International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–14
© The Author(s) 2016
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342016654215
hpc.sagepub.com
SAGE

OSWALD: OpenCL *S*mith–*W*aterman on *A*ltera's FPGA for *L*arge Protein *D*atabases

Enzo Rucci[1], Carlos Garcia[2], Guillermo Botella[2], Armando E De Giusti[1], Marcelo Naiouf[1] and Manuel Prieto-Matias[2]

Abstract
The well-known Smith–Waterman algorithm is a high-sensitivity method for local sequence alignment. Unfortunately, the Smith–Waterman algorithm has quadratic time complexity, which makes it computationally demanding for large protein databases. In this paper, we present OSWALD, a portable, fully functional and general implementation to accelerate Smith–Waterman database searches in heterogeneous platforms based on Altera's FPGA. OSWALD exploits OpenMP multithreading and SIMD computing through SSE and AVX2 extensions on the host while taking advantage of pipeline and vectorial parallelism by way of OpenCL on the FPGAs. Performance evaluations on two different heterogeneous architectures with real amino acid datasets show that OSWALD is competitive in comparison with other top-performing Smith–Waterman implementations, attaining up to 442 GCUPS peak with the best GCUPS/watts ratio.

Keywords
Bioinformatics, Smith–Waterman, FPGA, Altera, OpenCL

1 Introduction

High-throughput structural genomic and genome sequencing have provided the scientific community with a huge amount of data to be processed from structures and sequences of many thousands of proteins. These 'big datasets' can assist researchers in obtaining useful and functional insights. One of the main computational approaches is bioinformatics, which uses statistical analysis of structures and protein sequences to identify the genome, recognize function and, additionally, anticipate structures when only sequence information is available. Bioinformatics has become one of the most powerful technologies in the life sciences nowadays; among other important applications, it is being used in research into evolution theories and protein design.

Sequence alignment is a common task in bioinformatics, and can be considered the basis of other biological tools. This procedure is used to compare primary biological sequence information, such as the amino acid sequences of different proteins or the nucleotides of DNA sequences.

The Smith–Waterman algorithm is the most accurate method for local sequence alignment; its high sensitivity comes from exploring all possible alignments between two sequences. This algorithm focuses on similar regions only in parts of the sequences, which means that the purpose of the algorithm is finding small, locally similar regions.

To calculate optimal local alignment scores, the Smith–Waterman algorithm has a linear space complexity and a quadratic time complexity.

Considering the performance aspect, the Smith–Waterman computation time may become impracticable, owing to its high complexity, especially with large-volume datasets. For this reason, several heuristics, such as BLAST (Altschul et al., 1990) and FASTA (Lipman and Pearson, 1985), have been developed to reduce the execution time, but these come at the expense of not guaranteeing discovery of optimal local alignments. Because of the computational cost of the

[1]Instituto de Investigacion en Informatica LIDI, Universidad Nacional de La Plata, Argentina
[2]Depto. Arquitectura de Computadores y Automatica, Universidad Complutense de Madrid, Spain

Corresponding author:
Carlos Garcia, Depto. Arquitectura de Computadores y Automatica Universidad Complutense de Madrid, Madrid 28040, Spain.
Email: garsanca@ucm.es

Smith–Waterman algorithm, the scientific community has made great efforts to design more efficient implementations in recent years. Most of the solutions proposed find and exploit the inherent parallelism in the alignment process as intratask and intertask parallelism (Rognes, 2011).

With the recent emergence of accelerator technologies, such as field-programmable gate arrays (FPGAs), vector processing units (SIMD) in many-core architectures and graphics processing units (GPUs), among others, the challenge of accelerating life science analysis problems has become more stimulating. Moreover, the affordability of these devices means that their exploitation is becoming an attractive solution.

As related work, we found a hybrid implementation of the Smith–Waterman algorithm (Qiu et al., 2010) that makes use of cloud computing and a cluster programmed with MPI. Moreover, there exist Smith–Waterman versions based on SIMD vector exploitation (Farrar, 2007; Rognes, 2011) which are now available on modern CPUs. In the field of heterogeneous computing, Farrar (2008) makes use of the outdated Cell/BE processors. Also, in the hardware accelerators scenario, the most successful solution is the *CUDASW++* software, and its newer versions (Liu et al., 2009, 2010, 2013), which offer a performance range from 30 to 185.6 GCUPS (billion cell updates per second) for single and multi-CUDA-enabled GPUs with concurrent CPU computing. More recently, an optimized hand-tuned Smith–Waterman implementations for Intel Xeon Phi coprocessors have been produced (Liu and Schmidt, 2014; Liu et al., 2014), denoted *SWAPHI* and *SWAPHI-LS* for protein and DNA sequence alignment, respectively. While *SWAPHI-LS* is able to achieve 30.1 GCUPS, *SWAPHI* runs at up to 58.8 GCUPS. Besides considering Intel Xeon Phi exploitation, Rucci et al. (2014, 2015b) have recently studied both the performance aspect and the energy footprint of a hybrid implementation that exploits both CPU and coprocessors simultaneously.

Weaver et al. (2003); Dydel and Bala (2004); Li et al. (2007); Yamaguchi et al. (2011) and Isa et al. (2011) present Smith–Waterman implementations on a FPGA. However, most of this software implements DNA alignment (which is simpler than protein alignment from an algorithmic perspective) or covers special cases in Smith–Waterman alignment (for example, query or database sequences of limited or fixed length and embedded sequences in the design). In addition, these implementations are based on hardware description languages, such as VHDL or Verilog, which limit their portability to other parallel devices. Under this premise, Altera promotes its FPGA usage through its support of the Open Computing Language (OpenCL) model (Khronos Groups. OpenCL: https://www.khronos.org/opencl), traditionally used in heterogeneous computing environments based on multicore and GPUs. Despite Altera staff having submitted an implementation of Smith–Waterman with OpenCL (Settle, 2014), their implementation focuses on non-real RNA sequence alignment with fixed query length.

Although previous studies have focused on exploiting the FPGA, to the best of our knowledge, our approach is the first high-level programming implementation on FPGAs using OpenCL with real amino acid datasets. Our implementation is a fully functional solution for any sequence length and is general for FPGA-based platforms with different hardware characteristics. This paper extends the insights already offered in our previous approach (Rucci et al., 2015a), with the following new contributions:

1. Among the main contributions, we can highlight the creation of a public git repository with the binary executable developed for this paper, denoted OSWALD (available online at https://github.com/enzorucci/OSWALD). OSWALD is a software to accelerate the well-known Smith–Waterman algorithm on heterogeneous platforms based on Altera's FPGA by means of high-level programming using OpenCL. OSWALD exploits OpenMP multithreading and SIMD computing through SSE and AVX2 extensions on the host, while taking advantage of pipeline and vectorial parallelism on FPGAs.
2. Regarding the original implementation, we have focused on OpenCL kernel optimization through FPGA resource stressing. The analysis includes a performance and resource usage evaluation of different kernel implementations.
3. We have extended the optimized single-FPGA implementation to allow multiple FPGAs and, subsequently, to support concurrent host computation. Performance was evaluated using two different protein databases over two heterogeneous architectures.
4. In addition, we have compared our hybrid CPU–FPGA implementation with other reference implementations. For this purpose, we chose the best performing CPU-based, Xeon Phi-based and GPU-based alternatives: *SWIMM* (Rucci et al., 2015b) was selected for Xeon and Xeon Phi processors while CUDASW++ 3.0 (Liu et al., 2013) was chosen for CUDA-compatible GPUs.
5. Finally, this paper not only focuses on performance analysis of FPGA-based architectures, but also considers power consumption. It explores different configurations to find the fastest performance, lowest power consumption and best performance-to-power ratio.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts of the Smith–Waterman algorithm, Section 3 introduces Altera's OpenCL programming extension and in Section 4 the methodology to program this alignment efficiently through different optimization techniques is described. Section 5 presents the results obtained; finally Section 6 outlines conclusions and future lines of work for this novel viability study.

2 Smith–Waterman algorithm

Smith and Waterman (1981) proposed an algorithm to find the optimal local alignment of two sequences. This algorithm is based on dynamic programming and was later improved by Gotoh (1982). The Smith–Waterman method guarantees optimal alignment because it explores all possible alignments between a pair of sequences.

To compute the optimal alignment of two sequences $q = q_1q_2q_3 \dots q_m$ and $d = d_1d_2d_3 \dots d_n$, the Smith–Waterman algorithm fills a matrix H, which keeps track of the degree of similarity between the two sequences compared. The matrix is computed according to the recurrence relations defined as

$$H_{i,j} = \max\begin{cases} 0 \\ H_{i-1,j-1} + SM(q_i, d_j) \\ E_{i,j} \\ F_{i,j} \end{cases} \tag{1}$$

$$E_{i,j} = \max\begin{cases} H_{i,j-1} - G_{\mathrm{oe}} \\ E_{i,j-1} - G_{\mathrm{e}} \end{cases} \tag{2}$$

$$F_{i,j} = \max\begin{cases} H_{i-1,j} - G_{\mathrm{oe}} \\ F_{i-1,j} - G_{\mathrm{e}} \end{cases} \tag{3}$$

where $H_{i,j}$ represents the score for aligning the prefixes of q and d ending at position i and j, respectively. $E_{i,j}$ and $F_{i,j}$ are the scores ending with a gap involving the first i residues of q and the first j residues of d, respectively. SM is the *substitution matrix*, which defines the substitution scores for all residue pairs. Generally, SM rewards with a positive value when q_i and d_j are identical or relatives, and punishes with a negative value otherwise. G_{oe} is the sum of gap open and gap extension penalties, while G_{e} is the gap extension penalty. Recurrences should be calculated with $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$, after initializing H, E and F with 0 when $i = 0$ or $j = 0$. The maximal alignment score in the matrix H is the optimal local alignment score S.

It is important to note that any cell of the matrix H has a dependency on three cells: the one to the left, the one above and the one from the upper left diagonal, as illustrated in Figure 1. So computation must advance from top to bottom and from left to right.

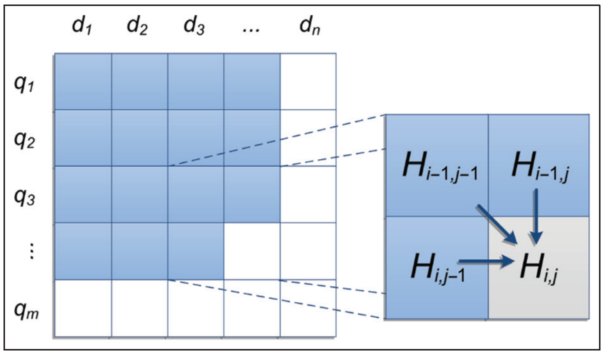

Figure 1. Data dependences in the alignment matrix H.

3 OpenCL extension of Altera's FPGA

OpenCL is a host-device-based framework for parallel implementation, working across heterogeneous platforms. The language is based on the C programming language and contains extensions that allow for the specification of parallelism. Nowadays, it is supported by most hardware devices, such as CPUs, GPUs, DSPs, and FPGAs. These devices (acting as coprocessors or accelerators) may have different instruction set architectures and may share memory with a host processor. OpenCL programming interfaces consider the heterogeneity between the host CPU and all connected devices.

The host–device model administers the following issues:

(a) the use of different contexts for specifically available accelerators;
(b) the management of memory transfers, controlling memory allocations;
(c) the compilation of OpenCL codes and kernel cores to be executed on target devices;
(d) the launch of the kernels on target devices, querying execution progress and checking for errors produced.

An OpenCL kernel is the basic unit of parallel code that can be executed on a target device. OpenCL organizes a program workload into *work-groups* and *work-items*. *Work-items* are grouped into a *work-group*, which are executed independently with respect to other *work-groups*. Data-level parallelism is regularly exploited by means of the SIMD philosophy, where several *work-items* are grouped according to the lane width capabilities of the target device.

The OpenCL memory model deals with different memory regions that are characterized by access type, performance and scope. Global memory is read–write accessible by all *work-items* across all *work-groups*, and usually corresponds to the DRAM memory device, which carries high-latency memory access but high capacity. Local memory is a shared read–write memory accessible from all *work-items* of a single *work-group*; it usually involves low-latency memory access. Constant memory is a read-only memory that is visible to all *work-items* across all *work-groups*; private memory is only accessible by a single *work-item*.

Since OpenCL is a cross platform standard for parallel programming, (oriented to heterogeneity between the host and connected devices, as mentioned), the developer can thus focus on behavioural algorithmic specifications, avoiding implementation details. On the one hand, the OpenCL specification defines a platform, memory and programming model that permits many add-ons that are vendor-specific, cross-vendor and from the Khronos consortium. There is considerable freedom in terms of implementing the platform, providing the final implementation satisfies the OpenCL specifications (Altera, 2014). On the other hand, FPGAs present programmable arrays containing logic elements, memory blocks and specific DSP blocks. This fact enables the design of dynamic custom instruction pipelines against fixed data-path architectures of CPUs, DSPs and GPUs. Hardware description languages (HDLs), such as VHDL or Verilog used to develop and verify FPGA designs are complex, error prone and affected by an extra abstraction layer, as they contain the additional concept of timing.

The main advantage of FPGA-based implementations using the OpenCL paradigm is the shorter time to market and faster developments in comparison with traditional FPGA developments using HDLs. FPGAs are dedicated coprocessor accelerators that contain a complex hierarchy memory model (see Table 1, particularized for the FPGA used in this research). The host processor is connected to the accelerators through a peripheral interface, such as a PCIe.

The OpenCL Altera (FPGA vendor) SDK supports the 1.0 specification, which is a subset of the 2.1 profile (current as of March 2015) with some flexible requirements and advanced features. As an example of these extensions, we can point to the advantage of using I/O channels and kernel channels by means of pipes (Khronos Group, 2014), which appeared in OpenCL 2.0. Altera's channel extension allows the transfer of data between work-items in the same kernel or between different kernels by means of a FIFO buffer. This fact makes it possible to pass data to another work-group without additional synchronization and without host interaction.

Table 1. OpenCL memory model for FPGAs.

OpenCL memory	FPGA memory	BittWare S5PHQ
Global	External	2×4 GB DDR3
Constant	Cache	16 kB DDR3
Local	Embedded	44 Mbits
Private	Registers	674 kbits

Each Altera FPGA can have multiple in-order command queues, to be executed independently and concurrently. Kernels are compiled previously and are then passed to create the OpenCL program object at runtime. Regarding the execution model, it is possible to use the *work-item* ordering within a pipeline, outperforming the obtained throughput thanks to this topology. The OpenCL paradigm model defines the execution of an instance of a kernel by a *work-item* up to *NDRange*. Kernels are executed across a global domain of *work-items*, where *work-items* are subsequently grouped into local *work-groups*. The execution model does not specify the *work-item* execution order.

4 Smith–Waterman implementation

In this section, we will address the programming aspects and optimizations applied to our implementations on FPGA-accelerated platforms. First, we present a heterogeneous implementation, in which alignments are carried out on a single FPGA. Then, this implementation is extended to support more than one FPGA. The final implementation concurrently exploits both host computing and FPGA devices. The algorithms comprise three stages.

1. *Preprocessing stage*. The reference database is preprocessed to adapt sequence data for parallel processing on multiple devices.
2. *Smith–Waterman stage*. After preprocessing the database, alignments among query sequences and database sequences are carried out.
3. *Sorting stage*. Finally, all alignment scores are sorted in descending order.

It is important to note that stages 1 and 3 are executed on the host in all the implementations developed. Stage 2 is offloaded to the FPGA(s) and partially computed on the host in the hybrid version.

4.1 Parallelization scheme

Alignments are computed following the intertask parallelization scheme, which takes advantage of the null data dependency between different alignments. Instead of aligning one database sequence against a query sequence at a time, multiple database sequences are aligned in parallel by means of the SIMD vector capabilities available on the target platform. For this reason, database sequences are processed in groups; the size of the groups is determined by the number of SIMD vector lanes. On the host, database sequences are grouped according to the vector processing unit's lane size. On the FPGA, it is possible to configure the number of sequences that are processed simultaneously. This aspect depends on the resources available on the FPGA.

4.2 Database preprocessing

Database sequences are sorted by their lengths in ascending order before being grouped and padded with dummy symbols. This is done to favour memory pattern access and minimize imbalances in group processing. In the FPGA implementations, the database is divided into chunks because FPGA global memory is not large and the sequence allocation space is limited. Moreover, the number of chunks should be a multiple of the number of FPGAs, and should also have the same size, to improve workload balance between accelerators.

We would like to point out that in the hybrid implementation, the database is split into two main parts to enable a balanced workload distribution. This strategy is described in the hybrid implementation section. The host database part is performed as a single piece, while on the accelerators it is divided again into several chunks, following the approach of the FPGA implementations. To avoid repeating this process, sequence databases are preprocessed separately on the host and accelerators. The databases are read from the FASTA format (http://blast.ncbi.nlm.nih.gov/blastcgihelp.shtml) and then transformed into an internal binary format that favours faster disk access.

4.3 Heterogeneous single-FPGA implementation

Algorithm 1 shows the pseudo-code for the host implementation. Memory management is performed in OpenCL by means of *clCreateBuffer* (memory allocation and initialization), *clEnqueueWriteBuffer* and *clEnqueueReadBuffer* (memory transfer to device or host). The kernel computes the alignments between a single query and a chunk of the sequence database. Kernels are invoked through the *clEnqueueNDRangeKernel* function.

The kernel is implemented following the task parallel programming model described in the OpenCL 1.0 specification, where the kernel consists of a single *work-group* that contains a unique *work-item*. This scheme is suitable because a single *work-item* does not require any synchronization stage. Algorithm 2 shows the pseudo-code for our kernel implementation. The alignment matrix is divided into vertical blocks and computed in a row-by-row manner (see Figure 2). This blocking technique not only improves data locality but also reduces the memory requirements for computing a block, which favours use of the private low-latency memory. The inner loop is fully unrolled by the compiler to increase performance. Since the compiler can

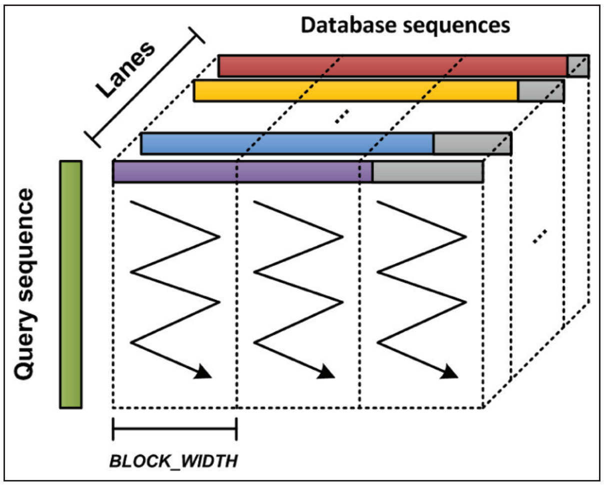

Figure 2. Schematic representation of our OpenCL kernel implementation.

perform loop unrolling, its boundaries must be constant values, so sequences are extended in the preprocessing stage to make their lengths a multiple of the fixed *BLOCK_WIDTH* value.

Additionally, we employed Altera OpenCL channels to efficiently transfer previously computed values in order to solve data dependences between blocks (last column H and E values are needed). The combination of these techniques is essential for the Altera OpenCL compiler to successfully generate parallel pipeline execution.

4.3.1 Substitution score selection. Our implementation is also based on score profile (*SP*) optimization (Rognes, 2011) to obtain scores from the substitution matrix. This technique involves constructing an auxiliary $n \times l \times |\sum|$ two-dimensional score array, where n is the length of the database sequence, l is the number of vector lanes and $\sum$ is the alphabet. Since each row of the score profile forms an l-lane score vector, its values can be loaded in parallel. To reduce FPGA hardware resource usage, the score profiles are built on the host using a set of SSE intrinsic functions and then transferred to the FPGA.

4.3.2 Data type selection. Optimizing FPGA area usage is critical to obtaining high-performance OpenCL applications. The alignment scores do not need wide-range data representation. For this reason, we explored different integer data types to compute alignments (*char, short* and *int*). When the data type proves to be insufficient to represent the similarity score, i.e. overflow occurs, the alignment is recalculated on the host using the next widest integer range. The host code employs SSE instructions and is based on the open-source SWIMM tool (Rucci et al., 2015b). To allow

Algorithm 1. Host pseudo-code for single-FPGA exploitation.

```
1:  ▷ $Q$ are the query sequences
2:  ▷ $vD$ is the preprocessed sequence database
3:  ▷ $SP$ are the Score Profiles
4:  ▷ $SM$ is the substitution matrix
5:  ▷ $S$ are the alignment scores
6:  ▷ $n$ are the lengths of database sequences
7:  ▷ $t_h$ is the number of host threads
8:
9:  clCreateBuffer's(...) ▷ Create buffers + transfer data
10: for $c \leqslant get\_num\_chunks(vD)$ do
11:     $SP_c = build\_SPs(vD_c, SM, t_h)$
12:     clEnqueueWriteBuffer($SP_c$) ▷ Score Profiles to device
13:     clEnqueueWriteBuffer($n_c$) ▷ Sequence lengths to device
14:     for $q \leqslant get\_num\_sequences(Q)$ do
15:         clEnqueueNDRangeKernel(...) ▷ Compute alignments among query $q$ and chunk $c$
16:     end for
17:     clEnqueueReadBuffer($S_c$)
18: end for
19: $S = recompute\_if\_overflow(S, Q, vD, SM, t_h)$ ▷ Recompute alignments that overflowed
20: $S = sort(S, t_h)$ ▷ Sort all scores in descending order
```

Algorithm 2. Pseudo-code for Smith–Waterman kernel.

```
1:  ▷ $numSequences$ is the number of sequences
2:  ▷ $q$ is the query sequence
3:  ▷ $m$ is the query length
4:
5:  __kernel void $SW\_kernel$ ( $numSequences, n, SP, q, m, S$ ) {
6:  for $s \leqslant numSequences$ do
7:      $numBlocks = n[s]/BLOCK\_WIDTH$
8:      for $k \leqslant numBlocks$ do
9:          for $i \leqslant m$ do ▷ each row
10:             if $k \neq 0$ then
11:                 ▷ Receive data from previous block
12:             end if
13:             #pragma unroll
14:             for $j \leqslant BLOCK\_WIDTH$ do
15:                 ▷ Calculate current cell value
16:             end for
17:             if $k \neq numBlocks - 1$ then
18:                 ▷ Send data to next block
19:             end if
20:         end for
21:     end for
22: end for
23: }
```

overflow detection on the host, saturated addition is used on the FPGA kernel (in particular, the *add_sat* function).

4.3.3 Host-side buffers and data transfers. Host-side buffers are allocated to be 64-byte aligned. This fact improves data transfer efficiency because direct memory access takes place to and from the FPGA. Common data to

Algorithm 3. Host pseudo-code for multi-FPGA implementation.

```
1:▷ $n_F$ is the number of FPGAs
2:
3:  $S = multi\text{-}FPGAsearch(Q, vD, SM, t_h, n_F)$ ▷ Compute alignments in FPGAs
4:  $S = recompute\_if\_overflow(S, Q, vD, SM, t_h)$ ▷ Recompute alignments that overflowed
5:  $S = sort(S, t_h)$ ▷ Sort all scores in descending order
6:
7:  function $multi\text{-}FPGAsearch$ ($Q, vD, SM, S, t_h, n_F$)
8:
9:  for $d \leqslant n_F$ do
10:     clCreateBuffer's(...) ▷ Create buffers + transfer data
11: end for
12: for ($i = 0; i \leqslant get\_num\_chunks(vD); i += n_F$) do
13:     for $d \leqslant n_F$ do
14:         $c = i + d$
15:         $SP_c = build\_SPs(vD_c, SM, t_h)$
16:         clEnqueueWriteBuffer($SP_c$) ▷ Score Profiles to device
17:         clEnqueueWriteBuffer($n_c$) ▷ Sequence lengths to device
18:     end for
19:     $wait()$ ▷ Block until previous transferences finish
20:     for $d \leqslant n_F$ do
21:         $c = i + d$
22:         for $q \leqslant get\_num\_sequences(Q)$ do
23:             clEnqueueNDRangeKernel (...) ▷ Compute alignments among query $q$ and chunk $c$
24:         end for
25:     end for
26:     $wait()$ ▷ Block until previous kernels finish
27:     for $d \leqslant n_F$ do
28:         $c = i + d$
29:         clEnqueueReadBuffer($S_c$)
30:     end for
31: end for
32: return S
33: end function
```

all alignments, such as the queries, are transferred when creating the device buffers.

4.4 Heterogeneous multi-FPGA implementation

A simple strategy for employing several FPGAs at the same time involves exploiting thread level parallelism on the host. Following this approach, an OpenMP thread is generated for each accelerator and the database chunks are distributed among the threads as soon as they become idle, using a *parallel for* directive. Unfortunately, this strategy is not practical because the Altera OpenCL library is not thread-safe at host level (Altera, 2014). To avoid this limitation and to allow simultaneous FPGA execution, host-device data transfers are called in a non-blocking way. Because kernels cannot be invoked before data transfers are completed, the *clFinish* function is used to synchronize the host

Algorithm 4. Host pseudo-code for hybrid heterogeneous implementation.

```
1:  ▷ $p$ is the database percentage used to compute $R_F$
2:  ▷ $R_F$ is the number of database residues assigned to FPGAs
3:  ▷ $vD_p$ is the preprocessed database chunk used to estimate relative compute power
4:  ▷ $vD_h$ is the host part of the preprocessed database
5:  ▷ $vD_F$ is the accelerators part of the preprocessed database
6:
7:  $vD_p = extract(vD, p)$ Extract database chunk to estimate relative compute power
8:
9:  $[R_F, S] = estimate\_compute\_power\ (Q, vD_p, SM, t_h)$ ▷ Calculate $R_F$ calling hybrid_search
10:
11: $[vD_h, vD_F] = split(vD, p, R_F, n_F)$ ▷ Split preprocessed database
12:
13: $S = hybrid\_search\ (Q, vD_h, vD_F, SM, t_h, n_F)$ ▷ Compute alignment in host and FPGA(s)
14:
15: $S = sort(S, t_h)$ ▷ Sort all scores in descending order
16:
17: function hybrid_search $(Q, vD_h, vD_F, SM, t_h, n_F)$
18: #pragma omp parallel num_threads(2)
19: {
20:     #pragma omp single nowait
21:     { $S_F = multi\text{-}FPGAsearch(Q, vD_F, SM, 1, n_F)$ } ▷ Compute alignments in FPGA(s)
22:     #pragma omp single
23:     { $S_h = SWIMMsearch(Q, vD_h, SM, t_h)$ } ▷ Compute alignments in host
24: }
25: $S = recompute\_if\_overflow(S_F, Q, vD_F, SM, t_h)$ ▷ Recompute alignments that overflowed
26: return S
27: end function
```

and devices. Algorithm 3 shows the pseudo-code for the host implementation. The kernel code remains invariant, as in the single-FPGA implementation.

4.5 Heterogeneous hybrid implementation

By exploiting thread level parallelism, we can take advantage of CPU and FPGA computations. Algorithm 4 shows the pseudo-code for the host implementation. The hybrid implementation is based on a nested parallel scheme: initially two threads are requested. The threads invoke one routine each. The *SWIMMsearch* routine creates a nested parallel region. For the CPU alignments, our code is based on the SWIMM tool once again, which is able to take advantage of multithreading and both SSE and AVX2 extensions. The *multi-FPGAsearch* routine computes the alignments as described in the multi-FPGA implementation of Algorithm 3.

4.5.1 Workload distribution strategy. A key to achieving a high level of performance is the workload balance between the host and the accelerators. Static techniques can lead to an almost perfect distribution. However, these techniques involve knowing some information in advance, such as hardware features related to computational capabilities, and memory hierarchy. In contrast, dynamic approaches do not need any previous information, at the expense of certain performance penalization due to imbalances or idling. To solve this issue, a semi-dynamic technique is performed that takes advantage of both approaches: initial tester workload (some pairwise alignments) is used to estimate the performance of any device.

In fact, the query sequences and a configurable percentage of the database residues (p) are performed, then a scheduler estimates the relative computational capabilities of both the host and the accelerators. The number of database residues assigned to the FPGAs (R_F) is evaluated as

$$R_F = |D| \times \frac{n_F \times GCUPS_F}{n_F \times GCUPS_F + GCUPS_h} \quad (4)$$

where $|D|$ is the total number of residues in the database, n_F is the number of FPGA devices and $GCUPS_F$ and $GCUPS_h$ correspond to the $GCUPS$ performance achieved by the FPGAs and the host, respectively. t_h threads are employed to estimate the compute power of the host while, in the FPGA case, a single accelerator is used. We assume that in a multi-FPGA system each FPGA has the same features. In an environment with different FPGAs, the scheduler should assess each FPGA's capabilities, to distribute the workload as homogeneously as possible.

5 Experimental results

5.1 Experimental environment and tests carried out

All tests were performed on two heterogeneous architectures running CentOS (release 6.5). The first architecture consists of two Intel Xeon CPU E5-2670 eight-core 2.60 GHz CPUs (hyper-threading enabled) and 32 GB main memory while the second has two Intel Xeon E5-2695 version 3 14-core 2.30 GHz CPUs (hyper-threading enabled) and 64 GB main memory. Both architectures are equipped with:

- Two Altera Stratix V GSD5 half-length PCIe boards with dual DDR3 (two banks of 4 GByte DDR3);
- A single NVIDIA Tesla K20c GPU (2496 CUDA cores) with 5 GB dedicated memory and Compute Capability 3.5;

- A single 57-core Xeon Phi 3120P coprocessor card (4 hw thread per core, 228 hw threads overall) with 6 GB dedicated memory.

We used Intel's ICC compiler (version 15.0.2) with the -*O3* optimization level by default. The synthesis tool used is Quartus II DKE version 12.0 2 with OpenCL SDK version 14.0. OpenMP threads were bound to processor threads using *scatter* affinity.

We evaluated our application by searching 20 query protein sequences against two well-known databases: Swiss-Prot (release 2013_11, http://web.expasy.org/docs/swiss-prot_guideline.html) and Environmental NR (release 2014_11, ftp://ftp.ncbi.nih.gov/blast/db/FASTA/env_nr.gz). The Swiss-Prot database comprises 192,480,382 amino acid residues in 541,561 sequences, 35,213 being the maximum length in amino acids. The Environmental NR database consists of 12,91,019,045 amino acid residues in 6,552,667 sequences with the longest containing 7557 amino acids. The queries ranged in length from 144 to 5478 amino acids, and were extracted from the Swiss-Prot database (accession numbers: P02232, P05013, P14942, P07327, P01008, P03435, P42357, P21177, Q38941, P27895, P07756, P04775, P19096, P28167, P0C6B8, P20930, P08519, Q7TMA5, P33450, and Q9UKN1). Moreover, BLOSUM62 was selected as the scoring matrix, and gap insertion and extension penalties were set at 10 and 2, respectively. Each particular test was run ten times; performance was calculated by the average of ten executions to avoid variability.

Since this paper considers energy consumption as well as performance, we describe the measurement environment used on hosts and accelerators.

1. *Host*. Intel processors provide monitoring capabilities via hardware counters, but it is not obvious how to determine power consumption in this way. To solve this issue, Intel has developed the Intel Performance Counter Monitor (PCM, http://www.intel.com/software/pcm) to take power measurements on the Intel Xeon processor. The Intel Performance Counter Monitor interface allows any programmer to analyze CPU resource consumption by means of hardware counters in an easy way.
2. *FPGA*. The FPGA is monitored by means of Furaxa's PCI-Express extender connected to a data acquisition device. The PCI-Express extender reports the current supplied in both 12 V and 3.3 V PCIe power supply lines. In particular, we use Furaxa's PCIeEXT16HOT model and the current is measured with a USB data acquisition device connected to an external computer. This ad-hoc environment allows FPGA power consumption monitoring at a suitable sampling frequency for our experiment.
3. *GPU*. Modern NVIDIA GPUs have on-board sensors to query power consumption at runtime. This information can be obtained using the NVIDIA System Management Interface (*nvidia-smi*, https://developer.nvidia.com/nvidia-system-management-interface) utility, which is based on the NVIDIA Management Library and intended to help in the management and monitoring of NVIDIA GPU devices.
4. *Xeon Phi*. In a similar way to the NVIDIA Management Library for NVIDIA GPUs, Intel provides power consumption information via the Intel System Management Controller tool (Reinders and Jeffers, 2014). The coprocessor features a microcontroller located on the circuit board, which monitors incoming DC power and thermal sensors. In this context, a software-based power analyzer developed by Intel makes it easy to obtain coprocessor power by means of the *micsmc* utility. Moreover, Igual et al. (2014) also conclude that the measurements taken using the Intel System Management Controller are completely reliable, with less than 1% deviation from directly measured consumption through Xeon Phi's PCI-e channel power.

We would like to point out that the experiments of the single and multi-FPGA implementations were carried out on a system based on Xeon E5-2695 version 3 processors using 28 OpenMP threads. The other experiments include both architectures.

With regard to databases, the experiments with the single and multi-FPGA implementations were carried out using Swiss-Prot. However, owing to its limited size, Environmental NR was used to complement the experiments in the multi-FPGA implementation and to carry out a performance comparison between the hybrid CPU–FPGA version and other Smith–Waterman implementations. This database was also used to analyze performance and power trade-off. Finally, the percentage of the database used as tester to evaluate performance capabilities on host and accelerators was fixed to 1% on the system based on Intel Xeon E5-2670 and to 2% on the system based on Intel Xeon E5-2695 version 3.

5.2 Performance results

The number of cell updates per second (CUPS) is a commonly used performance measure in the Smith–Waterman context, because it allows removal of the dependency on the query sequences and the databases utilized for the different tests. A CUPS value represents the time for a complete computation of one cell in matrix H, including all memory operations and the corresponding computation of the values in the E and F

Table 2. Performance and resource usage comparison for OpenCL kernels with different integer data types.

Kernel	Performance (GCUPS)	Resource usage				Performance increase	Resource usage decrease
		ALMs	Regs	RAM	DSPs		
int16	17.6	76%	39%	75%	1%	—	—
short16	22.7	54%	28%	48%	1%	1.29×	0–0.36×
char16	27.0	41%	24%	41%	1%	1.53×	0–0.46×

Table 3. Performance and resource usage comparison for OpenCL kernels with different SIMD widths.

Kernel	Performance (GCUPS)	Resource usage				Performance increase	Resource usage increase
		ALMs	Regs	RAM	DSPs		
scalar	4.0	28%	16%	28%	1%	—	—
char4	14.2	34%	18%	31%	1%	3.56×	0–1.21×
char8	27.7	43%	22%	38%	1%	6.95×	0–1.54×
char16	47.5	60%	30%	70%	1%	11.9×	0–2.5×

arrays. Given a query sequence Q and a database D, the GCUPS (billion cell updates per second) value is calculated as

$$\frac{|Q| \times |D|}{t \times 10^9} \tag{5}$$

where $|Q|$ is the total number of residues in the query sequence, $|D|$ is the total number of residues in the database and t is the runtime in seconds (Liu et al., 2009). In this work, the runtime t includes the device buffer creation, the transfer time of host data to FPGA, the calculation time of the Smith–Waterman alignments, and the transfer-back time of the scores.

5.2.1 Performance results of the single-FPGA implementation. To evaluate FPGA performance rates, we considered different kernel implementations according to data-parallelism degree and memory hierarchy exploitation. The main differences are:

1. The *scalar* version is the baseline code where non-optimization is performed.
2. SIMD versions employ different integer data types and exploit data-level parallelism by enabling vectorization. Vectorial nomenclature refers to SIMD width; i.e. *int4* means small vectors of 4 elements, while *int8* and *int16* use 8 and 16 integer packages, respectively. The name prefix denotes the integer data type used; i.e. *int, short* and *char* represent 8, 16 and 32 bit integer data types, respectively.
3. Regarding memory exploitation, *constant Q* and *private Q* versions refer to the use of read-only constant memory and private memory to place query sequences, respectively.

Table 2 presents FPGA resource utilization and performance achieved for OpenCL kernels with different integer data types. The same *BLOCK_WIDTH* value was used in these experiments; it was set to 8 because *int16* resource consumption did not allow a higher value. As can be observed, the best option proves to be *char16*, not only in terms of GCUPS but also considering resource usage. *char16* reports an increase of 1.53 × in performance and a reduction of 0–0.46 × in resource usage with respect to *int16*. Although *int16* does not require host recomputation, alignment scores do not need a wide-range data representation. Therefore, it is convenient to compute alignments using 8 bit integers on the FPGA and recompute them on the host using wider integer types when overflow occurs.

Table 3 shows FPGA resource utilization and performance achieved for OpenCL kernels with different SIMD width. Unlike the experiments of Table 2, the *BLOCK_WIDTH* constant could be set to 16, owing to a smaller resource usage from these kernels. Without using vectorization (denoted *scalar*), our implementation performs poorly. The exploitation of data-level parallelism by enabling vectorization allows significant performance improvements. The highest GCUPS are obtained by the *char16* version, which reports a speed-up with respect to *scalar* of 11.9 × at the cost of 0–2.5 × increase in resource usage.

The *BLOCK_WIDTH* constant determines the number of vertical blocks in the alignment matrices. Table 4 exhibits FPGA resource utilization and performance achieved for *char16* kernel with different block widths. A larger *BLOCK_WIDTH* means better performance and higher resource consumption, although the performance gain falls as *BLOCK_WIDTH* increases. Since sequence lengths must be a multiple of the

Table 4. Performance and resource usage comparison for *char16* kernel with different block widths.

BLOCK_WIDTH	Performance (GCUPS)	Resource usage				Performance increase	Resource usage increase
		ALMs	Regs	RAM	DSPs		
4	11.7	40%	21%	36%	1%	—	—
8	27.0	41%	24%	41%	1%	2.31×	0–1.14×
12	37.9	53%	29%	46%	1%	3.24×	0–1.38×
16	47.5	60%	30%	70%	1%	4.06×	0–1.94×
20	52.5	75%	39%	74%	1%	4.49×	0–2.06×
24	55.0	82%	41%	81%	1%	4.7×	0–2.25×
28	57.3	92%	42%	88%	1%	4.9×	0–2.44×

Table 5. Performance and resource usage comparison for *char16* kernel with different memory exploitations.

Kernel	Performance (GCUPS)	Resource usage			
		ALMs	Regs	RAM	DSPs
char16	57.3	92%	42%	88%	1%
char16 + constant *Q*	57.2	92%	41%	88%	1%
char16 + private *Q*	58.0	92%	42%	89%	1%

BLOCK_WIDTH constant to permit successful parallel pipeline execution, larger values imply longer sequences and, as a consequence, the overhead in alignment computing increases. The best performance achieved is 57.3 GCUPS.

The effect of using constant and private memories to place query sequences was also evaluated. Table 5 shows FPGA resource utilization and the performance achieved for the kernels used in this experiment. Copying query sequences to constant memory (*char16 + constant Q*) slightly reduces performance, contrary to the expected behaviour. Constant memory is optimized for high cache hit performance. Query residues are used to index the corresponding *SP* and one residue is accessed for each row of a processed vertical block. Because global memory incorporates extra hardware to improve long memory latencies, better performance can be obtained if query sequences are transferred directly to this memory. However, private memory usage for query sequences effectively delivers a minor performance improvement with an insignificant increase in resource consumption, as can be seen in the *char16 + private Q* implementation.

We also evaluated the impact of the query length; Figure 3 illustrates the performance of different kernel implementations with varying query lengths. It can be seen that the *scalar* kernel hardly improves performance while vectorized kernels benefit from larger workloads. Lastly, the *char16 + private Q* version outperforms all other kernel implementations, attaining up to 52.9 GCUPS.

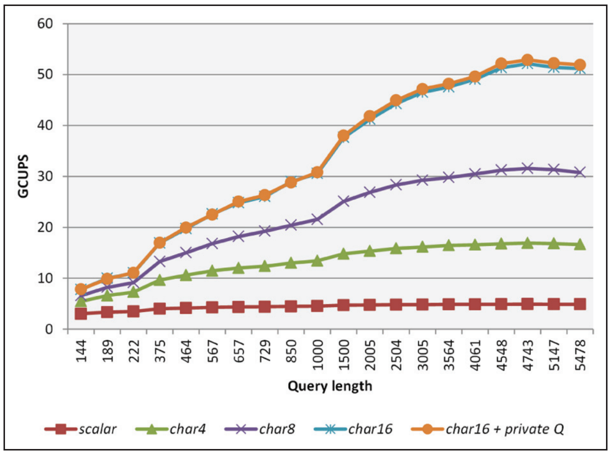

Figure 3. Performance of different OpenCL kernel implementations with queries of varying length.

5.2.2 Performance results of the multi-FPGA implementation. Table 6 shows the performance of the multi-FPGA implementation on varying the number of accelerators for the two databases selected. As can be seen, this implementation benefits from larger workloads. It is also possible to scale its performance with good workload balance when using more than one accelerator. Owing to the limited size of the Swiss-Prot database, the multi-FPGA implementation achieves a speed-up of 1.85× when using two accelerators. However, the speed-up increases to 1.96× with the larger Environmental NR database.

5.2.3 Performance results of the hybrid implementation. We compared our hybrid version with other Smith–Waterman implementations on the two heterogeneous architectures used. SWIMM (version 1.0.3) was

Table 6. Performance of multi-FPGA implementation.

Database	FPGAs	
	1	2
Swiss-Prot	58.0	107.1
Environmental NR	58.4	114.7

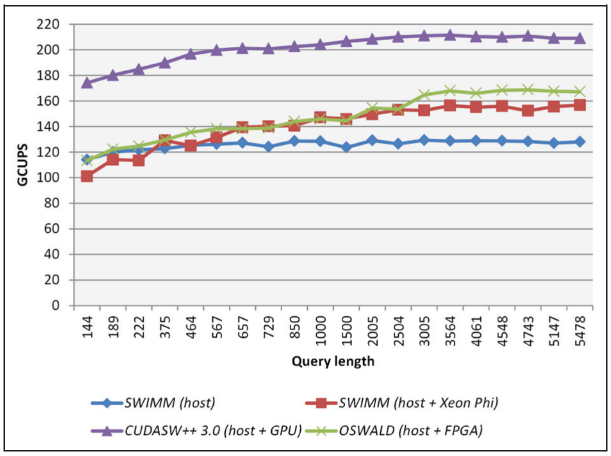

Figure 4. Performance comparison between Smith–Waterman implementations in system based on Intel Xeon E5-2670.

selected for pure Xeon and hybrid Xeon–Xeon Phi computing (Rucci et al., 2015b). SWIMM accelerates similarity searches by exploiting multithreading and takes advantage of SSE and AVX2 extensions on the host and KNC instructions on the coprocessor. Regarding hybrid CPU–GPU computing, the fastest Smith–Waterman implementation on CUDA-based GPUs, CUDASW++ 3.0 (version 3.1), was chosen (Liu et al., 2013). This implementation processes database sequences of short and medium length on the GPU device while long sequences are processed on the host using the SSE instruction set, as in the SWIPE approach (Rognes, 2011).

Figure 4 shows the performances achieved on the heterogeneous system based on Intel Xeon E5-2670 processors. As can be observed, pure SWIMM presents an almost flat curve because of multithreading and intertask parallelism exploitation through SSE extensions. The addition of Xeon Phi allows SWIMM to improve its performance, except for shorter queries, where this implementation is unable to take advantage of all the compute power available. The absence of low-range integers in the KNC instruction set of the Xeon Phi coprocessor is the cause of the small performance improvement. The peak performances are 129.3 and 156.7 for pure and hybrid versions, respectively. However, OSWALD's performance is always better than that of SWIMM (including for short queries), and

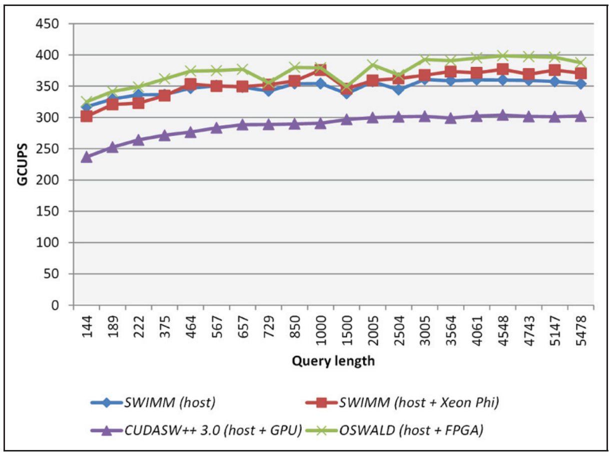

Figure 5. Performance comparison between Smith–Waterman implementations in system based on Intel Xeon E5-2695 version 3.

the difference increases as the query length increases. Thanks to a balanced workload distribution, OSWALD achieves up to 168.3 GCUPS. Lastly, CUDASW++ 3.0 outperforms all other implementations, achieving an impressive 210 GCUPS, principally due to NVIDIA's K20c computational power.

Figure 5 shows the performance achieved on the heterogeneous system based on Intel Xeon E5-2695 version 3 processors. Unlike the other heterogeneous architecture, this system features more hardware threads and the AVX2 instruction set, which permits higher data-level parallelism. The behaviour of SWIMM is similar to the previous case. Pure SWIMM achieves a nearly flat curve, achieving 360 GCUPS. Xeon Phi incorporation decreases performance for short queries and provides little additional GCUPS for the rest. In contrast with the results obtained with the previous system, CUDASW++ 3.0 presents the slowest performance. This is because CUDASW++ 3.0 exploits the SSE2 extension on the host and cannot take advantage of its more powerful AVX2 counterpart. Finally, OSWALD achieves the best performance ratios, with close to 400 GCUPS at peak. We would like to point out that the AVX2 exploitation and the well-balanced workload are key aspects in OSWALD's performance.

5.3 Performance and power consumption comparison

Finally, Table 7 presents a summary of the average performance and consumption achieved on the different architectures under study. It can be seen that FPGA computing is the worst approach from a performance perspective. However, it can be a good choice from the power point of view; its low power consumption (thermal design power <25 W) can be useful in

Table 7. Performance and power consumption summary.

System	Compute units	Cores	GCUPS	Power (watts)	GCUPS/watt
Based on Intel Xeon E5-2695 version 3	Host	28	309.3	228.2	1.355
	Host	56	354.8	240	1.478
	FPGA*	1	53.5	69	0.775
	FPGA*	28	58.4	83.1	0.702
	2× FPGA*	28	114.7	169.5	0.677
	Host + Xeon Phi	56 + 228	450.5	380	0.843
	Host + GPU	56 + 2496	298.8	328.2	0.910
	Host + FPGA	56	401.1	265.6	1.510
	Host + 2 × FPGA	56	441.6	291.2	1.516
Based on Intel Xeon E5-2670	Host	16	110.1	209.7	0.525
	Host	32	127.5	230	0.554
	Host + Xeon Phi	32 + 228	165.5	438.5	0.377
	Host + GPU	32 + 2496	206.2	303.2	0.680
	Host + FPGA	32	178.9	253.1	0.707
	Host + 2 × FPGA	32	225.1	271	0.830

* Host takes part in overflow recomputation.

environments with power restrictions or when power is the main concern. Also, following the same purpose, it can be observed that the use of a single thread in the host is a convenient way to reduce power consumption (16%) at the cost of a smaller performance detriment (8%). In the opposite sense, the use of a heterogeneous architecture based on Xeon and Xeon Phi processors is not a good option from power perspective. The incorporation of Xeon Phi coprocessor delivers low performance gain and decreases the GCUPS/watt ratio compared with host-only computing. The inability of the Xeon Phi to take advantage of low-range integer vectors prevents it from achieving better results. Moreover, the exploitation of wider vector capabilities is a key aspect in improving the GCUPS/Watt ratio, as it is evidenced in the Xeon E5-2695 version 3 (AVX2) compared to the Xeon E5-2670 (SSE). It is also observed that the use of hyper-threading (2 hw threads per core instead of a single thread) improves the GCUPS/watt ratio in both architectures. Conversely, a CPU combined with a GPU can be a good alternative in systems that feature SSE instruction set, although AVX2 extensions are not available. In the system based on the Intel Xeon E5-2670, CUDASW++ 3.0 improves the GCUPS and GCUPS/watt ratio compared with SWIMM (the host-only version). However, it is unable to repeat this performance in the system based on Intel Xeon E5-2695 version 3 because CUDASW++ 3.0 only exploits SSE instructions. Finally, hybrid CPU–FPGA computing stands as the best option from the performance/power point of view, considering that it achieves the highest GCUPS/watt ratio in both systems. Moreover, the inclusion of an additional FPGA improves this ratio in both architectures. We would like to conclude that the use of FPGA significantly improves the GCUPS/watt ratio in both systems, highlighting an improvement by 20% in the 'less' powerful system, mainly due to a more homogeneous workload distribution between the host and FPGAs.

6 Conclusions

The Smith–Waterman algorithm is one of the most popular algorithms in sequence alignment because it performs an exact local alignment. However, owing to its high computational demands, scientists have developed several parallel implementations to reduce response time. In addition, with the emergence of heterogeneous computing it is necessary to evaluate not only computationally scalable solutions but also the energy efficiency of the system. Taking into account these considerations, this paper examines the benefits of a highly innovative technology in the form of supporting the OpenCL parallel programming model in the field of FPGAs. To the best of the our knowledge, our proposal is the first high-level programming implementation on FPGAs using OpenCL with real amino acid datasets.

The main contributions of this study can be summarized as follows.

1. Starting from a single-FPGA implementation, we have stressed FPGA resources to achieve a fast kernel implementation. In this sense, the exploitation of a low-range integer type is a key aspect to improving performance and reducing resource usage simultaneously. Data-level parallelism is also critical to achieving successful performance rates at the expense of a moderate increase in resource usage. With respect to OpenCL hierarchy memory exploitation, private memory reports considerable benefits, although constant memory must be carefully studied before use. Our most successful single-FPGA implementation achieves up to 58.4

GCUPS, a significant improvement on the Altera staff implementation (Settle, 2014).

2. We have extended the single-FPGA implementation to allow execution on multiple devices and to support concurrent host execution by means of OpenMP multithreading and SIMD computing using SSE and AVX2 extensions. Estimating the relative compute power of the host and the FPGAs to calculate Smith–Waterman alignments before dividing workload contributes to a well-balanced distribution. At the same time, this strategy allows us to generalize our approach to different hardware characteristics of FPGA-based platforms. Performance evaluations on two different heterogeneous architectures demonstrate that OSWALD is competitive with other top-performing Smith–Waterman implementations, with up to 442 GCUPS at peak.
3. Finally, evaluated the performance of the different implementations from an energy point of view, considering power consumption and GCUPS/watt ratio. FPGA computing (without host concurrent execution) can be a good choice when power is the top priority. In the opposite sense, heterogeneous systems based on Xeon Phi coprocessors are not a good option for Smith–Waterman protein searches. The absence of low-range integer vectors on this coprocessor is the cause of its poor energy efficiency. Furthermore, taking advantage of wider vector capabilities is critical to improving the GCUPS/watt ratio, as indicated by the Xeon E5-2695 version 3 (AVX2) compared with the Xeon E5-2670 (SSE). However, GPU-based systems can lead to higher GCUPS (especially on those without AVX2 support), with acceptable GCUPS/watt ratios. Based on our experiments, hybrid CPU–FPGA computing stands out as the best option from a performance-to-power perspective, since it achieves the highest GCUPS/watt ratio on both systems. Furthermore, the inclusion of an additional FPGA improves this ratio in the two architectures used.

The programming cost and the lack of portability of FPGA code have traditionally limited its applicability for Smith–Waterman alignments. OSWALD is a portable, completely functional and general implementation for accelerating similarity searches on FPGA-based architectures. We expect OSWALD to become an established option for accelerating Smith–Waterman searches in an energy-efficient way.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: Enzo Rucci holds a Ph.D. CONICET fellowship from the Argentinian Government. This work has been supported by the EU (FEDER) and the Spanish MINECO, under grants TIN 2015-65277-R and TIN2012-32180.

References

Altera (2014) *Altera SDK for OpenCL Programming Guide*, version 14.0. San Jose, CA: Altera Corporation.

Altschul SF, Gish W, Miller W, et al. (1990) Basic local alignment search tool. *Journal of Molecular Biology* 215(3): 403–410.

Dydel S and Bala P (2004) Large scale protein sequence alignment using FPGA reprogrammable logic devices. In: Becker J, Platzner M and Vernalde S (eds) *Field-Programmable Logic and Application. Lecture Notes in Computer Science*, vol. 3203. Berlin: Springer, pp.23–32.

Farrar M (2007) Striped Smith–Waterman speeds database searches six time over other SIMD implementations. *Bioinformatics* 23(2): 156–161.

Farrar MS (2008) *Optimizing Smith–Waterman for the cell broad-band engine*, http://cudasw.sourceforge.net/sw-cellbe.pdf.

Gotoh O (1982) An improved algorithm for matching biological sequences. *Journal of Molecular Biology* 162: 705–708.

Igual FD, Jara LM, Gómez-Pérez JI, et al. (2014) A power measurement environment for PCIe accelerators. *Computer Science – Research and Development* 30(2): 115–124.

Isa M, Benkrid K, Clayton T, et al. (2011) An FPGA-based parameterised and scalable optimal solutions for pairwise biological sequence analysis. In: *2011 NASA/ESA conference on adaptive hardware and systems*, San Diego, CA, 6–9 June 2011, pp.344–351. New York: IEEE.

Howes L and Munshi A (2014) *The OpenCL Specification*, version 2.0. Beaverton, OR: Khronos Group.

Li TI, Shum W and Truong K (2007) 160-fold acceleration of the Smith–Waterman algorithm using a field programmable gate array (FPGA). *BMC Bioinformatics* 8: I85.

Lipman D and Pearson W (1985) Rapid and sensitive protein similarity searches. *Science* 227: 1435–1441.

Liu Y and Schmidt B (2014) SWAPHI: Smith–Waterman protein database search on Xeon Phi coprocessors. In: *IEEE 25th international conference on application-specific systems, architectures and processors, ASAP 2014*, Zurich, Switzerland, 18–20 June 2014, pp.184–185. New York: IEEE.

Liu Y, Maskell DL and Schmidt B (2009) CUDASW++ : optimizing Smith–Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes* 2(1): 1–10.

Liu Y, Schmidt B and Maskell DL (2010) CUDASW++ 2.0: enhanced Smith–Waterman protein database search on CUDS-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes* 3(1): 1–12.

Liu Y, Tran TT, Lauenroth F, et al. (2014) SWAPHI-LS: Smith–Waterman algorithm on Xeon Phi coprocessors for long DNA sequences. In: *2014 IEEE international conference on cluster computing, CLUSTER 2014*, Madrid, Spain, 22–26 September 2014, pp.257–265. New York: IEEE.

Liu Y, Wirawan A and Schmidt B (2013) CUDASW++ 3.0: accelerating Smith–Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics* 14(1): 1–10.

Qiu J, Ekanayake J, Gunarathne T, et al. (2010) Hybrid cloud and cluster computing paradigms for life science applications. *BMC Bioinformatics* 11(Suppl 12): S3.

Reinders J and Jeffers J (2014) Power analysis on the Intel® Xeon Phi^{TM} coprocessor. In: *High Performance Parallelism Pearls, Volume 1: Multicore and Many-Core Programming Approaches*. Boston: Morgan Kaufmann, pp.xli–xliv.

Rognes T (2011) Faster Smith–Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics* 12(1): 1–11.

Rucci E, De Giusti A, Naiouf M, et al. (2014) Smith–Waterman algorithm on heterogeneous systems: a case study. In: *2014 IEEE international conference on cluster computing (CLUSTER)*, Madrid, 22–26 September 2014, pp.323–330, New York: IEEE.

Rucci E, Garcia C, Botella G, et al. (2015a) Smith–Waterman protein search with OpenCL on an FPGA. In: *Trustcom/BigDataSE/ISPA, 2015 IEEE*, Helsinki, 20–22 August 2015 vol. 3. pp.208–213.

Rucci E, García C, Botella G., et al. (2015b) An energy-aware performance analysis of SWIMM: *S*mith–*W*aterman implementation on *I*ntel's *M*ulticore and *M*anycore architectures. *Concurrency and Computation: Practice and Experience* 27(18): 5517–5537.

Settle SO (2014) High-performance dynamic programming on FPGAS with OpenCL. In: *IEEE High Performance Extreme Computing Conference*, Waltham, MA, 10–12 September 2013, pp. 1–6. New York: IEEE.

Smith TF and Waterman MS (1981) Identification of common molecular subsequences. *Journal of Molecular Biology* 147(1): 195–197.

Weaver N, Markovskiy Y, Patel Y, et al. (2003) Post-placement C-slow retiming for the xilinx virtex FPGA. In: *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on field programmable gate arrays*, Monterey, CA, 23–25 February 2003, pp.185–194. New York, NY, USA: ACM.

Yamaguchi Y, Tsoi H and Luk W (2011) FPGA-based Smith–Waterman algorithm: analysis and novel design. In: Koch A, Krishnamurthy R, McAllister J, et al. (eds.) *Reconfigurable Computing: Architectures, Tools and Applications, Lecture Notes in Computer Science*, volume 6578. Berlin: Springer, pp.181–192.

Author biographies

Enzo Rucci received a B.Sc. in computer science in 2011 and a Ph.D. in computer science in 2016, both from the National University of La Plata, Argentina. He is currently a postdoctoral fellow at CONICET (Argentina) and an assistant professor at the School of Computer Science of the National University of La Plata. His research interests include high-performance computing, green computing and bioinformatics.

Carlos Garcia received an M.S. in physics and a Ph.D. in computer science from the Universidad Complutense de (UCM) in 1999 and 2007, respectively. His research interests include parallel computing, computer architecture and code optimization for high-performance computing. His current research also addresses heterogeneous systems and energy-aware trade-off.

Guillermo Botella received an M.A.Sc. degree in physics in 1999, an M.A.Sc. degree in electronic engineering in 2001 and a Ph.D. degree in 2007, all from the University of Granada, Spain. He was a research fellow funded by the EU, working at the Department of Architecture and Computer Technology of the Universidad de Granada, Spain and the Vision Research Laboratory at University College London, UK, before joining the Department of Computer Architecture and Automation of Complutense University of Madrid, Spain, first as Assistant Professor and currently as Associate Professor. His current research interests include digital signal processing, image, audio and video Processing and IP protection, as well as different implementation platforms, such as FPGAs, GPGPUs, multicores and embedded systems.

Armando E De Giusti received an engineering degree in telecommunications from the National University of La Plata, Argentina, in 1974. He is currently Head Professor at the School of Computer Science of the National University of La Plata and Principal Researcher at CONICET (Argentina). His areas of interest are parallel and distributed processing, real-time systems and computer science applied to education.

Marcelo Naiouf received a B.Sc. in computer science in 1989 and a Ph.D. in computer science in 2004, both from the National University of La Plata, Argentina. He is currently Head Professor at the School of Computer Science of the National University of La Plata. His research interests include parallel and distributed processing and intelligent systems.

Manuel Prieto-Matias received a Ph.D. in computer science from the Complutense University of Madrid (UCM) in 2000. His research interests include areas of parallel computing and computer architecture. Most of his activities have focused on leveraging parallel computing platforms and complexity-effective micro-architecture design. His current research addresses emerging issues related to asymmetric processors, heterogeneous systems and energy-aware computing, with a special emphasis on the interaction between the system software and the underlying architecture. He has co-written numerous articles in journals and for international conferences in the field of parallel computing and computer architecture.