Smith-Waterman Algorithm on Heterogeneous Systems: A Case Study

Enzo Rucci
and Armando De Giusti
and Marcelo Naiouf
Instituto de Investigacion en Informatica LIDI (III-LIDI)
Universidad Nacional de La Plata
La Plata (1900), Buenos Aires, Argentina
Email: {erucci,degiusti,mnaiouf}@lidi.info.unlp.edu.ar

Guillermo Botella
and Carlos García
and Manuel Prieto-Matias
Dept. Computer Architecture,
Complutense University of Madrid, Madrid 28040, Spain
Email: {gbotella,garsanca,mpmatias}@ucm.es

***Abstract*—The well-known Smith-Waterman (SW) algorithm is a high-sensitivity method for local alignments. However, SW is expensive in terms of both execution time and memory usage, which makes it impractical in many applications. Some heuristics are possible but at the expense of loosing sensitivity. Fortunately, previous research have shown that new computing platforms such as GPUs and FPGAs are able to accelerate SW and achieve impressive speedups. In this paper we have explored SW acceleration on a heterogeneous platform equipped with an Intel Xeon Phi co-processor. Our evaluation, using the well-known Swiss-Prot database as a benchmark, has shown that a hybrid CPU-Phi heterogeneous system is able to achieve competitive performance (62.6 GCUPS), even with moderate low-level optimisations.**

Keywords*—Bioinformatics, Smith-Waterman, HPC, Intel Xeon Phi, heterogeneous computing.

I. Introduction

High throughput structural genomic and genome sequencing are delivering huge amounts of data from the structures and sequences of thousand of proteins. To keep pace with these new technologies and to be able to extract useful information and insights from these massive data, new computational tools have to be developed in the coming years, being essential the acceleration of key primitives and fundamental algorithms.

In this paper, we have focused on the acceleration of the classic Smith-Waterman (SW) algorithm without heuristics. Almost all the applications of new sequencing technologies are based on sequence alignment [1] and SW is still (or could be) a critical and basic primitive in many of those applications. In high-throughput sequencing, the SW algorithm itself, or variations of it, are often used to align sequencing reads to reference sequences. Unfortunately, identifying the optimal alignment score using SW is computationally expensive (linear space complexity and a quadratic time complexity) since it performs an exhaustive search to find the optimal local alignment between two sequences. However, it guarantees the optimal alignment, which is essential in some applications. Furthermore, SW has been also used as the basis for many subsequent heuristic algorithms that were developed over the last few years.

BLAST (Basic Local Alignment Search Tool) is a popular example of such heuristic algorithms [2], [3] that increase speed at the cost of reduced sensitivity. This algorithm keeps the position of each k-length subsequence (k-mer) of a query sequence in a hash table (k is usually 11 for a DNA sequence), with the k-mer sequence being the key, and scans the reference database sequences looking for k-mer identical matches, which are the so-called seeds. Once those seeds have been identified, BLAST performs seed extensions and joins (first without gaps), and then it refines them using again the classic SW algorithm. Over the years, BLAST has been significantly improved adding new functionalities although keeping the same seed-and-extend structure: some proposals have enhanced the seeding process, while others have improved the seed extension [1]. Overall, the point is that accelerating SW is still a priority even though sequence alignment operations are also speeded up using heuristic tools.

Fortunately, the alignment process exhibits inherent parallelism that can be exploited to mitigate the high cost of SW. Two well-known tools that take advantage of such parallelism for SW sequence database searches are Swipe [4] and CUDASW++ [5]. The former focuses on CPUs with multimedia extensions such Intel's SSE. CUDASW++ exploits CUDA-enabled GPUs from NVIDIA and its latest version (CUDASW++3.0) uses a hybrid implementation that takes advantage of GPUs and CPUs simultaneously. Recently, Liu and Schmidt presented SWAPHI, a tool for Intel Xeon Phi accelerators [6]. There are also other proposals for SW acceleration on grid architectures [7], cloud-based systems using MapReduce [8], an even FPGAs implementations [9], [10], [11].

Our focus in this paper is on heterogeneous Intel Xeon server equipped with Intel Xeon Phi coprocessor. Unlike previous research that have focused on extracting the most of the Xeon Phi coprocessor using low-level optimisations [6], we are interested on evaluating the potential of a moderate optimised SW code that can be easily recompiled onto different processors with SIMD extensions. This way, we are

The final authenticated version is available online at https://doi.org/10.1109/CLUSTER.2014.6968784

trading portability for performance which could facilitate the optimisation on more elaborated sequencing tools that improve SW with heuristics. Nevertheless, we are able to compete with some of those previous tools taking advantage of both Xeon and Xeon Phi processors simultaneously.

Section II introduces the basic concepts of the Smith-Waterman algorithm. Section III briefly introduces the Intel's Xeon-Phi architecture and in Section IV we describe our implementation of the SW algorithm. In Section V we discuss performance results and finally in Section VI we conclude with some ideas for future research.

II. Smith-Waterman Algorithm

Smith-Waterman (SW) is a well-known algorithm for performing *local sequence alignment* that is able to return the optimal local alignment between two sequences. It is based on a dynamic programming approach and its high sensitivity comes from exploring all the possible alignments between two sequences.

In the following paragraphs we explain how the SW algorithm find a similarity score between two sequences.

Given two sequences: $A = a_1 a_2 a_3 \dots a_M$ and $B = b_1 b_2 b_3 \dots b_N$, a matrix H of $(N+1) \times (M+1)$ is built, in such a way that the residues that form sequence A label its rows (starting with 1), and those from sequence B label its columns (starting with 1). The following steps are applied to calculate the values of H that yield the similarity score between A and B:

1) Initialise the first row (row zero) and the first column (column zero) of H with zero, as shown in Equation 1.

$$H_{i,0} = H_{0,j} = 0 \quad for\ \ 0 \le i \le M and\ \ 0 \le j \le N \tag{1}$$

2) $H_{i,j}$ measures the maximum similarity between two segments ending in a_i and b_j, respectively, $\forall i \in [1, \dots, M]$ and $\forall j \in [1, \dots, N]$. This score is computed using Equation 2:

$$H_{i,j} = max \begin{cases} 0 \\ H_{i-1,j-1} + V(a_i, b_j) \\ C_{i,j} \\ F_{i,j} \end{cases} \tag{2}$$

where,

a) $V(a_i, b_j)$ is the *substitution matrix*. This is a table that describes the probability of a residue from sequence A at position i to occur in sequence B at position j. There are different options available depending on the target problem. In most cases, $V(a_i, b_j)$ rewards with positive value when a_i and b_j are identical, and punishes with a negative value otherwise.

b) $C_{i,j}$ is the score in column j considering a gap, and is calculated with Equation 3.

$$C_{i,j} = max_{1 \le k \le i} \{H_{i-k,j} - g(k)\} \tag{3}$$

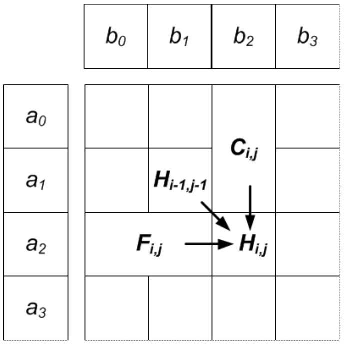

Fig. 1. Data dependences to compute H.

c) $F_{i,j}$ is the score in row i considering a gap, and is calculated with Equation 4.

$$F_{i,j} = max_{1 \le l \le i} \{H_{i,j-l} - g(l)\} \tag{4}$$

d) $g(x)$ is the penalization function for a gap of length x, and is obtained with Equation 5, q being the penalization applied for opening a gap and r the penalization for prolonging it.

$$g(x) = q + rx \quad (q \ge 0; r \ge 0) \tag{5}$$

3) Obtain the maximum similarity score as indicated in Equation 6.

$$G = max_{(0 \le i \le N; 0 \le j \le M)} \{H_{i,j}\} \tag{6}$$

4) Finally, a *backtracking* process finds the pair of segments with maximum similarity: starting from the element of matrix H where G was found, which represents the tail of the highest-scoring alignment between both sequences, until reaching a zero value position (which represent the head of the local alignment).

It is importante to note that H values can not be computed in any order due to the data dependences inherent to this problem. To be able to calculate the value of any cell, all the values of the previous cells at the same row and column have to be computed first, as shown in Figure 1. These dependences restrict the ways in that H can be computed.

III. Intel's Xeon-Phi

The adoption of accelerators within the HPC community continues to grow and it is expected that new designs from Intel, NVIDIA and AMD will likely dominate most production systems in the next few years.

The Intel Xeon Phi (Phi) is a many-core co-processor with the MIC (Many Integrated Cores) architecture that derived from the defunct Larrabee project [12]. In its current generation, the Phi features up to 61 x86 pentium cores with extended vector units (512-bit) capable of executing SIMD vector instructions and simultaneous multithreading (four hardware threads per core). Each core integrates an L1 cache (32 KB data + 32 KB instructions) and there is also a fully coherent L2 cache associated with it (512 KB combined data and instructions). A high-speed ring interconnect allows data transfer between the L2 caches and the memory subsystem.

The Phi can support up to 8 memory controllers, each one with two GDDR5 channels, and is connected to the host server through a PCIe Gen2 bus. From a software perspective, one of the strengths of these platforms is the support of existing parallel programming models used traditionally on HPC systems such as OpenMP or MPI, which simplifies code development and improves portability over other alternatives based on accelerator specific languages and other specific runtime systems.

Unlike GPUs, the Xeon Phi can be run as a completely standalone computing system, which allows running applications using exclusively the resources of the co-processor. This is called the *native mode*. Building a Xeon Phi native application usually involves minimal code modifications. In fact, many HPC codes written for the general purpose processor clusters can run in this mode without modification just recompiling them for this platform using the *mmic* compiler flag. Nevertheless, the key to Phi performance is the efficient use of the per core vector units and the small cache. In other words, just portability usually does not translate into high performance on the Phi.

The native mode can be inefficient for applications with frequent sequential parts or those with high I/O rates. In those cases, it results better to employ the Phi as a coprocessor device using the *offload mode*, which is the Phi's primary mode of operation. The programming model in this case is similar to the corresponding to other accelerators such as CUDA-enabled GPUs. The host CPU runs the sequential code of the application and invokes kernel execution on the Phi. From a software perspective, the *offload mode* is activated adding the Intel's compiler Phi specific *#pragma offload*, as show in Figure 2 for a simple axpy subroutine. This is the basic pragma to annotate those kernels (code regions) that run on the Phi. The additional *in/out* tags allow programmers to specify the required data transfers between the host and the Phi. The *#pragma omp parallel* OpenMP pragma shown in the example distributes loop iterations across Phi cores.

```
void axpy(float *y, float *x, float a
        int n)
{

  #pragma offload target(mic) \
     in(x,y:length(n)) out(y:length(n))
  {
     #pragma omp parallel for
     for(int i=0; i<n; i++)
        y[i] = a*x[i] + y[i];
  }
}
```

Fig. 2. Source code snippet that implements the axpy subroutine kernel on Intel's Xeon Phi

A programmer who is familiar with OpenMP, one of the most popular shared memory programming models within the HPC community, can find this model easier to learn than OpenCL or CUDA. This is one of the advantages over previous accelerators that Intel's marketing claims. Nevertheless, the main aspects to be addressed in order to achieve high performance are still (1) the efficient exploitation of the memory hierarchy, especially when handling large datasets, and (2) how to structure the computations to take advantage of the Phi SIMD extensions.

Ideally, programmers would only need to introduce some directives to inform the compiler about data dependencies, pointer disambiguation or data alignment, and introduce minimal code modifications to allow automatic vectorization. From the sofware point of view, this sort of guided vectorization is usually one of the best options since it allows cross-platform portability. However, in practice, programmers often need to hand-tuned their codes using language intrinsics to specify vector operations. Despite intrinsics may inhibit other loop-level optimisations that improve performance (this is unusual if codes are well tuned), hand-tuned codes usually outperform their guided counterparts. Indeed, intrinsics are the only option for complex applications with irregular access patterns or with data dependencies that can be hidden using specific code transformations. Unfortunately, most processors families have incompatible SIMD instruction sets (even from the same vendor) and the gains in performance are at the expense of developing multiple code branches that are usually difficult to maintain.

IV. SW IMPLEMENTATION

In this section we describe our mapping of the SW algorithm on both the Intel's Xeon and Intel's Xeon Phi platforms. One of our goals is to to use the same baseline code for both platforms. As mentioned above, it should be the compiler the responsible of optimising the core of the computation and vectorizing it using AVX (Advanced Vector Extensions) 256-bit extensions when targeting the Intel Xeon Processor or the MIC 512-bit extensions on the Xeon Phi.

Our application consists of four major steps:

1) Load query and database sequences.
2) Pre-process database sequences.
3) Perform SW alignments in parallel.
4) Sort all the alignment scores in descending order.

The third step performs several sequence alignments in parallel using both thread-level and simd-level parallelism. It is based on the inter-task scheme proposed by previous research [4]. Other authors have also explored fine-grained vectorization schemes [13] that are able to exploit the simd parallelism available within a single sequence alignment. However, the inter-task approach usually outperform the intra-task counterpart, especially when aligning short sequences. Essentially, when aligning several pairs in parallel, we avoid the data dependences that limit the performance of intra-task approaches [4]. Nevertheless, special care should be taken to exploit data locality as well as to avoid unbalanced execution. For instance, alignment operations take different execution time depending on the length of the sequences. A straightforward optimisation consists in pre-processing the reference database and sorting its sequences by length in advance. This way, consecutive alignments operations take similar time [14].

Algorithm 1 shows the pseudo-code of our SW baseline code. Thread level parallelism is exploited using the OpenMP programming model. The main loop that iterates over the groups of database sequences is distributed across cores with *#pragma omp parallel for* directive. The iterations are distributed according to the selected scheduling policy. The possible values are: *static*, *guided* and *dynamic*. In our observations, *dynamic* outperforms *static* significantly. The performance difference with *guided* is slightly minor. This has sense taking into account that the workload associated to each iteration is different.

At this level, the code is essentially the same for both the Intel's Xeon and the Phi, with as mention above only requires the *#pragma offload* directive and the tags that specifies the data transfers between the host and the device.

The code has also been annotated with directives that inform the compiler which loops are independent and their memory access pattern. As shown in the pseudocode, one of those directives is the new *#pragma omp simd* introduced by OpenMP4.0, which instructs the compiler to enforce vectorization of the corresponding loops.

Since data locality is the other key element to achieve, high performance especially on the Phi, we have also implemented a blocking optimisation to reduce the number of cache misses [15]. Furthermore, data structures has also been aligned to avoid the overhead of misaligned memory accesses.

Our baseline code also implements other well-know optimisations of the SW algotithm that have been proposed by previous research such as Query and Sequence Profile techniques [13], [5]. The former is based on constructing an auxiliary two-dimensional array of size $|Q| \times |E|$, where Q is the query sequence and E is the alphabet, in the pre-processing stage before performing the database search. Each row of this matrix holds the scores of the corresponding query residue against each possible residue in the alphabet. Because of each thread compares the same query residue with different database residues, this optimisation improves data locality. It increases memory requirements but it is negligible since the size of the alphabet is usually quite small compared to the size database. The sequence profile optimisation is based on constructing an auxiliary $N \times L$ sequence array , where N is the length of the database sequences and L is the number of vector lanes. Because each row of the sequence profile forms a L-lane residue vector, all its values can be gathered using a single vector load. Note that in this case, there is one array per group of reference sequences and these profiles cannot be constructed in the pre-processing stage.

Using the same baseline code for both processors has allowed us the implementation of a simple hybrid version that is able to take advantage of both the Intel Xeon and Xeon Phi coprocessor simultaneously. As shown in Figure 2, we just need to introduce an additional splitting stage that distributes the computation between both processors using a simple static distribution of the database sequences. Query distribution is also possible but it would require a different load balancing strategy.

Algorithm 1 SW($file\_database$, $file\_querysequence$, $SUBMAT$)

```
$Q = read\_file(file\_database)$                      ▷ (1)
$D = read\_file(file\_querysequence)$                 ▷ (1)

$vD = sort\_by\_length(D)$                            ▷ (2)

#pragma offload target (mic)
        in(Q,vD, SUBMAT) out(G)
{
 $G = SW\_core(Q, vD, SUBMAT)$                        ▷ (3)
}
$scores = sort(G)$▷ in descending order               ▷ (4)

function SW_CORE(Q,vD, SUBMAT)
    if $query\_profile$ then
        $QP = get\_query\_profile(Q, SUBMAT)$
    end if

    #pragma omp parallel for
    for $t \leq |Q| * |vD|$ do
        $q = get\_query\_sequence(Q, t)$
        $d = get\_database\_sequence(vD, t)$
        if $sequence\_profile$ then
            $SP = get\_sequence\_profile(SUBMAT, d)$
        end if

        for $i \leq |q|$ do
            #pragma omp simd
            for $j \leq |d|$ do
                if $query\_profile$ then
                    $V = get\_V(QP, q_i, d_i)$
                end if
                if $sequence\_profile$ then
                    $V = get\_V(SP, q_i, d_i)$
                end if
                $H_{i,j} = value(H_{i-1,j-1}, V, C_{i,j}, F_{i,j})$
            end for
        end for
        $G = get\_score(H)$▷ save similarity scores
    end for
    return $G$
end function
```

V. Experimental Results

A. Environmental Features

All tests were performed in a computer with a 2×Intel Xeon CPU E5-2670 8-core 2.60GHz CPU with hyperthreading and 32 GB installed RAM memory. The Operating System used is GNU-Linux (CentOS release 6.5). The Xeon Phi coprocessor has 60 cores supporting the execution of four hardware threads (240 hardware threads in total) and 5GB installed RAM memory. Additionally, the algorithms used in this work were compiled using Intel's ICC compiler (icc compiler version 14.0.2.144) with optimizations *-O3*. It incorporates OpenMP library for multithreading on Intel Xeon and Xeon Phi. This compiler also supports hand-tunned by means of intrinsic functions and guided vectorization (enabled with *-vec* flag compiler).

B. Tests Carried Out

This work consists on a first evaluation of heterogeneous computing approach based on Intel's Xeon and Xeon Phi systems. Initially, vector capabilities are assessed. Our studies include the evaluation of guided vectorization using pragmas

Algorithm 2 SW_het($file\_database$, $file\_querysequence$, $SUBMAT$)

```
$Q = read\_file(file\_database)$                          ▷ (1)
$D = read\_file(file\_querysequence)$                     ▷ (1)

$[vD_{CPU}, vD_{MIC}] = sort\_and\_split(D)$              ▷ (2)

#pragma offload target (mic)
    in(Q,vD$_{MIC}$, SUBMAT) out(G$_{MIC}$) signal(sem)
{
 $G_{MIC} = SW\_core(Q, vD_{MIC}, SUBMAT)$               ▷ (3)
}

 $G_{CPU} = SW\_core(Q, vD_{CPU}, SUBMAT)$               ▷ (3)

#pragma offload target(mic) wait(sem)
$scores = sort(G_{MIC}, G_{CPU})$                        ▷ (4)
```

denoted as *simd* in the experiments and vectorization based on intrinsic instructions labeled as *intrinsic*. As baseline code, we have considered a version without SIMD exploitation which is named as *no-vec*. In addition, we have considered two substitution score schemes known as query-profile and sequence-profile (*QP* and *SP* in experiments). To provide the most relevant study, the experiments are performed with the Swiss-Prot database (release 2013_11)[1]. This database comprises 192480382 amino acids in 541561 sequences with the largest sequence length equal to 35213. The 20 query protein sequences for performance evaluation were selected from the aforementioned database (accession numbers: P02232, P05013, P14942, P07327, P01008, P03435, P42357, P21177, Q38941, P27895, P07756, P04775, P19096, P28167, P0C6B8, P20930, P08519, Q7TMA5, P33450, and Q9UKN1), ranging in length from 144 to 5478. Moreover, BLOSUM62 was used as scoring matrix, and values of 10 and 2 as gap insertion and extension penalties, respectively.

C. Performance Results

This subsection evaluates the performance of the heterogeneous system. Initially, the results are shown for each platform separately and after that, the analysis is completed regarding the entire system. In order to avoid dependencies from input sequence and database, performance results are expressed in GCUPS, a widely used metric by scientific community [4], [14], [16].

1) Intel Xeon results: Figure 3 shows the performance on Intel Xeon for the different approaches under evaluation varying the thread number from 1 to 32. As it was expected, the two non-vectorized versions hardly offer performances. Vector capabilities usage improves performance being more plausible with hand-tunned vectorization (intrinsic version). The best results correspond to the *SP* scheme which reaches up-to 30.4 GCUPS with 32 threads.

SW scalability is generally successful, being close to the ideal efficiency in most cases. In particular, we observe an efficiency from 99% to 88% with 4 and 16 threads respectively in *intrinsic-SP* test (when *hyper-threading* is enabled, it's reduced to 70% for 32 threads). The efficiency for *intrinsic-QP* is slightly less (73% with 16 threads), mainly motivated by the difficulty in hand-tuned vectorization which is addressed below.

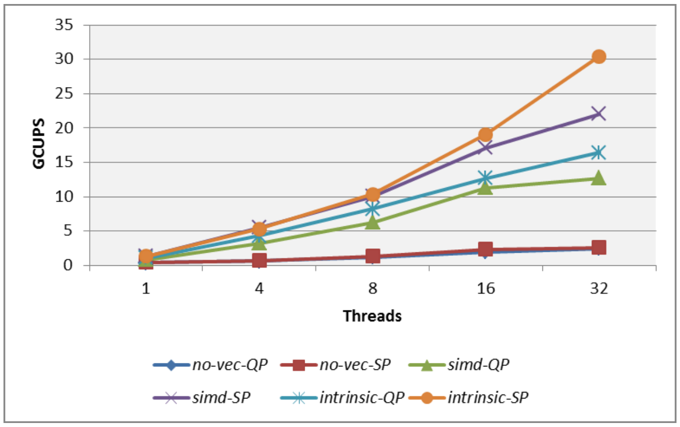

Fig. 3. Performance on Intel Xeon algorithm with different number of threads.

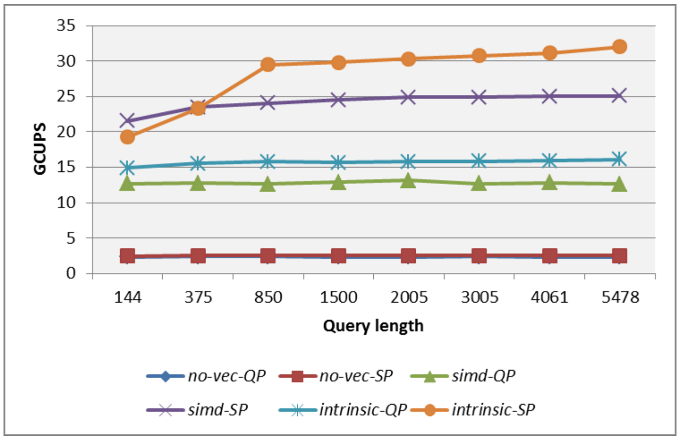

Fig. 4. Performance in Intel Xeon algorithm with a variable query length.

Figure 4 illustrates performance evolution varying the query length with the most favorable configuration of 32 threads. According to the results obtained, we notice that the query length has practically no impact on the performance in most of experiments. However, it exists a light improvement trend in sequence-profile versions (labeled with *SP* in this Figure) that reflects a 25.1 and 32 GCUPS for *simd-SP* and *intrinsic-SP* respectively.

There are considerable differences between *QP* and *SP* approaches which are more noticeable using intrinsic functions. This aspect is mainly due to non-contiguous memory accesses pattern inside substitution scheme. Despite of the substitution scores for a matching are close regarding *QP* approach, they are not necessary consecutive. Since Intel's Xeon does not incorporate vector gather functionality, the substitution scores matrix cannot be loaded into vector registers in a single operation (shuffle intrinsic instructions are needed).

2) Intel Xeon Phi results: Figure 5 shows Intel's Xeon Phi performance with different versions running from 30 to 240 threads. As in the Intel's Xeon, the non-vectorized

[1] The Swiss-Prot database available at http://web.expasy.org/docs/swiss-prot_guideline.html

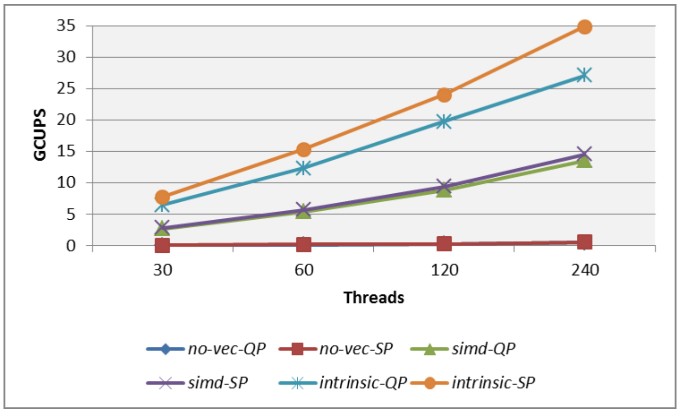

Fig. 5. Performance of the different Intel Xeon Phi algorithm variants using a variable number of threads.

versions barely exhibit performances. Both guided vectorization implementations labeled with *simd*, present similar behavior, achieving a maximum of 13.6 and 14.5 GCUPS for *QP* and *SP*, respectively. Regarding Intel Xeon case, both versions developed with intrinsic functions offer much more performance rates: 27.1 and 34.9 for *QP* and *SP* approaches. Because Intel Xeon Phi provides vector gather capabilities, non-contiguous memory accesses in query profile scheme have less influence on *intrinsic-QP* performance. Results suggest that hand-vectorization have more impact in term of GCUPS than in Intel Xeon. Regarding to multi-threading scalability, this figure shows that OpenMP implementations are scalable with the number of threads. This fact suggests that future coprocessors with more cores and threads per core will provide better GCUPS.

Figure 6 illustrates the performance on Intel's Xeon Phi algorithm varying query lengths with a configuration of 240 threads. As in previous analysis, we can confirm that as the query length is longer, there is more performance achieved since there exists more parallelism to be exploited. Besides, a synergistic effect is achieved on the exploitation of thread level parallelism with intrinsic vectorization. Respect to *QP* and *SP*, consecutive memory accesses for *SP* substitution scheme allow better performance for Xeon Phi intrinsic versions. Even so, both implementations show good scalability figures.

Because exploiting data locality is a key aspect for improving performance in parallel applications, it is interesting to study the impact of applying a block technique on each device. Figure 7 illustrates the performance evolution of blocking and non-blocking versions of the most advantageous Intel Xeon and Intel Xeon Phi tests (*intrinsic-SP* in both cases) using all available threads on each device. We can notice that exploiting data locality can seriously improve the performance on both devices. We would like to remark that this optimization has a larger improvement in the Intel's Xeon Phi because its cache size is lower than its counterpart Intel's Xeon.

3) Results in Heterogeneous system: In this section we evaluate SW behavior using both processor and coprocessor. To analyze the performance we chose *SP* version which provided better GCUPS as remarked in previous sections.

This version is based on database sequences distribution between both devices (see Algorithm 2), where the key to success is a well balanced distribution of the workload. Figure 8 shows the performance achieved in the heterogeneous system where the workload between the host and the accelerator is varied. The percentage of workload sent to Intel's Xeon-Phi is shown in abscissa, and thus the remainder workload is performed in the host.

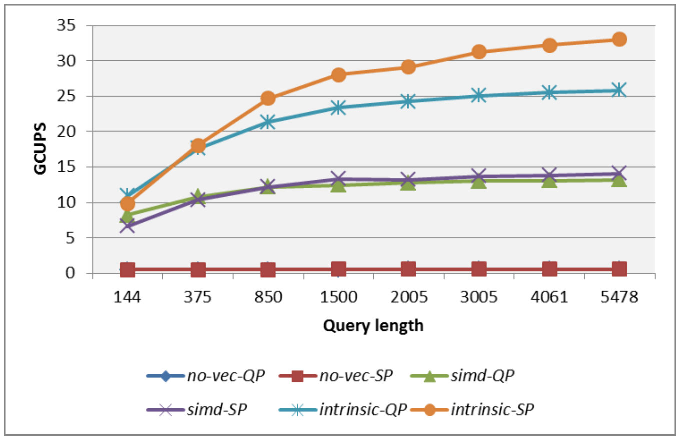

Fig. 6. Performance of the different Intel Xeon Phi algorithm variants using variable query lengths.

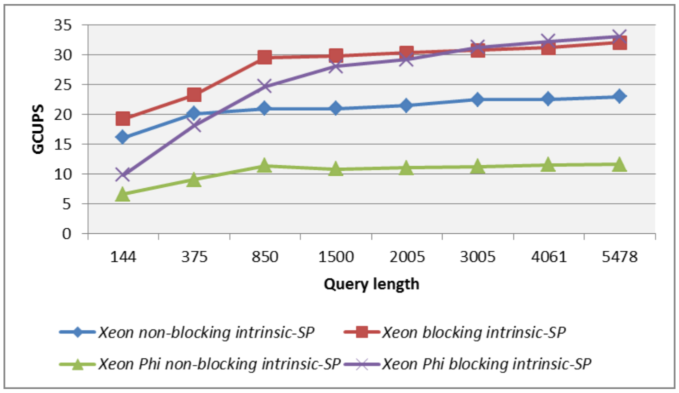

Fig. 7. Performance of blocking and non-blocking Intel Xeon and Intel Xeon Phi algorithms variants using variable query lengths.

As shown, the best configuration is close to a homogeneus distribution (45% in Xeon and 55% in Xeon-Phi). The performance achieved is almost the combination of their individual throughputs (30.4 and 34.9 GCUPS in Xeon and Xeon-Phi respectively) which is totaled to 62.6 GCUPS.

These successful results open the possibility of considering the heterogeneous computing not only from the performance point of view, but also considering other aspects such as power consumption, price cost, etc. From the point of view of power consumption we would suggest that it seems appropriate to explore others configurations with lower consumption since the TDP (thermal design power) on Intel's Xeon chip is 120 watts meanwhile the Xeon-Phi is 240 watts. Once again, there is a limb difference so that workload distribution could determinate other aspects. As future work we are considering undertaking this study.

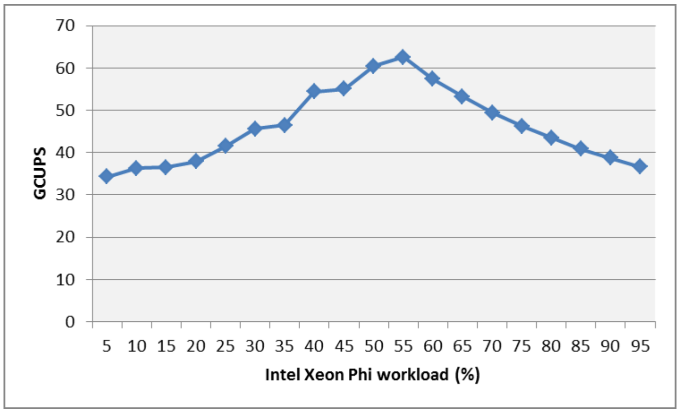

Fig. 8. Performance of the heterogeneous algorithm for different workload distributions.

VI. Conclusions

The SW algorithm is one of the methods for local sequence alignments. Nevertheless, in practice, various heuristics are used due to its computational complexity. In addition, Xeon Phi is a recent coprocessor designed by Intel specifically for high performance computing. Among its advantages, it can be found the compatibility with Intel Xeon codes and the great multi core vectorization capabilities. In this work, we have demonstrated the compute capability of current heterogeneous systems for accelerating SW database searches without losing precision in the results. Among main contribution of this research we can summarised:

- To obtain successful performance rates in this type of devices, exploiting two levels of parallelism is needed: thread-level parallelism by means of OpenMP and data-level parallelism using SIMD instructions.
- Regarding the well-known Swiss-Prot database, 32 and 34.9 GCUPS is achieved on the Intel's Xeon and the Intel Xeon Phi respectively. For heterogeneous computing, it reaches 62.6 GCUPS.
- The key to have good scalability in a heterogeneous system is to find an optimal distribution workload.

According to the results obtained, we plan to analyze other workload distribution strategies taking into account other considerations as power consumption, device prices, an so on. We are also interested in evaluating the performance of these algorithms with larger sequences databases, as UniProt-TrEMBL. This will allow us to asses the impact of transferences between host and coprocessor.

Acknowledgment

Enzo Rucci holds a PhD CONICET Fellowship under Argentinian Government. This work has been partially supported by the Spanish research project TIN 2012-32180 and the CAPAP-H4 network (TIN2011-15734-E).

References

[1] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in Bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010. [Online]. Available: http://bib.oxfordjournals.org/content/11/5/473.abstract

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool." *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, Oct. 1990. [Online]. Available: http://dx.doi.org/10.1006/jmbi.1990.9999

[3] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast: a new generation of protein database search programs." *Nucleic Acids Res*, vol. 25, no. 17, pp. 3389–3402, September 1997.

[4] T. Rognes, "Faster Smith-Waterman database searches with inter-sequence SIMD parallelization," *BMC Bioinformatics*, vol. 12:221, 2011.

[5] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions," *BMC Bioinformatics*, vol. 14:117, 2013.

[6] Y. Liu and B. Schmidt, "Swaphi: Smith-waterman protein database search on xeon phi coprocessors," in *25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2014)*, 2014.

[7] F. Chichizola, M. Naiouf, L. D. Giusti, IsmaelRodriguez, and A. D. Giusti, "Overhead Analysis in Parallel Processing DNA Sequences on Grid Architectures," in *Proceedings of the LAGrid08 (2nd International Latin American Grid Workshop 2008)*, 2008.

[8] J. Qiu, J. Ekanayake, T. Gunarathne, J. Y. Choi, S. H. Bae, H. Li, B. Zhang, T. Wu, Y. Ruan, S. Ekanayake, A. Hughes, and G. Fox, "Hybrid cloud and cluster computing paradigms for life science applications." *BMC Bioinformatics*, vol. 11 (Suppl12), 2010.

[9] Y. Yamaguchi, K. H. Tsoi, and W. Luk, in *ARC*, A. K. 0001, R. Krishnamurthy, J. McAllister, R. Woods, and T. A. El-Ghazawi, Eds. Springer, pp. 181–192.

[10] C. W. Yu, K. H. Kwong, K. H. Lee, and P. H. W. Leong, "A smith-waterman systolic cell," in *In Proceedings of the 13th International Workshop on Field Programmable Logic and Applications FPL 2003*. Springer, 2003, pp. 375–384.

[11] T. I. Li, W. Shum, and K. Truong, "60-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)," *BMC Bioinformatics*, vol. 8:I85, 2007.

[12] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *IEEE Micro*, vol. 29, no. 1, pp. 10–21, 2009.

[13] M. Farrar, "Striped Smith-Waterman speeds database searches six time over other SIMD implementations," *Bioinformatics*, vol. 23 (2), pp. 156–161, 2007.

[14] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," *BMC Research Notes*, vol. 2:73, 2009.

[15] E. Rucci, "Computacin eficiente del alineamiento de secuencias de adn sobre cluster de multicores," Master's thesis, Universidad Nacional de La Plata, Argentina, 2013.

[16] Y. Liu, W. Huang, J. Johnson, and S. Vaidya, "GPU Accelerated Smith-Waterman," *Lecture Notes in Computer Science*, vol. 3994, pp. 188–195, 2006.