Actually, a single time-sharing stream is implemented as there are no concurrent read/write operations.

To compute one cell (current score : $score_C$), one PE need to have: the upper cell's score ($score_U$), the left cell's score ($score_L$), the upper-left cell's score ($score_{UL}$), and the two characters X_C and Y_C.

Algorithm 1 describes PE behavior. This description is the input of the ImpulseC compiler to provide an FPGA implementation. Timing isn't specified and is automatically deduced by the ImpulseC compiler from data dependencies. Actually, real code is a little bit more complex, as it contains initialization (not represented here) and it considers more complex alignment computation.

3.2 Design Optimizations

This section explains the various optimizing steps which have been done to reach the final architecture.

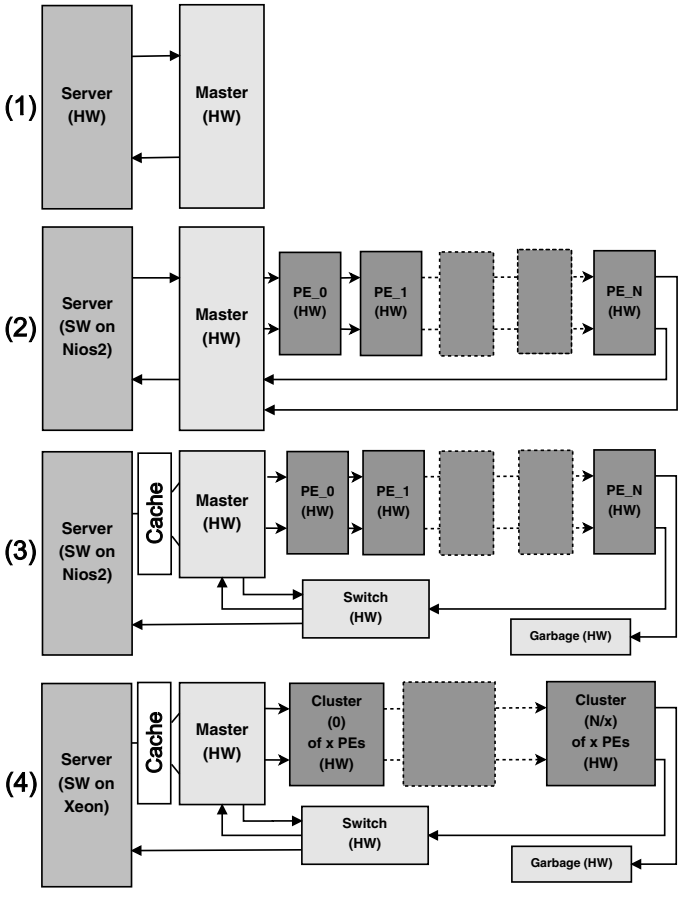

Fig. 2. Design Releases

Speeding-Up Processes. Once the overall multi-process system has been functionally validated, each process has been separately optimized. This has been done through simple source code transformations or by adding compiler directives (pragmas) such as the `CO PIPELINE` pragma, which will lead to a pipelined execution on the most inner loop. For example, in Algorithm 1, both inner loops are important loops to pipeline. In addition, code transformations may be necessary to help the ImpulseC compiler to provide better efficient scheduling. In our case, both loops have a rate of 1, meaning that one loop iteration can be started every clock cycle.

Splitting Slow Processes. In some cases, processes exhibits codes which are too complex to achieve efficient schedule or pipeline. For example, Master in figure 2(2) must perform all the following tasks: reading 32-bit data on incoming streams from host; serializing these data to push them character by character

Algorithm 1. Simple Processing Element Pseudo-Code

```
ProcessingElement(x_I, x_O, y_I, y_O, score_I, score_O)
loop
  x_C ← stream_read(x_I)
  while x_TMP ≠ EOS do
    x_TMP ← stream_read(x_I)
    stream_write(x_TMP) → x_O
  end while
  for n = 1 to N do
    y_C ← stream_read(y_I)
    stream_write(y_C) → y_O
    score_L ← stream_read(score_I)
    score_C ← compute_score( score_L, score_UL, score_U, , x_C, y_C)
    stream_write(score_C) → score_O
    score_U ← score_C
    score_UL ← score_L
  end for
end loop
```

into the systolic array; storing sequences that need to be compared to multiple sequences; controlling systolic array for different sequences; etc. To execute these tasks, the Master code uses several nested multi-conditional controlled loops.

One of the ImpulseC restrictions is that nested loops cannot be pipelined. While Master is waiting for the last result coming from the systolic array, it could have started to send the next sequence, as first PEs are ready to work. One way to overcome this issue is to merge these loops. However, this transformation leads to complex code and poor scheduling performance.

Then, splitting processes with loops that can be executed in parallel can help to improve the overall system efficiency. In the accelerator system, many processes have been created to simplify the Master process code (Figure 2(3)). A garbage process reads useless data from the systolic array. A switch process, configured by the Master, carries results either to the server software process or to the Master when needed for the next computation. A cache system reads alignment structure coming from the Server, stores sequences which need to be sent again, and does the sequence serialization. This splitting mechanism provides an efficient electronic system, minimizing process idle periods and generating simple code which is better suited for ImpulseC scheduler capabilities.

Minimizing Resources. As the systolic array represent the major resource utilization, a lot of efforts have been done to minimize individual PE and stream resources. Streams are implemented as FIFOs and have been reduced to their minimal size, allowing memory block FIFOs to be advantageously replaced by register FIFOs. PE arithmetic has been limited to 14 bits which is the theoretical upper bound of score values. Some operators have also been transformed to

simple ones when possible. All these optimizations can easily be specified at ImpulseC level and do not generate additional complexity in the design process.

3.3 Clustering PEs

Specifying one PE by one process provides efficient hardware (cf. section 4.2): the architecture is fully distributed. Signals (control and operand) remain local, leading to small fanout and easy synthesis. Nevertheless, it uses a lot of resources: each PE duplicates identical control logic and several intermediate streams.

Reducing the duplication of control logic (and streams) resources would enable a larger systolic array (section 4.2). A way to achieve this objective is to cluster PEs. At ImpulseC level, this is done by specifying the behavior of several PEs in a single process (Figure 2(4)). As an example, algorithm 2 shows the code of one cluster of 2 PEs.

Algorithm 2. Dual Processing Element Cluster Pseudo-Code

```
ProcessingElement(x_I, x_O, y_I, y_O, score_I, score_O)
loop
  x_C1 ← stream_read(x_I)
  x_C2 ← stream_read(x_I)
  while x_TMP ≠ EOS do
    x_TMP ← stream_read(x_I)
    stream_write(x_TMP) → x_O
  end while
  for n = 1 to N do
    #pragma CO PIPELINE
    y_C ← stream_read(y_I)
    score_L ← stream_read(score_I)
    score_C1 ← compute_score( score_L1, score_UL1, score_U1, x_C1, y_C1)
    score_U1 ← score_C1
    score_UL1 ← score_L1
    score_L2 ← score_C1
    y_C2 ← y_C1
    score_C2 ← compute_score( score_L2, score_UL2, score_U2, x_C2, y_C2)
    score_U2 ← score_C2
    score_UL2 ← score_L2
    stream_write(y_C2) → y_O
    stream_write(score_C2) → score_O
  end for
end loop
```

Clustering PE doesn't decrease loop rate, nor increases critical path length (max unit delay), at least for datapath. However it may increase control signals fanout as one state machine has to control more operators.

Reducing the number of FIFO stages decreases the systolic array latency, but increases cluster latency. A small systolic array latency is adapted to process sequences larger than the systolic array size: computation are done in several

Table 1. Impact of clustering PEs on Resources and Performances

Design	HLL(1)	HLL(2)	HLL(3)	HLL(4)
Cluster size (x)	1	2	4	8
8 PEs ALM util.	4336	3680	2246	1578
8 PEs M9K util.	8	4	4	4
Design Logic Util.	88 %	88 %	74 %	81 %
Max number of PE (N)	224	256	288	384
BCUPS	20.5	23.7	30.5	38.4

passes and results of the last PE are needed as soon as it is possible for the next computation. On the other hand, a small PE cluster latency is preferable when the query size fit the systolic array size: the first character of a bank sequence can be pushed as soon as the cluster has processed the last character of the previous bank sequence.

Depending of applications, a tradeoff between the size of the query sequence, the total number of PEs and the size of a PE cluster need to be found.

4 Performances

4.1 Platform Description

The platform used for our experiments is the XD2000i Development platform from XtremeData, Inc. [4]. This hardware holds a dual processor socket motherboard where one of the initial processor chip has been replaced by a XD2000i module. This In-Socket Accelerator embeds 2 Altera EP3ES260 FPGA components.

4.2 Design Variations Comparison

Clustering PEs had two goals: (1) share control logic and (2) decrease the number of inter-PEs FIFOs. Table 1 shows FPGA resource utilization (Adaptive Logic Module and M9K memory blocks) equivalent to 8 PE stages for different sizes of clusters. As the number of PE per ImpulseC process increases, resource utilization of equivalent partition of the design is reduced by 2.5, allowing the systolic array to be larger. For each size of clusters, we tried to fit as many PEs as possible on the FPGA, provided the design could sustain a frequency of 200MHz. Measured BCUPS (Billion Cells Updates Per Seconds) are also reported in table 1. This unit represents the number of matrix cells (cf. section 2.2) which are computed in one second, and is the standard unit to measure genomic sequence comparison algorithm performance.

It could be interesting to investigate if better performance could be reached by increasing the cluster size over 8, but, at this point, manually writing the code becomes too complex. This situation reveals the current tool limits.

Table 2. HLL versus HDL design performance

Design	HDL	HLL(1)	HLL(4)
Systolic Array Size	300	224	384
BCUPS	9	20.5	38.4

4.3 Comparison with Hand-Written HDL Code

In [7] the authors present a Smith & Waterman HDL-implementation on the same platform. The Smith & Waterman algorithm is one of the most popular genomic sequence comparison and is usually used as reference. In this hand-written code implementation, performance reaches 9 BCUPS. Table 2 summarizes performance of this implementation (HDL), of the first ImpulseC optimized release (HLL(1)) and of the highly-optimized ImpulseC release (HLL(4)).

Our final design, HLL(4), is 4.2x more powerful than HDL-implemented design.

4.4 State of the Art

While lots of paper show great speed-ups of genomic sequence comparison over sequential processor, it still remains very difficult to provide fair comparisons. Hardware platforms and algorithms are constantly evolving leading to huge design differences. Thus, we restrict our comparison to the well known Needleman & Wunsch and Smith & Waterman algorithms managing protein sequences and affine gap penalty. As in our implementation, these algorithms compute every cells of the matrix and don't make any optimization such as the diagonal-band-limited optimization which limits computation near the main diagonal of the matrix.

In [6] and [9], both design targets Virtex II XC2V6000 FPGA and respectively get 3.1 GCUPS and 10.54 GCUPS. XC2V6000 are now outdated FPGAs that contain only 33K slices. It would be interesting to see how the proposed architectures scale with current devices.

Two more recent designs, [7], on XD2000i platform, and [8], on XD1000 platform, exhibit respectively 15 GCUPS theoretical performance (9 GCUPS actually measured) and 25.6 GCUPS pick performance. These implementations seems currently to be the best references.

On conventional processors, Farrar [10] holds actually the best performance (up to 3 GCUPS) with the use of SIMD processor instruction set. A multi-threaded implementation on a 4 core Intel Xeon processor (as the processor available on the XD2000i platform) would achieved 12 GCUPS. Our hardware accelerator could also benefit from multi-threaded opportunity. Currently, only half of the FPGA resources of the XD2000i platform is used. As our architecture is not bandwidth limited, using both FPGAs shouldn't alter the design efficiency, providing up to 76.8 GCUPS, or a x6 speed-up over optimized multi-threaded SIMD software.

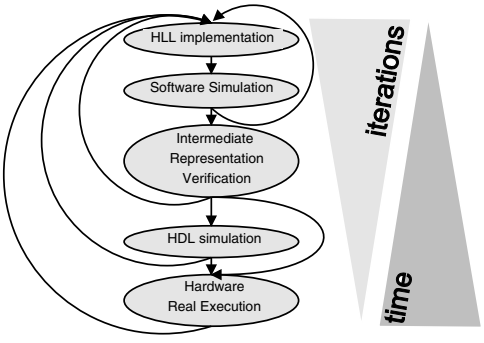

Fig. 3. ImpulseC design flow

5 Discussion

5.1 Fast and Reliable Design Process

Designing hardware accelerator with High Level Language significantly speeds up design process. In this experiment, a first systolic array has been created from scratch within a few hours, while it could have taken days in HDL. First of all, using HLL allows designers to have a behavioral description at a higher level compared to HDL, avoiding lots of signal synchronization problems that occur when dealing with low level signals (data_ready, data_eos and data_read for example). Instead, the use of ImpulseC stream read and write functions allows the designer to specify identical behavior in a very safe way. Inside a process, operation scheduling are deduced from ImpulseC code and from data dependencies. No other timing constraint need to be specified. With ImpulseC, changing operation order, for example, is thus quickly done, while it can be error-prone, or at least time consuming, in HDL.

Working with HLL makes implementation step faster, as shown in figure 3. Software simulation is even faster and is sometimes a sufficient test to pass to the next design process iteration, skipping HDL simulation which can be a more difficult way of debugging. In the same way, an intermediate model is generated during HDL generation, that can be explored with the *Stage Master Explorer* tool and from which designer can have a view of process performance (scheduling, pipeline rate,...).

Finally, this fast design methodology allows designers to try several design options, or add extra features that can be simply crippling with HDL languages. For example, in the design considered in this paper, adding features such as variable sequence length management or dynamic load of substitution matrix is much easier than it could be with HDL.

5.2 Reusable and Portable Design

HLL makes design specification more understandable, easier to maintain and also easier to reuse. Using corresponding Platform Support Package to port an

ImpulseC design from a platform to another one means that application designers would not have to manage architecture dependent issues. This role is devoted to API level and to the platform designers.

Unfortunately, the PSP we experimented is still in its early development stage and was not able to provide all standard features. We had to rewrite part of our software and slightly adapt hardware processes, when moving from Nios2 to the host Xeon processor. We can expect that the future PSP releases will make this transition transparent.

However, moving an accelerated application from an FPGA architecture to a very different one (with different RAM block size or a higher memory bandwidth for example), could require design modifications in order to get maximum performance. Unchanged design should at least provide functional designs and technology modification would be easier to do within HLS tool environment.

5.3 Obtaining Real Performance

One typical claim against HLS tools is that the reduction of design time comes at the price of a loss of performance, as these tools prevent designers from optimizing the details of the architecture. We believe such claims do not hold when it comes to the parallelization of high performance computing applications on FPGAs. As a matter of fact, we even believe that using HLL can actually help designer improving performance as it allows designers to abstract away all non critical implementation details (ex: optimizing the RTL datapath of a Processing Elemenent [7]) and instead focus on performance critical system level issues (ex: storage/communication balance).

Indeed, thanks to the expressivity of HLL, designers effectively explore a much larger design space, leading them to optimal architectural solutions they would have never considered in the first place. Even though such system level optimizations could also be implemented at the RTL level using HDLs, it turns out they rarely are so. The explanation is quite straightforward : the impact of these optimizations on performance is often discovered at posteriori (or too late in the design flow) and reenginnering the whole design at the HDL level would require too much effort.

Last, thanks to the drastic reduction in design time to get an efficient implementation, designers can afford to implement several non critical design optimizations. Even if the impact of one of such optimization alone is generally limited (often 10%-30% performance gain), their combined effect on global performance is generally multiplicative/cumulative and ultimately boosts global performance, for very little design effort.

5.4 Toward User Assisted Source to Source Transformation Tools

It is clear that HLS do not provide a direct algorithmic to gate design path. In a similar way high performance architecture programmer do, designers need to modify and rewrite their source code so as to get the tool to derive the hardware architecture they actually want for their application. Even though such a rewrite

process is much easier to carry with a HLL, it still leaves a lot of burden to the designer.

As a consequence, we feel that there is a need for HLS oriented source to source refactoring tools, which would help designers re-structuring their programs by providing them with a code transformation toolbox tailored to their needs. Such a toolbox would of course encompass most of the transformations that can be found in semi automatic parallelizing source to source compilers (loop and array layout transformations for example), but should also provide domain-specific code transformations specifically targeted at HLS tools users (process clustering could be one of them).

Acknowledgment

We would like to thank Impulse Accelerated Technologies for their valuable help. This work has been supported by the French ANR BioWIC (ANR-08-SEGI-005).

References

1. Holland, B., Vacas, M., Aggarwal, V., DeVille, R., Troxel, I., George, A.D.: Survey of C-based application mapping tools for reconfigurable computing. In: Proceedings of the 8th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD 2005), Washington, DC, USA (September 2005)
2. Lavenier, D., Giraud, M.: Bioinformatics Applications, in Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays. In: Gokhale, M.B., Graham, P.S. (eds.), ch. 9. Springer, Heidelberg (2005)
3. Pellerin, D., Thibault, S.: Practical FPGA Programming in C. Pearson Education, Inc., Upper Saddle River, NJ (2005)
4. `http://old.xtremedatainc.com/`
5. Aung, L.A., Maskell, D., Oliver, T., Schmidt, B.: C-Based Design Methodology for FPGA Implementation of ClustalW MSA. In: Rajapakse, J.C., Schmidt, B., Volkert, L.G. (eds.) PRIB 2007. LNCS (LNBI), vol. 4774, pp. 11–18. Springer, Heidelberg (2007)
6. Oliver, T., Schmidt, B., Nathan, D., Clemens, R., Maskell, D.: Multiple Sequence Alignment on an FPGA. In: Proceedings of 11th International Conference on Parallel and Distributed Systems, July 22-22, vol. 2, pp. 326–330 (2005)
7. Allred, J., Coyne, J., Lynch, W., Natoli, V., Grecco, J., Morrissette, J.: Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer. In: International Parallel and Distributed Processing Symposium, pp. 1–4. IEEE Computer Society, Los Alamitos (2009)
8. Zhang, P., Tan, G., Gao, G.R.: Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. In: Proceedings of the 1st International Workshop on High-Performance Reconfigurable Computing Technology and Applications: Held in Conjunction with Sc 2007, HPRCTA 2007, Reno, Nevada, November 11-11, pp. 39–48. ACM, New York (2007)
9. Yilmaz, C., Gok, M.: An Optimized System for Multiple Sequence Alignment. In: International Conference on Reconfigurable Computing and FPGAs, pp. 178–182. IEEE Computer Society, Los Alamitos (2009)
10. Farrar, M.: Striped Smith-Waterman speeds database searches six times over other SIMD implementations. Bioinformatics 23, 15661 (2007); A Smith-Waterman Systolic Cell