Accelerating Smith-Waterman Alignment of Long DNA Sequences with OpenCL on FPGA

E. Rucci[1], C. Garcia[2], G. Botella[2], A. De Giusti[1], M. Naiouf[3], and M. Prieto-Matias[2]

[1] III-LIDI, CONICET, Facultad de Informática, Universidad Nacional de La Plata
La Plata (1900), Buenos Aires, Argentina
Email: {erucci,degiusti}@lidi.info.unlp.edu.ar
[2] Depto. Arquitectura de Computadores y Automática,
Universidad Complutense de Madrid, Madrid 28040, Spain
Email: {gbotella,garsanca,mpmatias}@ucm.es
[3] III-LIDI, Facultad de Informática, Universidad Nacional de La Plata
La Plata (1900), Buenos Aires, Argentina
Email: {mnaiouf}@lidi.info.unlp.edu.ar

Abstract. With the greater importance of parallel architectures such as GPUs or Xeon Phi accelerators, the scientific community has developed efficient solutions in the bioinformatics field. In this context, FPGAs begin to stand out as high performance devices with moderate power consumption. This paper presents and evaluates a parallel strategy of the well-known Smith-Waterman algorithm using OpenCL on Intel/Altera's FPGA for long DNA sequences. We efficiently exploit data and pipeline parallelism on a Intel/Altera Stratix V FPGA reaching upto 114 GCUPS in less than 25 watt power requirements.

1 Introduction

In recent years, genome research projects have produced a vast amount of biological data. In fact, biologists are working in conjunction with computer scientists to extract relevant biological information from these experiments. The comparison of millions of sequences [13] is one of the most useful mechanisms known in Bioinformatics, commonly solved by heuristic methods.

Smith-Waterman (SW) algorithm compares two sequence in an exact way and corresponds to the so-called local methods because it focuses on small similar regions only. Besides, this method has been used as the basis for many subsequent algorithms and is often used as basic pattern to compare different alignment techniques. However, one of the main drawbacks is the cost of this approach in computing time and memory requirements which makes it unsuitable in some cases.

The final authenticated version is available online at https://doi.org/10.1007/978-3-319-56154-7_45

Regarding on the performance aspect, many approaches, such as BLAST [1] and FASTA [14] considerably reduce execution time at the expense of not guaranteeing the optimal result. Nevertheless, accelerating the SW algorithm is still a great challenge for the scientific community. SW is usually used to align two DNA sequences or a protein sequence to a genomic database. A single matrix must be built for each alignment and the matrix size depends on sequence lengths. In DNA alignment, the matrix can be huge since these sequences can have upto hundreds of million nucleotides. As protein sequences are shorter, multiple small matrices are usually computed simultaneously since they are independent among them [15].

In the last years, we have witnessed parallel proposal for both cases. These works reduce SW execution time through the exploitation of High-Performance Computing (HPC) architectures. However, most implementations focus on short sequences, particularly protein sequences [17]. For very long sequences, as the DNA case, few implementations are available. In the hardware accelerators scenario, we highlight *SW#* [9] and *CUDAlign* [18] (and its newer versions [4, 5]) that focus on the alignment of huge DNA sequences with multi CUDA-enabled Graphics Processor Units (GPUs). Meanwhile, Liu et al. have presented *SWAPHI-LS* [12] a highly optimized hand-tuned implementation for Intel Xeon Phi accelerators. In addition, several proposals based on FPGA speedup sequence alignments [22, 2] in the context of DNA comparison. Moreover, Wienbrandt presents a study of multi-FPGAs version in [21].

Nowadays, HPC capabilities are changing in data-centers scenario. FPGAs are being integrated with CPUs due to accelerators consolidation in HPC community as a way of improving performance while keeping power efficiency. Recently, Microsoft announced that their data-centers equipped with FPGAs increase dramatically Bing's engine capacities, and they have also incorporated them to Azure ecosystem [6]. In the same way, Amazon has included FPGAs in its Amazon Web Services [10]. With the acquisition of Altera in 2015, Intel plans to incorporate FPGA capabilities in the next Xeon's server processors because they expect to be used by at least 30% percent of data-center in next years [11].

Our paper proposes and evaluates a SW implementation, capable of aligning DNA sequences of unrestricted size, for Altera's FPGA. Unlike previous works on literature, we employed the novel OpenCL paradigm on FPGAs. Although Altera's developers promote a similar implementation in [19], no-real sequence data of fixed, limited size (n=256 residues) are used. As this issue can radically differ from real bioinformatic contexts, the observed behavior becomes unpredictable for these scenarios, especially in alignment of long DNA sequences. We would like to point out that, unlike GPU or Xeon Phi accelerators which need to be purchased separately, the hardware used in this study will be available soon in the next processor generation at null cost. In that sense, this study can be taken as starting point for future hybrid CPU-FPGA solutions.

The rest of the paper is organized as follows. Section 2 describes SW algorithm. Section 3 introduces Altera's OpenCL programming extension while Section 4 addresses our parallelization of the SW algorithm using OpenCL on

FPGAs. Section 5 presents experimental results and finally Section 6 outlines conclusions and future lines of work for this novel viability study.

2 Smith-Waterman Algorithm

The SW algorithm is used to obtain the optimal local alignment between two sequences and was proposed by Smith and Waterman [20]. This method employs a dynamic programming approach and its high sensitivity comes from exploring all the possible alignments between two sequences.

Given two sequences S_1 and S_2, with sizes $|S_1| = m$ and $|S_2| = n$, the recurrence relations for the SW algorithm with affine gap penalties [7] are defined below.

$$H_{i,j} = max\{0, H_{i-1,j-1} + SM(S_1[i], S_2[j]), E_{i,j}, F_{i,j}\} \quad (1)$$

$$E_{i,j} = max\{H_{i,j-1} - G_{oe}, E_{i,j-1} - G_e\} \quad (2)$$

$$F_{i,j} = max\{H_{i-1,j} - G_{oe}, F_{i-1,j} - G_e\} \quad (3)$$

$H_{i,j}$ contains the score for aligning the prefixes $S_1[1..i]$ and $S_2[1..j]$. $E_{i,j}$ and $F_{i,j}$ are the scores of prefix $S_1[1..i]$ aligned to a gap and prefix $S_2[1..j]$ aligned to a gap, respectively. *SM* is the *scoring matrix* which defines match/mismatch scores between nucleotides. G_{oe} is the sum of gap open and gap extension penalties while G_e is the gap extension penalty. The recurrences should be calculated with $1 \leq i \leq m$ and $1 \leq j \leq n$, after initialising H, E and F with 0 when $i = 0$ or $j = 0$. The maximum value in the alignment matrix H is the optimal local alignment score.

It is important to note that any cell of the matrix H can be computed only after the values of the upper, left and upper-left cells are known, as shown in Figure 1. These dependences restrict the ways in that H can be computed.

3 OpenCL Extension on Altera's FPGA

OpenCL is a framework for developing parallel programs across heterogeneous platforms. It is currently supported by several hardware devices such as CPUs, GPUs, DSPs and FPGAs. The OpenCL is based on host-device model, where host is in charge of managing device memory, transferring data from/to device and launching the kernel code.

Kernel corresponds to a piece of code which expresses the parallelism of a program. OpenCL programming model divides a program workload into *work-groups* and *work-items*. Usually, in the denoted *data parallel programming* model, *work-items* are grouped into *work-groups*, which are executed independently on a procesing element. Data-level parallelism is ordinarily exploited in SIMD way, where each *work-item* is mapped to a lane width of the target device.

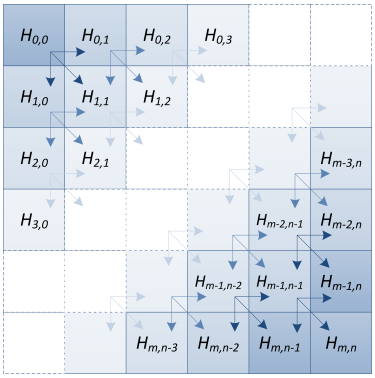

Fig. 1. Data dependences in the alignment matrix H.

OpenCL memory model uses a particular memory herarchy which is also characterized by the access type, performance and scope. Global memory is read-write accessible by all *work-items* which implies a high latency memory access. Local memory is a shared read-write memory that can be accessed from all *work-items* of a single *work-group*. Besides, it usually involves a low latency memory access. Constant memory is a read-only memory visible by all *work-items* across all *work-groups*, and private memory as the name suggests it is only accessible by a single *work-item*.

OpenCL allows a programmer to express parallelism abstracting the target platform details. We can highlight portability and reduction in development time as the main advantages. FPGAs permit programming networks composed of logic elements, memory blocks and specific DSP blocks. In order to verify and create digital designs, Hardware Description Languages (HDLs) are generally used, which are complex, error prone and affected by an extra abstraction layer as they contain the additional concept of time.

Regarding the execution model, Altera's OpenCL SDK [3] recommends the use of *task parallel programming* model, where the kernel consists of a single *work-group* with a unique *work-item*. The Altera OpenCL Compiler (AOC) is capable of extracting parallelism from each loop iteration in a loop-pipelined way allowing to process the loop in a high-throughput fashion.

Altera's OpenCL extension also take advantage of I/O channels and kernel channels as in OpenCL 2.0 by means of pipes [8]. Altera's channel extension allows to transfer data between work-item's in the same kernel or between different kernels without host interaction.

4 SW Implementation

In this section we will address the programming aspects and optimizations applied to our implementations on FPGA accelerated platforms. Algorithm 1 shows the pseudo-code for the host implementation. Memory management is

Algorithm 1 Host pseudo-code

```
1: clCreateBuffer's(...)                          ▷ Create buffers + transfer sequences
2: NB = n/BW                                      ▷ NB is the number of vertical blocks
3: for b ≤ NB do
4:     clEnqueueTask(...)                         ▷ Compute b-th block
5:     swap(prevLastColH,curLastColH)
6:     swap(prevLastColE,curLastColE)
7: end for
8: clEnqueueReadBuffer(maxScore)
```

Algorithm 2 Pseudo-code for Smith-Waterman kernel

```
1: __kernel void SW_kernel ( S1, S2, m, b, match, mismatch, Goe, Ge,
   prevLastColH, curLastColH, prevLastColE, curLastColE, maxScore )
   {
2: Load the BW residues of S2 corresponding to b-th block from global memory
   to private memory
3: for i ≤ m do                                                    ▷ each row
4:     Load the i-th residue of S1 from global memory to private memory
5:     Read previous block data from global memory (prevLastColH and
   prevLastColE)
6:     #pragma unroll
7:     for j ≤ BW do
8:         Calculate H_{i,j} in private memory
9:     end for
10:    Write data for next block to global memory (curLastColH and
   curLastColE)
11: end for
12: Update maxScore in global memory (if appropiate)
13: }
```

performed in OpenCL by means of *clCreateBuffer* (memory allocation and initialization) and *clEnqueueReadBuffer* (memory transfer to host). Kernels are invoked through the *clEnqueueTask* function.

The kernel is implemented following the *task parallel programming* model mentioned in Section 3. Algorithm 2 shows the pseudo-code for our kernel. The alignment matrix is divided into vertical blocks (BW means Block Width) and each block is computed in row-by-row manner: from top to bottom, left to right direction (see Figure 2). Besides improving the data locality, this blocking technique reduces the memory requirements for block execution, which favors the exploitation of the private low-latency memory. In that sense, we employed two buffers to store one row for matrices H and F. Additionally, both sequences are partially copied to private memory.

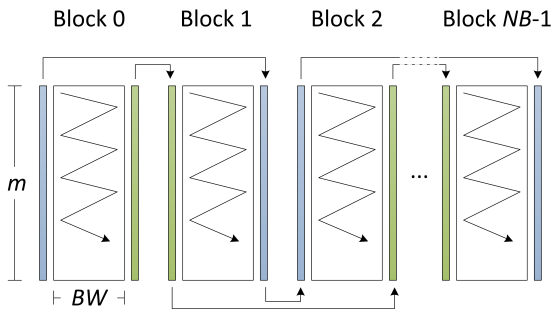

Fig. 2. Schematic representation of our OpenCL kernel implementation.

Fully unrolling of the inner loop represents an essential aspect of this kernel from performance point of view. This technique allows the AOC to exploit loop instruction pipelining and, in consequence, more operations per clock cycle are performed. As the compiler needs to know the number of iterations at compile phase, S_2 sequence must be extended with dummy symbols to make its length a multiple of fixed BW value.

Due to the data dependences mentioned in Section 2, each block needs the last column H and E values of the previous block. Global memory buffers are employed to communicate these data. To avoid read-write dependences in global memory, separate buffers are used: one for reading the values from the previous block and one for writing the values for the next block, so after each kernel invocation buffers are swapped in the host. It is important to mention that, although in *OSWALD* implementation [16] Altera OpenCL channels are used to exchange these information, the use of this technique is not affordable in DNA context with million of nucleotide bases involved.

Moreover, host-side buffers are allocated to be 64-byte aligned. This fact improves data transfer efficiency by means of Direct Memory Access. Both sequence are transferred when creating the device buffers and optimal score is retrieved after all kernels finished.

5 Experimental Results

5.1 Experimental Platforms and Tests Carried Out

Tests have been performed on different platforms running CentOS (release 6.6):

- A server with two Intel Xeon CPU E5-2670 8-core 2.60GHz, 32 GB main memory and an Altera Stratix V GSD5 Half-Length PCIe Board with Dual DDR3 (two banks of 4 GByte DDR3).
- A server with two Intel Xeon CPU E5-2695 v3 16-core 2.30GHz, 64 GB main memory and two NVIDIA GPU cards: one Tesla K20c (Kepler architecture, 2496 CUDA cores, 5GB dedicated memory and Compute Capability 3.5)

Table 1. Information of the sequences used in the tests.

Sequence 1		Sequence 2		Matrix size	Score
Accesion	Size	Accesion	Size		
AF133821.1	10K	AY352275.1	10K	10K × 10K	5027
NC_001715.1	57K	AF494279.1	57K	57K × 57K	51
NC_000898	162K	NC_007605	172K	162K × 172K	18
NC_003064.2	543K	NC_000914.1	536K	543K × 536K	48
CP000051.1	1M	AE002160.2	1M	1M × 1M	82091
BA000035.2	3M	BX927147.1	3M	3M × 3M	3888
AE016879.1	5M	AE017225.1	5M	5M × 5M	5220775
NC_005027.1	7M	NC_003997.3	5M	7M × 5M	157
NC_017186.1	10M	NC_014318.1	10M	10M × 10M	10235056
NT_033779.4	23M	NT_037436.3	25M	23M × 25M	9059

and one GTX 980 (Maxwell architecture, 2048 CUDA cores, 4GB dedicated memory and Compute Capability 5.0).
- A server with two Intel Xeon CPU E5-2695 v3 16-core 2.30GHz, 128 GB main memory and a single Xeon Phi 3120P coprocessor card (57 cores with 4 hw thread per core, 6GB dedicated memory).

We have used the Intel's ICC compiler (version 16.0.3) with the *-O3* optimization level by default. The synthesis tool used is Quartus II DKE V12.0 2 with OpenCL SDK v14.0 and the CUDA SDK version is 7.5.

To provide the most relevant study, tests were made with the real DNA sequences retrieved from the National Center for Biotechnology Information (NCBI) [4], ranging from thousands to millions of nucleotide bases. The accession numbers and sizes of the sequences are presented in Table 1. Also, for the sake of validation, optimal alignment scores are included in Table 1. The scoring scheme used was: +1 for match; -3 for mismatch; -5 for gap open; and -2 for gap extension. Each particular test was run ten times and performance was calculated with the average of ten execution times to avoid variability.

5.2 Performance and Resource Usage Evaluation

The metric GCUPS is used to performance assessment in the Smith-Waterman scenario [18]. In order to evaluate FPGA performance rates, we have considered different kernel implementations according to integer data type and BW value. We detail below the main differences:

- The name prefix denotes the integer data type used; i.e. *int*, *short* and *char* represent 32, 16 and 8 bit integer data types, respectively.
- The name suffix denotes the BW value used; e.g. *bw32* means that BW value was set to 32.

[4] Sequences are available in `http://www.ncbi.nlm.nih.gov`

Table 2. Performance and resource usage comparison for different OpenCL kernel implementations.

Kernel		*int_bw32*	*int_bw64*	*int_bw128*	*int_bw256*	*short_bw256*	*short_bw512*	*char_bw512*	*char_bw768*
Integer type		int (32 bits)				short (16 bits)		char (8 bits)	
Maximum value		2147483647				32767		127	
BW		32	64	128	256	256	512	512	768
Resource usage	ALMs	30%	36%	48%	69%	50%	76%	49%	68%
	Regs	11%	12%	12%	12%	10%	13%	10%	15%
	RAM	22%	22%	22%	25%	21%	24%	20%	33%
	DSPs	0%	0%	0%	0%	0%	0%	0%	0%
Matrix size	10K × 10K	2.22	4.60	7.45	14.47	18.43	23.23	-	-
	57K × 57K	4.27	8.42	16.32	30.35	38.01	58.21	73.07	91.89
	162K × 172K	4.75	9.33	18.04	33.45	41.85	66.34	84.67	109.29
	543K × 536K	4.97	9.66	18.86	35.51	43.36	70.21	86.01	113.78
	1M × 1M	5.12	9.93	19.44	36.52	-	-	-	-
	3M × 3M	5.24	10.14	19.87	37.32	45.58	73.39	-	-
	5M × 5M	5.27	10.18	19.93	37.49	-	-	-	-
	7M × 5M	5.28	10.20	20	37.56	45.86	73.73	-	-
	10M × 10M	5.28	10.21	20.03	37.61	-	-	-	-
	23M × 25M	5.29	10.23	20.07	37.68	46.01	74	-	-
		GCUPS							

Table 2 presents FPGA resource utilization and the performance achieved for our OpenCL kernels implementations. Larger BW means better performance but also higher resource consumption. Adaptive logic modules (ALMs) are the most influenced resources; registers (Regs) and RAM blocks (RAMs) are slightly increased while DSP blocks (DSPs) stay intact. In spite of the fact there are still available resources to allow increasing BW parameter, larger values could not be used because AOC reports the appearence of read-write dependences in private memory associated to matrices H and F. In fact, these dependences considerably reduce the number of operations per clock cycle decreasing performance rates.

Regarding integer data type, as can be observed, smaller data type not only improves performance but also decreases resource consumption. This behavior is clearly exposed when comparing *int_bw256* and *short_bw256* kernels: using the same BW configuration, *short_bw256* reports an increase from up to 1.22× in performance with a reduction from up to 0−0.28× in resource usage against to *int_bw256* counterpart. A similar behavior is observed with *short_bw512* and *char_bw512* kernels: *char_bw512* presents an increase from up to 1.28× in performance with a reduction from up to 0−0.36× in resource usage respect to *short_bw512*. However, it is also important to take into account that the use of narrower integer data types also involves an important reduction in representation range. Due to this fact, there are three alignment scores out of ten that can not be represented when using 16 bit integer data. It is also observed for the experiments with 8-bits data type where only three experiments can be carried out [5].

[5] The symbol '-' indicates an alignment that can not be computed because the optimal score exceeds the corresponding maximum value.

From sequence length point of view, all kernels benefit from larger workloads regardless of sequences similarity. The best performances achieved are 37.68, 74 and 113.78 GCUPS for *int*, *short* and *char* kernels, respectively.

5.3 Performance and Power Efficiency Comparison with other SW Implementations

In this subsection the behavior of our implementation is compared with other SW implementations: the Xeon Phi-based *SWAPHI-LS* program (v1.0.12) [12] and the GPU-based *SW#* [9] and *CUDAlign* (v3.9.1.1024) [4] programs. Both GPU implementations were configured to perform only their score version.

Table 3 presents performances for the *SWAPHI-LS*, *SW#* and *CUDAlign* implementations. *SWAPHI-LS* yields an average performance of 25.89 GCUPS and a peak of 34.38 GCUPS being outperformed by *int_bw256* version in all cases. In particular, the most impressive performance difference occurs for smaller matrix sizes where *int_bw256* runs on average 21.8× faster. For the rest of the tests, the performance gain decreases but still improves on average 1.45×.

Unlike our FPGA kernels, both GPU implementations are sensitive to sequences similarity; better results are obtained on alignments with higher scores. On Tesla K20c, *SW#* achieves an average performance of 44.38 GCUPS and a maximum performance of 90.15 GCUPS, outperforming *CUDAlign* by a factor of 1.78× on average. *int_bw256* improves *SW#* on the three shortest alignments while the latter runs 1.62× faster on average than the former on the largest ones. In contrast, *SW#* only beats *CUDAlign* on the four shortest alignments on GTX 980. *CUDAlign* reports 71.23 GCUPS on average and a peak of 163.77 GCUPS, achieving an average speedup of 1.24× on the six largest matrix sizes. Just as the previous case, *int_bw256* improves both GPU implementations on the three shortest alignments while is outperformed in the rest. In that sense, *CUDAlign* runs 2.76× faster on average than *int_bw256*.

In FPGA context, a theoretical comparison with other OpenCL implementations is the only possibility since the absence of available source codes. We would like to note that the Altera staff implementation [19] uses 32-bit integer data type and imposes a fixed, limited size (n=256 residues) to S_2 sequence, which substantially differs from the common ones used in DNA analysis. In spite of this fact, Altera's staff implementation reported a peak of 24.7 GCUPS on Stratix V meanwhile our approach achieves an average of 33.8 GCUPs.

We have also analyzed the power efficiency of each implementation. Table 4 presents power efficiency ratios considering the GCPUS peak performance and Thermal Design Power (TDP) in each accelerator. It can be seen that the worst ratio observed is in *SWAPHI-LS* due to low performance rates and high TDP value of the Xeon Phi. GPU implementations place at an intermediate position because they obtain the highest performance peak but at the expense of higher power requirements. As expected, both GPU implementations obtain better GCUPS/Watts ratios on GTX 980 in comparison with Tesla K20c since its better performance and power capacities. We would like to remark that FPGA implementation reaches the best GCUPS/Watts ratios. Despite not having the

Table 3. Performance of other SW implementations.

Implementation	*SWAPHI-LS*	*SW#*	*CUDAlign*	*SW#*	*CUDAlign*
Device	Intel Xeon Phi 3120P	NVIDIA Tesla K20c		NVIDIA GTX 980	
10K × 10K	0.42	0.16	0.06	0.3	0.03
57K × 57K	7.69	5.86	1.57	7.62	1.08
162K × 172K	21.24	32.78	10.23	33.33	8.18
543K × 536K	30.67	42.36	29.71	64.53	45.89
1M × 1M	32.84	51.01	39.64	75.24	79.21
3M × 3M	33.9	43.48	39.73	69.54	84.05
5M × 5M	34.16	90.15	79.53	120.92	160.79
7M × 5M	34.38	43.64	39.55	68.84	84.43
10M × 10M	33.19	90.22	79.96	118.81	163.77
23M × 25M	30.36	44.19	40.69	67.55	84.84
Matrix size	GCUPS				

Table 4. Power efficiency comparison.

Implementation	Device	GCUPS	Watts	GCUPS/Watts
int_bw256	Altera Stratix V	37.68	25	1.51
short_bw512		74		2.96
char_bw768		113.78		4.55
SWAPHI-LS	Intel Xeon Phi 3120P	34.38	270	0.13
SW#	NVIDIA Tesla K20c	90.22	225	0.4
CUDAlign		79.96		0.36
SW#	NVIDIA GTX 980	120.92	165	0.73
CUDAlign		160.79		0.97

highest performance peak, its low power consumption leads to the best choice in this aspect. Further, the lowest performance FPGA kernel considering integer data type (*int_bw256*) outperforms *SWAPHI-LS* by a factor of 11.62× and the GPU implementations by a range of 1.56-3.78×. It is important to mention that if this analysis is carried out considering average GCUPS instead of GCUPS peak, larger differences in favor of FPGA implementations will be found.

6 Conclusions

In this paper, we addressed the benefits of a parallel SW implementation using OpenCL on Intel/Altera's FPGA, not only from performance perspective but also from power efficiency point of view. To the best of the author's knowledge, this is the first time that a paper examines an implementation of this kind with real, long DNA sequences. In addition, as Intel will incorporate FPGA capabilities in its next Xeon processors in a free manner, this study can be useful as an starting point for future hybrid CPU-FPGA implementations.

Among main contributions of this research we can summarize:

- Data type exploitation is crucial to achieve successful performance rates. Narrower data-types reports better performance rates and lesser resource usage, but at expense of decreasing representation width. In fact, a peak of 114 GCUPS is reached when using 8 bit integers.
- From performance perspective, our most successful 32-bit implementation reaches 37.7 GCUPS peak, running 1.53× faster than other OpenCL implementation on FPGA [19]. Despite being competitive in performance terms respect to other solutions on accelerators (such as GPUs or Xeon Phi), our implementation significantly improves all of them from power efficiency perspective. In particular, the fastest 32-bit FPGA kernel outperforms *SWAPHI-LS* by a factor of 11.62× and the GPU implementations by a range of 1.56-3.78×.

Taking into account these encouraging results, as future lines we will consider three aspects:

- As all kernels developed still have free available resources, we will try to exploit them to improve performance rates. On one hand, we expect to solve performance limitations imposed by AOC with larger BW values. On the other hand, we plan to combine distinct integer data width kernels to explore different configurations in order to find the best performance-representation width trade-off.
- As OpenCL allows multiple devices exploitation, we would like to extend this work to a multi-FPGA implementation and explore most successful workload distribution.
- As real power consumption on accelerators can differ from TDP values due to a variety of reasons, we plan to measure the instant power consumption in all devices in order to make a more fairly performance vs power analysis.

Acknowledgments This work has been partially supported by Spanish government through research contract TIN2015-65277-R and CAPAP-H5 network (TIN2014-53522).

References

1. Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped blast and psiblast: a new generation of protein database search programs. *NUCLEIC ACIDS RESEARCH*, 25(17):3389–3402, 1997.
2. Gabriel Caffarena, Carlos E. Pedreira, Carlos Carreras, Slobodan Bojanic, and Octavio Nieto-Taladriz. FPGA Acceleration for DNA Sequence Alignment. *Journal of Circuits, Systems, and Computers*, 16(2):245–266, 2007.
3. Altera Corporation. Altera SDK for OpenCL Programming Guide, v14.0, 2014.

4. Edans F. de O. Sandes, Guillermo Miranda, Alba Cristina Magalhaes Alves de Melo, Xavier Martorell, and Eduard Ayguad. CUDAlign 3.0: Parallel Biological Sequence Comparison in Large GPU Clusters. In *CCGRID*, pages 160–169. IEEE Computer Society, 2014.
5. Edans Flavius de Oliveira Sandes, Guillermo Miranda, Xavier Martorell, Eduard Ayguad, George Teodoro, and Alba Cristina Magalhaes Alves de Melo. CUDAlign 4.0: Incremental Speculative Traceback for Exact Chromosome-Wide Alignment in GPU Clusters. *IEEE Trans. Parallel Distrib. Syst.*, 27(10):2838–2850, 2016.
6. Michael Feldman. Microsoft Goes All in for FPGAs to Build Out AI Cloud, 2016. `https://www.top500.org/news/microsoft-goes-all-in-for-fpgas-to-build-out-cloud-based-ai/`.
7. O. Gotoh. An improved algorithm for matching biological sequences. In *Journal of Molecular Biology*, volume 162, pages 705–708, 1981.
8. Khronos Group. The OpenCL Specification. version 2.0, 2014.
9. Matija Korpar and Mile Sikic. SW# - GPU-enabled exact alignments on genome scale. *Bioinformatics*, 29(19):2494–2495, 2013.
10. George Leopold. AWS Embraces FPGAs, Elastic GPUs, 2016. `https://www.hpcwire.com/2016/12/02/aws-embraces-fpgas-elastic-gpus/`.
11. George Leopold. Intels FPGAs Target Datacenters, Networking, 2016. `https://www.hpcwire.com/2016/10/06/intels-fpgas-target-datacenters-networking/`.
12. Y. Liu, T. T. Tran, F. Lauenroth, and B. Schmidt. SWAPHI-LS: Smith-Waterman Algorithm on Xeon Phi coprocessors for Long DNA Sequences. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 257–265, 2014.
13. David W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Mount, Bioinformatics. Cold Spring Harbor Laboratory Press, 2004.
14. W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, April 1988.
15. Enzo Rucci, Carlos Garca, Guillermo Botella, Armando De Giusti, Marcelo Naiouf, and Manuel Prieto-Matas. An energy-aware performance analysis of SWIMM: SmithWaterman implementation on Intel's Multicore and Manycore architectures. *Concurrency and Computation: Practice and Experience*, 27(18):5517–5537, 2015.
16. Enzo Rucci, Carlos Garca, Guillermo Botella, Armando De Giusti, Marcelo Naiouf, and Manuel Prieto-Matas. OSWALD: OpenCL Smith-Waterman Algorithm on Altera FPGA for Large Protein Databases. *International Journal of High Performance Computing Applications*, page 1094342016654215, 06 2016.
17. Enzo Rucci, Carlos García, Guillermo Botella, Armando De Giusti, Marcelo Naiouf, and Manuel Prieto-Matías. *State-of-the-Art in Smith–Waterman Protein Database Search on HPC Platforms*, pages 197–223. Springer, 2016.
18. Edans Flavius O. Sandes and Alba Cristina M.A. de Melo. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, PPoPP '10, pages 137–146, New York, NY, USA, 2010. ACM.
19. Sean O Settle. High-performance Dynamic Programming on FPGAs with OpenCL. In *2013 IEEE High Performance Extreme Computing Conference(HPEC '13)*, pages 1–6, 2013.
20. Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

21. Lars Wienbrandt. *Bioinformatics Applications on the FPGA-Based High-Performance Computer RIVYERA*, pages 81–103. Springer New York, New York, NY, 2013.
22. Y. Yamaguchi, H. K. Tsoi, W. Luk, A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, and T. El-Ghazawi. *FPGA-Based Smith-Waterman Algorithm: Analysis and Novel Design*, pages 181–192. Springer Berlin Heidelberg, 2011.