Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 1967–1976

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Massively parallel FPGA-based implementation of BLASTp with the two-hit method

Lars Wienbrandt[a,1], Stefan Baumgart[b], Jost Bissel[b], Florian Schatz[a], Manfred Schimmler[a]

[a]*Department of Computer Science, Christian-Albrechts-University of Kiel, Hermann-Rodewald-Str. 3, 24118 Kiel, Germany*
[b]*SciEngines GmbH, Fraunhoferstr. 13, 24118 Kiel, Germany*

Abstract

Protein database search requests are generally being performed using the BLASTp algorithm, introduced by NCBI [1]. Since it is computationally intensive, it becomes more and more ineffective with today's growth of sequence database sizes. The needs for an efficient parallel implementation arise. In this paper, we focus on a massive parallelization using the FPGA-based hardware architecture RIVYERA [2]. The aim is to reach speedups in orders of magnitude with a flexible implementation while saving energy costs compared to PC-based database search.

We keep our implementation close to the structure published by Kasap et al. [3, 4] and include ideas from Sotiriades et al. [5] such that all parts of the algorithm are organized in components of a long pipeline. We also use the idea of the two-hit method [6] to keep the computational effort small. Besides the related work, we perform the very last step of the algorithm to produce a gapped alignment with the Needleman-Wunsch algorithm in software, only with the option of hardware processing after reconfiguration. This saves FPGA-resources and allows an even higher degree of parallelism.

Keywords: BLASTp, FPGA, sequence alignment, protein database search, Swiss-Prot

1. Introduction

Biological sequence alignment is a very common task to find similarities of a query sequence to one or several sequences in a large database. The aim of this operation is e.g. to find out information of a newly sequenced protein by comparison to already known protein sequences from the database. With this information a biologist may suggest the functionality or effect of this protein. BLAST [7] was originally developed to efficiently perform this task, both for nucleotide sequences and for proteins. Since it is heuristic, it is much faster than dynamic programming algorithms like the Smith-Waterman or Needleman-Wunsch algorithm [8, 9], although not always providing optimal results. However, with the exponential growth of database sizes as well as the increasing length of the query sequences, the original BLAST becomes inefficient. With continuing the development and maintenance of BLAST by NCBI [1], several enhancements, such as the two-hit method and gapped BLAST [6], were applied to the algorithm to reduce computation time and improve result quality. Anyway, the execution of the BLAST algorithm still suffers from long

Email address: lwi@informatik.uni-kiel.de (Lars Wienbrandt)
[1]Corresponding author

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.215

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	-1	-1	-2	-2	0	-3	-1	4

Figure 1: BLOSUM62 scoring matrix for amino acid comparison.

runtimes, especially for more than one query or parameter set, and utilization of fast computing architectures becomes inevitable.

FPGAs have already been proven to be suitable for tasks in bioinformatics, due to their feature of flexible programming and ASIC-like performance [10, 5, 3, 4]. Massive parallelization of such hardware implemented algorithms provide speedups of several orders of magnitude [11, 12, 13].

We are now addressing this problem by discussing the implementation of BLASTp on a massively parallel FPGA-based hardware architecture, namely RIVYERA [2]. Our implemented pipeline is closely oriented to Kasap et al. [3] who first implemented gapped BLASTp with the two-hit method on a Virtex-4 FPGA by Xilinx using the Handel-C language. Later, Kasap et al. presented a variation of their implementation to support PSI-BLAST, an iterative version of BLASTp [4, 6]. Sotiriades et al. [5] also published a hardware implementation of BLAST, freely configurable to use with BLASTn, BLASTp, BLASTx, TBLASTn and TBLASTx, but without using gapped BLAST or the two-hit method.

Other BLASTp hardware implementations include Mercury BLASTp [14], implementing especially the ungapped and gapped extension steps on a Xilinx Virtex-II device and reaching significant speedups while providing 99% accuracy compared to the NCBI software version. Mahram and Herbordt [15] speed up the process while outsourcing the ungapped and gapped extension on an Altera Stratix-III FPGA connected to 4.5*GB* DRAM. Both implementations benefit from devices with immense logic and memory resources, providing a high potential to speed up the BLASTp algorithm.

However, our approach is to use a Xilinx Spartan-3 5000 device, a low-cost FPGA with the drawback of a smaller performance potential, but therefore low energy consumption, compared to the Xilinx Virtex devices. We are still able to get three of our presented BLASTp pipelines on each FPGA and compare this to the runtime of a standard PC. A massive parallelization step is done in using our design on the hardware architecture RIVYERA (s. section 3). The energy consumption is being compared to a standard PC cluster.

2. BLAST

BLAST is short for Basic Local Alignment Search Tool. It is an heuristic algorithm which can be used to find alignments of both, protein (*BLASTp*) or nucleotide sequences (*BLASTn*). There are also other variations like *BLASTx* or *PSI-BLAST*. It is generally used for protein or DNA sequence database searches to find sequence similarities. Since it is heuristic it benefits from fast execution but does not guarantee to find the optimal alignment, unlike the Smith-Waterman algorithm [8]. Because of many already existing publications with well described explanations of the BLAST algorithm, we only show a short summary in the following.

Several steps have to be performed during a BLAST run. First, the query sequence is preprocessed to find k-words. k-words are k-mers (short sequences of size k) which are part of the query sequence or at least similar to parts.

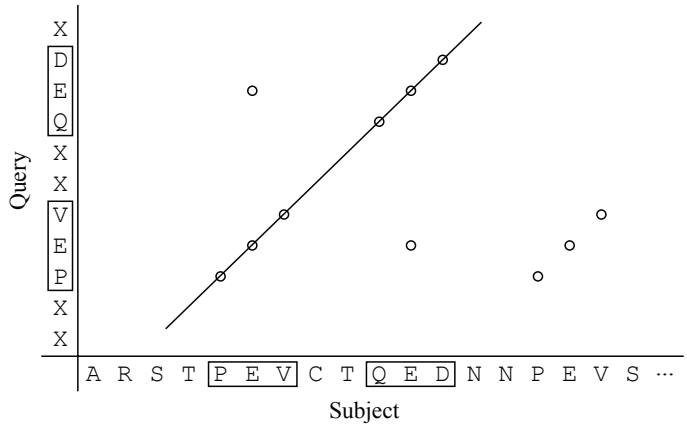

Figure 2: Example for a two-hit in a dot-plot diagram.

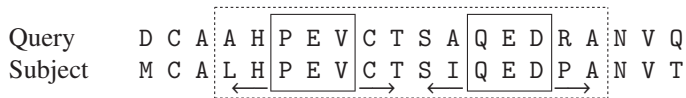

Figure 3: Example for the extension of a two-hit.

The value k is fixed, but different for either BLASTn ($k = 11$) or BLASTp ($k = 3$). The similarity of two k-mers is calculated by direct comparison. Scores of the faced residue pairs are looked up in a scoring matrix, e.g. BLOSUM62 for proteins (s. fig. 1), and summed up resulting in the similarity score. According to a predefined threshold value a k-mer becomes a k-word or not.

Second, exact matches (*hits*) of the k-words in the database sequences are located. Third, each hit is finally being further examined by extending it in both directions. A new similarity score is calculated with every new pair of residues in the extension. If the score declines a certain cut-off distance below the so far reached maximum, the extension is rolled back to its previous state where it reached the maximum score. If this score exceeds another threshold value, the hit is being reported and the result of this extension step states a local alignment of the related database sequence and the query sequence.

2.1. Two-hit Method and Gapped Alignment

Obviously, the quality of the alignment of the method described above is dependend on the given parameters which highly affect the runtime of the algorithm. A high cut-off distance results in long runtimes for the extension of each hit, and a high threshold implies less results but higher quality. To get such high-quality alignments while reducing the runtime of the algorithm Altschul et al. have presented the two-hit method [6].

Each pair of hits of the first step in the algorithm is being compared to be in the same distance in the query sequence as well as in the subject sequence, i.e. they are on the same diagonal in a dot-plot diagram (s. ex. in fig. 2). To save runtime and memory the distance is bounded to a certain parameter A. Therefore, the following equation shows the condition for a two-hit whereby s_0 and s_1 state the location of two hits in the subject and q_0 and q_1 their locations in the query respectively:

$$3 \leq q_1 - q_0 = s_1 - s_0 < A$$

Only such *two-hits* are being further processed by the next step, the ungapped extension. It differs from the previous extension step by extending two hits (the *two-hit*) at once now. The scores are being combined by accumulation, but for the rest the same rules apply. An example is shown in figure 3. The solid rectangles mark the two-hit, the dashed one the extension. Arrows indicate the direction of the extension.

If the score of this new extension step exceeds a certain threshold value, a gapped alignment is being processed as an additional last step introducing gaps. The alignment is created by a slightly modified version of the well-known

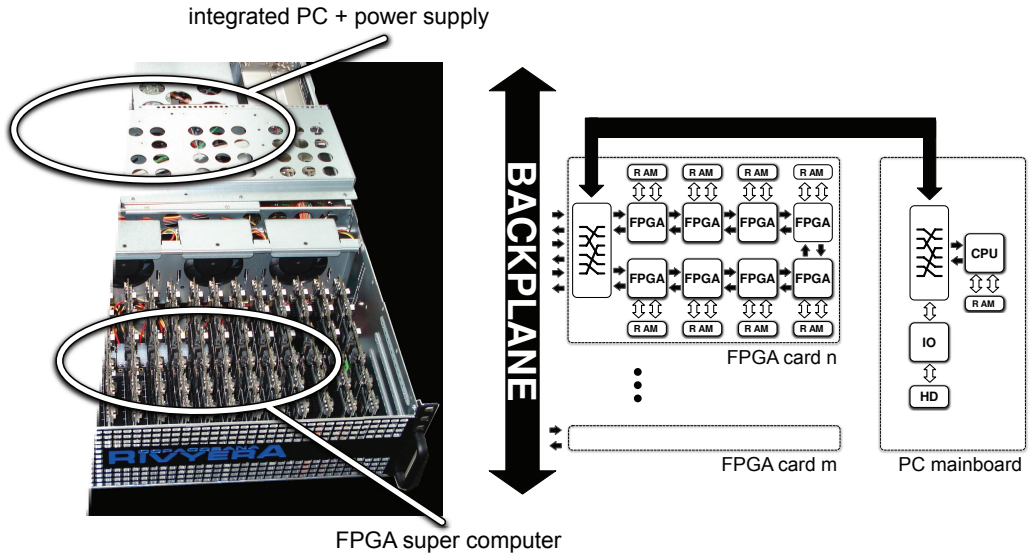

Figure 4: The RIVYERA S3-5000

Needleman-Wunsch algorithm [9] with affine gap penalties. The modification includes clipping to the boundaries of the extension of the two-hit, and the traceback to determine the alignment is being started at the cell with the calculated maximum score. Additionally, runtime is being reduced by omitting the calculation of cell values where the score declines below the certain cut-off distance (already described above) from the so far calculated maximum cell value.

The obtained gapped alignment now presents a result of the database query. Both enhancements - the two-hit method and gapped alignment - are included in current versions of BLAST, formally known as BLAST2.

3. RIVYERA

Introduced in 2008, RIVYERA [16] is the direct successor of the original COPACOBANA [17, 18], the Cost Optimized PArallel COde Breaker and ANAlyzer presented in 2006.

A brief overview of the RIVYERA is provided in the following. The reconfigurable massively parallel FPGA-based hardware architecture RIVYERA is designed to be a fully scalable system consisting of two basic elements. The first is the massively parallel FPGA-based super computer. The second is a rack mountable 3HE chassis including two redundant 650 watt power supplies and a standard eATX based server grade mainboard. The latter is equipped with an Intel Core i7-930 processor, 12*GB* of RAM and 2*TB* of hard disk space.

The FPGA super computer consists of the RIVYERA backplane and the RIVYERA FPGA cards. The backplane provides 16 slots mechanically compatible to PCI-Express. It offers three independent bus systems, a shared bus, a systolic bus system and JTAG. Each bus system has the ability to address various applications. In general the provided JTAG is intended to be used for debug and in-system analysis. It is compliant to the Xilinx JTAG interface. The classical shared bus system offers backward compatibility to the original COPACOBANA. It can be used for data broadcast to all FPGAs for example. The systolic bus system can be described as a point-to-point bus system connecting every two neighboring FPGAs directly. It has been designed to provide high-throughput communication with minimal latency. In a traditional point-to-point chain a system of 128 FPGAs or more might suffer from high latencies. Within RIVYERA a configurable routing scheme has been realized to provide minimum latency and hops for each data package being transferred. Therefore data sinks and sources can be defined without re-memorizing the low level architecture.

The backplane slots provide 16 communication lanes per direction. Each lane consists of a differential coupled pair of wires utilizing an LVDs standard and impedance driven design. The direction of each lane can be configured within the API. The typical setup is a symmetric point-to-point connection, 16 lanes for receiving data and 16 for transmitting.

The computational performance is provided via the RIVYERA FPGA cards. In general, various FPGA types and vendors can be mixed to optimize the hardware for dedicated applications. In case of the RIVYERA S3-5000, which is the underlying architecture for this application, each FPGA card offers eight Xilinx Spartan-3 5000 FPGAs for user code. Each user FPGA provides 64*MB* of directly attached memory in two dedicated 32*MB* DRAM chips. Furthermore, an up to 32*GB* large SD-HC based flash memory card can be attached directly to each FPGA. Up to 16 FPGA cards fit on the RIVYERA backplane for a fully equipped machine.

PCI-Express based interface cards allow multiple full bandwidth connections between the integrated PC and the FPGA computer. Typical setups alter from one up to seven controller interface cards, either configured as PCIe 1x or PCIe 16x, depending on the standard main board used inside the machine and the bandwidth needed. For this application one PCIe 16x controller is used. Nevertheless, the machine offers an integrated self configuring uplink offering the capability to connect several machines to a larger one.

For application development an integrated API is provided for the software interface as well as for the hardware interface to handle the data transfer between host and user FPGAs. An overview of structure of the RIVYERA S3-5000 is depicted in figure 4.

4. Application Structure and Implementation

For the implementation of the BLAST algorithm we concentrate on BLASTp for searching amino acid sequences in protein databases. The application consists of two parts - the hardware part to accelerate the computation intensive steps of the application and the software part for the pre- and postprocessing steps as well as controlling the hardware. Therefore, the hardware part performs those steps of the algorithm which have to be processed very frequently during the execution and, hence, require most of the runtime. These include the basic hit finding, the two-hit method and the ungapped extension of the two-hits. Since the gapped extension with the Needleman-Wunsch algorithm would take a lot of hardware resources and is needed for an expected small number of hits which pass the ungapped extension step only (s. performance results in table 2), this part of the algorithm is processed by the host software per default. Since we are aiming to run the BLASTp algorithm on the RIVYERA machine in a high degree of parallelism, this process may become the bottleneck of the whole system. Hence, the RIVYERA could be (partly) reconfigured to speedup the processing of the necessary Needleman-Wunsch alignments. Additionally, the software is responsible for the preprocessing of the query to find the k-words, the initialization of the hardware pipelines (submission of k-words and queries), database streaming and, of course, the postprocessing of reported hits to present human-readable alignments. The detailed description of both parts can be seen in the following.

4.1. BLASTp hardware pipeline

A BLASTp hardware pipeline consists of several components, each performing designated tasks to support the implementation of each step of the algorithm. The pipeline starts with the *HitFinder* component which searches for the k-words in the submitted subject sequence. These hits are routed through a *HitMux* component to be stored in one of four *HitFIFOs*, each containing a different part of all hits (see detailed description below). Each HitFIFO is connected to a *TwoHitMethod* component which analyzes each pair of hits to be a two-hit. Afterwards, each two-hit is recorded in a *TwoHitFIFO*. The two-hits of each of the four TwoHitFIFOs are routed through a *FIFOSwitch* to be extended in the next step. The *UngappedExtender* is subsequently processing the extension of these two-hits from the TwoHitFIFOs. This component is in direct interaction with two memories - the *QueryBuffer* and the *SubjectBuffer* - each containing the current query and subject sequence respectively.

The interface to the RIVYERA API is implemented in a *ControlUnit* which is responsible for the data transfer between host software and BLASTp pipeline, i.e. routing initialization data to the respective components, processing the input subject sequence and reporting hits passing the ungapped extension step. In the following all main components are described in detail. An overall structure is shown in figure 5.

HitFinder: The HitFinder searches for occurences of k-words in the subject sequence. Each k-mer of the subject is simply looked up in a hashtable, which is precomputed and initialized by the host software (s. section 4.2). If there exist occurences in the query sequence, the return values are valid positions and will be recorded through the HitMux component in one of four HitFIFOs, otherwise the k-mer is ignored for further processing.

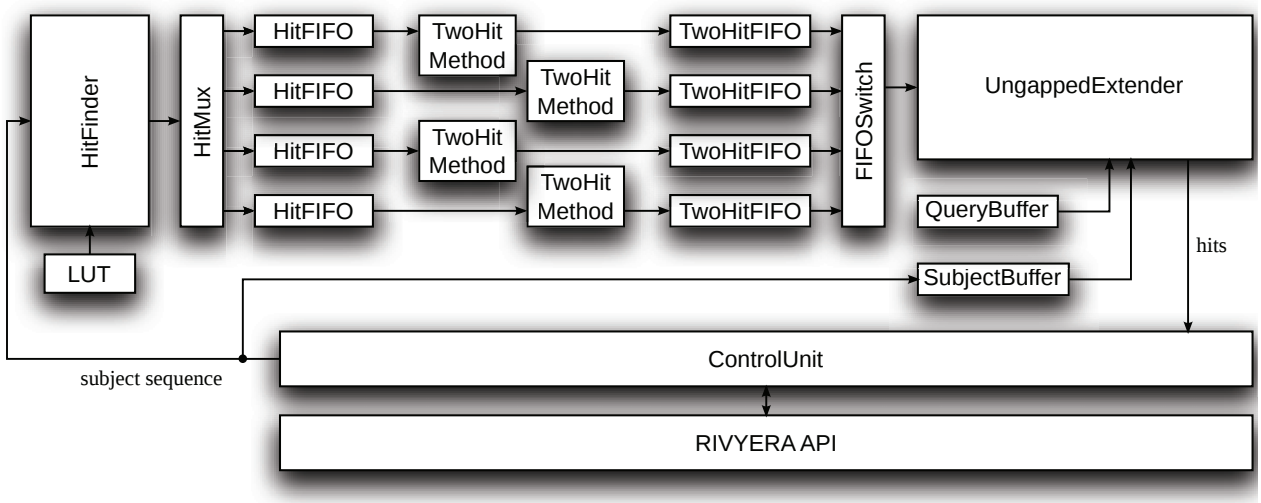

Figure 5: Structure of the BLASTp hardware pipeline.

HitMux: The HitMux component decides which FIFOs have to store a hit. This depends on its position in the subject sequence. Every 64 amino acids this component routes the hits to the next FIFO. This makes it easier to run four independent TwoHitMethod components (one for each FIFO) concurrently. To ensure a correct behaviour of the comparisons of two hits, each FIFO also records the 64 next hits for the following FIFO. The comparators are then able to process all comparisons of all hits until the maximum distance for a two-hit correctly.

HitFIFO: Hits from the HitFinder are stored in a HitFIFO. The HitFIFO allows access and deletion of the first element, like in a usual FIFO. Additionally, another port allows simultaneous read-only access of any other element in the queue. This is necessary for the TwoHitMethod component described next.

TwoHitMethod: Each pair of entries in a HitFIFO is being compared by the TwoHitMethod component. Since the entries of a HitFIFO are sorted by their location in the subject sequence, the first occurence of a k-word is the first element in the queue. Hence, this element is being compared to every other element in the queue in ascending order, testing the condition described in section 2.1, until the distance of two hits in the subject sequence exceeds the algorithm parameter A for the maximum two-hit distance. Then, the first element is being removed. Any two hits are reported as a two-hit if the distance in the query sequence match the distance in the subject sequence exactly. Since this is generally the most computational part of the algorithm, four comparisons are performed concurrently in each clock cycle in each TwoHitMethod component.

TwoHitFIFO: Besides the HitFIFO component this TwoHitFIFO acts as a usual FIFO storing all hits from the two-hit method.

FIFOSwitch: Whenever a TwoHitFIFO component holds an element and the extension block is ready to process the next hit, the element is routed to the UngappedExtender. All four connected TwoHitFIFOs are tested subsequently.

UngappedExtender: To extend a two-hit two steps are implemented in the UngappedExtender. The first step is the extension directed inwards to fill the gap between the two hits. The score calculated so far states the local maximum to start the extension outwards in the second step. Now, with every new pair of residues on both ends, the new calculated score is supervised not to drop the predefined cut-off distance below the so far calculated maximum score. If this happens, the extension is rolled back to the local maximum and, if the score exceeds a certain threshold, this hit is being reported to the ControlUnit and the next two-hit from a TwoHitFIFO is being processed.

For the comparison of the two pairs of residues during the extension process, the UngappedExtender needs direct dual-ported access to the scoring matrix, e.g. BLOSUM62, for this run of the algorithm. This is easily implemented in a dual-ported ROM.

QueryBuffer and SubjectBuffer: For the extension process the UngappedExtender also needs immediate access to two residues of the query and subject sequence each in one clock cycle. Therefore, the QueryBuffer and SubjectBuffer, which contain the query and the so far processed subject sequence respectively, are each implemented in dual-ported block RAM and directly connected to the UngappedExtender.

ControlUnit: The ControlUnit handles the trivial but most complex task of routing incoming data to the respective component while distinguishing between initialization and normal operation. The incoming subject sequence is provided to the four ports of the HitFinder component. It is distributed in a way that each k-mer for one port has a distance at least exceeding the maximum distance parameter for the two-hit method. Hence, no comparison between any two elements from different HitFIFOs is needed by the TwoHitMethod component.

Additionally, hits reported from the ungapped extension step are sent back to the host immediately.

4.2. Software Control

The software needed to support the hardware pipeline runs on the internal PC of the RIVYERA. It mainly consists of three parts, which are pre- and postprocessing as well as controlling the hardware pipeline. So far, there are only two computationally intensive tasks, one in the pre- and one in the postprocessing step each. The former is to generate all k-words according to a given query sequence. These k-words have to match a certain threshold value T. To determine all k-words at a specific location, a backtracking technique is used. This way, runtime can be reduced, compared to the naive and exhaustive method. The backtracking is described in the following.

We start taking a k-mer from the query and storing its location. Since, in general, the perfect match generates the highest score in commonly used scoring matrices, e.g. BLOSUM62, this k-mer can be taken as a k-word if its score exceeds T. Otherwise, all substitutions of amino acids in this k-mer would also fail the threshold T. If this k-mer has been inserted in the k-words list, we start substituting the first amino acid with all other possibilities. For all recorded entries we try a second substitution etc. Afterwards, we continue the same way substituting the second amino acid from the original k-mer and finally, the third.

Let S be the scoring matrix, a_i, b_i the amino acids from the original and the altered k-mer at index i respectively, and $S(a_i, b_i)$ the matrix entry for these two amino acids. Then, the score s of the altered k-mer against the original k-mer can easily be calculated and the altered k-mer will be inserted in the k-words list if the following condition holds:

$$s \geq T \qquad \text{whereby} \qquad s = \sum_{i=0}^{k-1} S(a_i, b_i).$$

Hence, the average runtime is significantly reduced for common threshold values T, but as a matter of course the overall complexity remains $O(20^k)$ of this problem.

The second computationally intensive task is the generation of gapped alignments from extended two-hits using the Needleman-Wunsch [9] algorithm. Fortunately, only a small percentage of the extended two-hits reach the predefined threshold value T_{ext} and therefore become reported hits. For exact values, the performance results in table 2 of section 5 show the number of processed alignments according to the corresponding query, its length and the number of reported hits from the first step of the algorithm.

However, the Needleman-Wunsch algorithm is computationally extensive, even if it is performed only on a small part of hits. Hence, to save runtime, the algorithm is optimized. If the score of a cell declines a certain cut-off distance below the so far calculated maximum (as for the extension of a two-hit), the neighboring cells, which would further decrease the score, are not being calculated. Hence, a CPU has got enough resources to deal with such a small number of alignments to be processed without becoming a performance-bottleneck. However, focussing the parallelization on the RIVYERA, processing multiple queries at once may still overload the CPU's resources. In such cases, our idea for prospective improvement is outsourcing the Needleman-Wunsch algorithm to hardware as well. The RIVYERA can be reconfigured to perform this task afterwards.

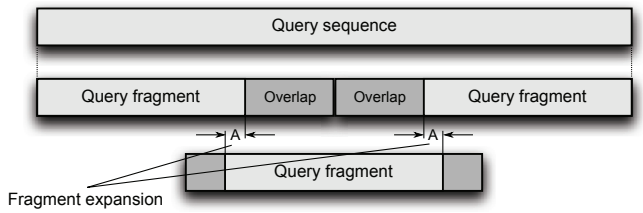

Figure 6: Distribution of a query sequence into overlapping fragments.

	Slices	%	Slice FFs	%	LUTs	%	BRAMs	%
one pipeline	4888	15%	5562	9%	6865	11%	37	36%
three pipelines	10877	33%	12440	19%	17101	26%	99	95%

Table 1: Device utilization of one and three BLASTp hardware pipelines on a Spartan-3 5000 including the RIVYERA API.

Large query sequences exceeding the capability of one BLASTp pipeline (currently 1024 amino acids) are handled by simply dividing them into smaller fragments overlapping the predecessing and the successing neighbors by at least the maximum distance A for a two-hit. The corresponding k-words and hashtable are calculated only for each particular fragment instead of the whole query.

In order to process the extension of any potential two-hit, the BLASTp pipeline will be initialized with the precomputed hashtable exclusively fitted for its fragment (s. above), but with a part of the query as large as possible to fill the QueryBuffer for the extension step. This includes at least an overlap of half of each neighboring query fragment. An example of a query fragmentation is shown in figure 6.

5. Performance Evaluation

The BLASTp pipeline described in section 4.1 has been implemented on a Spartan-3 5000 FPGA for the RIVYERA using Xilinx ISE 12.3. We chose the maximum query size to be 1024 amino acids, longer queries have to be split. Then, according to the synthesis results, one pipeline utilizes one third of the FPGA, whereby block RAM utilization is the limitative factor. Hence, we decided to implement three independent pipelines per RIVYERA FPGA resulting in a fully utilized machine. The device utilization of a Spartan-3 5000 implementing one and three BLASTp pipelines respectively including the RIVYERA API can be seen in table 1.

The components of the hardware implementation are clocked at $100MHz$. The intention was to process incoming data almost immediately while it is being streamed without stalling the communication flow for most cases. However, depending on the length of the query and consequentially the number of hits found in the first step, the component analyzing potential two-hits unfortunately forms the bottleneck already for queries around 300 amino acids. Hence, the TwoHitMethod component is already in our focus for further optimizations.

The total runtime of our hardware implementation results from two parts, the hardware and the software part, but both are overlapping each other. As described in section 4.2, the host has to precompute a hashtable of k-words for each query and to initialize the BLASTp pipelines with the query sequences and those hashtables in advance. This step contributes no significant delay to the runtime of the main computational part. The computationally more intensive part of calculating the Needleman-Wunsch alignment for each succesfully extended two-hit would have a greater influence. Fortunately, this step is being processed in parallel to further hit finding, such that only alignments from results at the very end of the database contribute to the total runtime, again resulting in no significant delay.

Table 2 shows the performance results for one BLASTp pipeline on one RIVYERA FPGA compared to BLASTp running on a standard PC (Intel Core2 Quad, $2.4GHz$, $8GB$ RAM) using one core. The test database (Swiss-Prot protein database, release 2010_12 [19]) contains 184,678,199 amino acids in total. The software used is NCBI BLAST v2.2.24 [1] with standard parameters. The same parameters were applied to the hardware. The hardware runtime includes the Needleman-Wunsch postprocessing step of the BLASTp algorithm.

Obviously, our hardware implementation of one BLASTp pipeline is about as fast as the PC. Compared to the performance results achieved by Kasap et al. [3] our total runtimes are slightly worse. In our opinion the reason

Query	Query length	Database hits	Extensions	N.-W. alignments	Final reports	Software runtime	HW pipeline runtime
P05013	189	56075559	2039728	6399	87	3.636s	3.893s
P14942	222	74909704	2741691	6601	168	5.748s	4.201s
P00762	246	93191597	3908499	7936	718	6.587s	4.585s
P07327	375	137952712	5797792	17469	618	9.254s	5.878s
P10635	497	171434422	7282462	21182	907	11.793s	7.173s
P03435	567	210346114	9060054	28299	318	13.371s	9.030s
P89680	1094	364120358	14713500	41203	146	17.742s	20.072s
P03589	1126	388210234	16331343	44850	152	19.775s	22.338s

Table 2: Performance of one hardware implemented BLASTp pipeline compared to a standard PC for several database sequences used as queries.

is firstly, their usage of a more powerful FPGA, a Xilinx Virtex-4 LX160, providing at least two times more logic resources including LUTs with 16 times higher capacity, and secondly, an older and consequentially smaller release of the Swiss-Prot database.

We have not used the `-a` option of NCBI BLAST to get a fair comparison of one CPU core against one hardware pipeline. If we had fully utilized all four cores of our test system we would have needed to split our queries the same way the BLAST software does to run them on four hardware pipelines concurrently as well. Our estimation is that this would lead to about the same relations of runtime results as presented in table 2.

5.1. Massive Parallelization

Now, a trivial approach to benefit from the usage of the RIVYERA is its ability to process several queries at once. In total three queries can be processed by one FPGA resulting in the processing of 384 queries concurrently. According to Amdahl's law, the postprocessing step including the Needleman-Wunsch algorithm prevents the execution speedup vs. a standard PC to be the number of queries being processed in parallel. Instead, testing with fully utilizing the RIVYERA shows a runtime increase of a factor of about two against the optimal runtime with ideal parallelization, resulting in a speedup of about 200. The query size for each query in this test was about 1000 amino acids, since this fits the maximum query size of our implementation. Figure 7 shows a diagram of the achieved speedups according to the number of BLASTp pipelines used.

Instead of processing different queries in parallel, it is certainly possible to process the same queries with different parameter sets as well. Another benefit is the ability to process large queries very fast. Since large queries can easily be divided by the method described in section 4.2 and each processing pipeline is able to process one fragment, the result is the concurrent processing of all fragments. The expected runtime now corresponds to the runtime of the same number of queries as fragments with the size of the largest fragment.

Furthermore, regarding energy consumption, our application running on a fully utilized RIVYERA machine consumes about $600W$ while a standard PC consumes about $250W$. Considering a PC cluster of 200 standard PCs, assuming they reach the same speedup as the RIVYERA, the cluster would consume $200 * 250W = 50,000W$. This is more than 80 times the power consumption of the RIVYERA!

6. Conclusion and Further Work

The massive parallelization of BLASTp using FPGA-based hardware mainly benefits from processing a large amount of different queries or different parameter sets concurrently. The same advantages are achieved if large queries are being fragmented such that all fragments are being processed in parallel. This way speedups vs. a standard PC of about 200 can be reached with only 1.25% of the energy consumed.

However, our implementation has several bottlenecks which should be optimized in the future. Firstly, higher speedups can be reached if it is possible to achieve a finer grained parallelization while even more fragmenting the queries directly on the hardware pipeline by utilizing more HitFinder components in parallel. Secondly, the two-hit method could be improved if more components can be applied to the same problem size as well as increasing the components frequency. Thirdly, the main disadvantage is the postprocessing of the hits with the Needleman-Wunsch

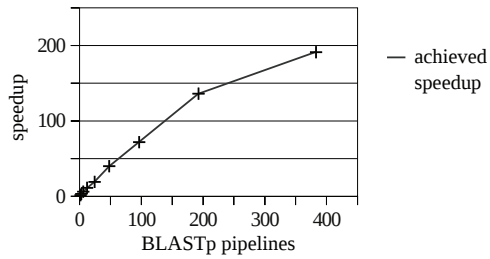

Figure 7: Achieved speedups according to the number of utilized BLASTp pipelines on the RIVYERA machine.

algorithm on the host PC. With a small grade of parallelization this does not take into count, but with a higher degree the necessity to postprocess hits concurrently appears. Our idea is to source this task out to a hardware implementation running in parallel on a part of the machine. This reduces the number of original BLASTp pipelines but may still lead to higher speedups. Otherwise, the complete RIVYERA machine can be reconfigured to process all Needleman-Wunsch alignments afterwards. This would lead to the same beneficial effect.

Altogether, BLAST remains one of the most important available tools to perform biological database searches. If only a few queries of small size are to be processed, using a standard PC or the online interfaces of most database websites is probably sufficient. But for huge amounts of longer queries against larger databases, using a massively parallelized implementation on a hardware architecture will help to save valuable time.

This work has been co-funded by the Innovationsstiftung Schleswig-Holstein, Pr.-No. 2008-45.

References

[1] NCBI BLAST Basic Local Alignment Search Tool, `http://blast.ncbi.nlm.nih.gov/Blast.cgi`.
[2] SciEngines GmbH, `http://www.sciengines.com`.
[3] S. Kasap, K. Benkrid, Y. Liu, Design and Implementation of an FPGA-based Core for Gapped BLAST Sequence Alignment with the Two-Hit Method, Engineering Letters 16.
[4] S. Kasap, K. Benkrid, Y. Liu, A High Performance FPGA-based Implementation of Position Specific Iterated BLAST, in: FPGA'09, 2009, pp. 249–252.
[5] E. Sotiriades, A. Dollas, A General Reconfigurable Architecture for the BLAST Algorithm, VLSI Signal Processing 48 (2007) 198–208.
[6] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, D. J. Lipman, Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, Nucleic Acids Research 25 (1997) 3389–3402.
[7] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, Basic Local Alignment Search Tool, Journal of Molecular Biology 215 (3) (1990) 403–410.
[8] T. F. Smith, M. S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 147 (1981) 195–197.
[9] S. B. Needleman, C. D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of Molecular Biology 48 (3) (1970) 443–453.
[10] T. F. Oliver, B. Schmidt, Y. Jakop, D. L. Maskell, High Speed Biological Sequence Analysis With Hidden Markov Models on Reconfigurable Platforms, IEEE Transactions on Information Technology in Biomedicine 13 (2009) 740–746.
[11] J. Schröder, L. Wienbrandt, G. Pfeiffer, M. Schimmler, Massively Parallelized DNA Motif Search on the Reconfigurable Hardware Platform COPACOBANA, in: PRIB2008, Lecture Notes in Bioinformatics, Vol. 5265, 2008, pp. 436–447.
[12] L. Wienbrandt, S. Baumgart, J. Bissel, C. M. Y. Yeo, M. Schimmler, Using the reconfigurable massively parallel architecture COPACOBANA 5000 for applications in bioinformatics, in: ICCS2010, Procedia Computer Science, Vol. 1, 2010, pp. 1027–1034.
[13] M. Schimmler, L. Wienbrandt, T. Güneysu, J. Bissel, COPACOBANA: A Massively Parallel FPGA-Based Computer Architecture, in: B. Schmidt (Ed.), Bioinformatics – High Performance Parallel Computer Architectures, CRC Press, 2010, pp. 223–262.
[14] A. Jacob, J. Lancaster, J. Buhler, B. Harris, R. D. Chamberlain, Mercury BLASTp: Accelerating Protein Sequence Alignment, ACM Transactions on Reconfigurable Technology and Systems 1.
[15] A. Mahram, M. C. Herbordt, Fast and Accurate BLASTP: Acceleration with Multiphase FPGA-Based Prefiltering, in: Proceedings of ICS'10, 2010, pp. 73–28.
[16] G. Pfeiffer, S. Baumgart, J. Schröder, M. Schimmler, A Massively Parallel Architecture for Bioinformatics, in: ICCS2009, Lecture Notes in Computer Science, Vol. 5544, Springer, 2009, pp. 994–1003.
[17] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, A. Rupp, M. Schimmler, How to Break DES for €8,980, in: SHARCS2006, Cologne, Germany, 2006.
[18] T. Güneysu, T. Kasper, M. Novotný, C. Paar, A. Rupp, Cryptanalysis with COPACOBANA, IEEE Transactions on Computers 57 (11) (2008) 1498–1513.
[19] Swiss-Prot Protein knowledgebase – ExPASy Proteomics Server, `http://expasy.org/sprot/`.