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 losing 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 coprocessor. 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, whereas the latter targets CUDA-enabled GPUs from NVIDIA. The latest version of CUDASW++ (version 3.0) uses a hybrid implementation that is able to take advantage of both GPUs and CPUs simultaneously. More recently, Liu and Schmidt have presented SWAPHI, a highly optimized hand-tuned SW implementation 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 an Intel Xeon Phi coprocessor. Unlike previous research that have focused on extracting the most of the Xeon Phi coprocessor using low-level optimizations [6], we are interested on evaluating the potential of a moderate optimized SW code that can be easily recompiled onto different platforms with SIMD extensions. This way, we are trading portability for performance, which could facilitate the optimization 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.

978-1-4799-5548-0/14/$31.00 ©2014 IEEE

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 $A = a_1a_2a_3 \dots a_M$ and $B = b_1b_2b_3 \dots b_N$, an alignment 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.

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

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.

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

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

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

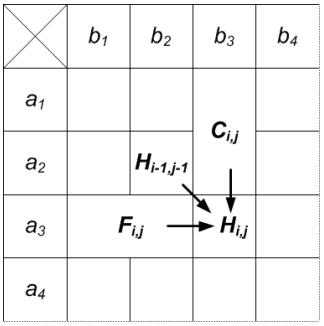

Fig. 1. Data dependences in the alignment matrix *H*.

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.

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

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

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

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 coprocessor 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) and simultaneous multithreading (four hardware threads per core). Each core integrates an L1 cache (32 KB data + 32 KB instructions) and has an associated fully coherent L2 cache (512 KB combined data and instructions). As shown in Figure 2, a high-speed ring interconnect allows data transfer between all the L2 caches of the Phi 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 this platform is the support of existing parallel programming

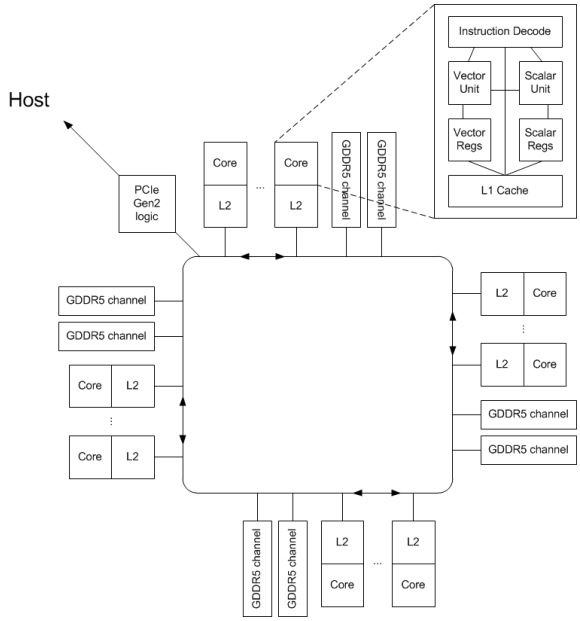

Fig. 2. Xeon Phi architecture.

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 programming languages.

Unlike GPUs, the Xeon Phi can be run as a completely standalone computing system, which allows running applications using exclusively the resources of the coprocessor. This is called the *native mode*. Building a Xeon Phi native application usually involves minimal code modifications. In fact, many HPC codes written for 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, easy portability does not automatically ensure high performance.

The native mode can be inefficient for applications with frequent sequential parts or those with high I/O rates. In those cases, it is 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 other accelerators such as CUDA-enabled GPUs. The host CPU runs the sequential code of the application and invokes kernel execution on the Phi. Figure 3 shows an example code that illustrates the basic concepts. The *offload* pragma annotates code regions that run as a kernel on the Phi. With the additional *in/out* tags, programmers specify the required data transfers between the host and the Phi memories. Finally, the OpenMP *omp parallel for* pragma instructs the compiler to distribute loop iterations across Phi threads (60 cores, 4 threads per core).

A programmer who is familiar with OpenMP will find this model easier to learn than OpenCL or CUDA. This is one of the advantages over GPU and FPGA accelerators that Intel's marketing claims. Nevertheless, the main aspects to be

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

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 vector units.

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. However in practice, *guided auto-vectorization* is not able to achieve the best performance and programmers often need to concentrate on hand-tuned their codes using language intrinsics. Despite intrinsics may inhibit other loop-level optimisations that improve performance, highly optimised 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, the gains in performance are at the expense of losing cross-platform portability. Most processors families, even from the same vendor, have incompatible intrinsics since they support different SIMD instruction sets. Therefore, programmers are forced to develop 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, compiler should take the responsibility of optimizing the core of the computation and vectorize 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 target application performs SW sequence database searches and consists of the following four major steps:

1) *Loading stage*: loading of the query and database sequences.
2) *Pre-processing stage*: preprocessing of the reference database.
3) *SW stage*: SW alignments.
4) *Sorting stage*: sorting by the alignment scores in descending order.

The SW stage 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 outperforms 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 optimization 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 *chunks* of database sequences is distributed across cores with the *#omp parallel for* pragma using a *dynamic* scheduling policy. Despite sorting the reference database, the *static* scheduling does not performance well as has been also highlighted in previous research [6]. However, we have not observed any noticeable improvement when using *guided* scheduling. Note that, 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 additional *#pragma offload* directive and the tags that specify 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 pseudo-code, one of those directives is the new *#pragma omp simd* introduced by OpenMP 4.0, 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, blocking is also necessary to reduce the number of cache misses [15]. Furthermore, data structures has also been aligned to avoid the overhead of misaligned memory accesses (64-byte aligned for the Phi and 32-byte aligned for the Xeon).

Our baseline code also implements other well-know optimizations of the SW algorithm that have been proposed by previous research such as the Query Profile (*QP*) and Sequence Profile (*SP*) optimizations [13], [5]. The former is based on constructing an auxiliary two-dimensional$|Q| \times |E|$ array, 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. Since each thread compares the same query residue with different database residues, this optimization improves data locality. It also increases memory requirements but this is negligible since the size of the alphabet is usually quite small compared to the size of the database. The sequence profile optimization 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,

Algorithm 1 SW($reference\_database$, $query\_sequences$, $SUBMAT$)

```
1:  Q = read_query_sequences()                                   ▷ (1)
2:  D = read_reference_database()                                ▷ (1)
3:
4:  vD = sort_by_length(D)                                       ▷ (2)
5:
6:  #pragma offload target (mic)
7:          in(Q,vD, SUBMAT) out(G)
8:  {
9:   G = SW_core(Q, vD, SUBMAT)                                  ▷ (3)
10: }
11: scores = sort(G)▷ in descending order                        ▷ (4)
12:
13: function SW_CORE(Q,vD, SUBMAT)
14:     if query_profile then
15:         QP = get_query_profile(Q, SUBMAT)
16:     end if
17:
18:     #pragma omp parallel for
19:     for t ≤ |Q| * |vD| do
20:         q = get_query_sequence(Q, t)
21:         d = get_database_sequence(vD, t)
22:         if sequence_profile then
23:             SP = get_sequence_profile(SUBMAT, d)
24:         end if
25:
26:         for i ≤ |q| do
27:             #pragma omp simd
28:             for j ≤ |d| do
29:                 if query_profile then
30:                     V = get_V(QP, q_i, d_j)
31:                 end if
32:                 if sequence_profile then
33:                     V = get_V(SP, q_i, d_j)
34:                 end if
35:                 H_{i,j} = value(H_{i-1,j-1}, V, C_{i,j}, F_{i,j})
36:             end for
37:         end for
38:         G = get_score(H)▷ save similarity scores
39:     end for
40:     return G
41: end function
```

there is one array per chunk 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 heterogeneous version that is able to take advantage of both the Intel Xeon and the Xeon Phi coprocessor simultaneously. Similar approaches have been investigated by previous research on heterogeneous CPU-GPU platforms [5], [16], [17]. The key to outperform the homogeneous implementations is to balance the runtimes between the Xeon Core and the Phi. This is a relatively easy task knowing in advance the relative performance of both processors, which can be found empirically. Sorting the reference database by length also simplifies it since it guarantees that consecutive alignments take similar time. With these assumptions, we can estimate statically the number of sequences assigned to the Phi to balance the runtime. As shown

in Figure 2, this distribution can be implemented introducing an additional splitting stage after sorting the database.

Algorithm 2 Heterogeneous_SW($reference\_database$, $query\_sequences$, $SUBMAT$)

1: $Q = read\_query\_sequences()$ ▷ **(1)**
2: $D = read\_reference\_database()$ ▷ **(1)**
3:
4: $[vD_{CPU}, vD_{MIC}] = sort\_and\_split(D)$ ▷ **(2)**
5:
6: #pragma offload target (mic)
7: in(Q,vD$_{MIC}$, SUBMAT) out(G$_{MIC}$) signal(sem)
8: {
9: $G_{MIC} = SW\_core(Q, vD_{MIC}, SUBMAT)$ ▷ **(3)**
10: }
11:
12: $G_{CPU} = SW\_core(Q, vD_{CPU}, SUBMAT)$ ▷ **(3)**
13:
14: #pragma offload target(mic) wait(sem)
15: $scores = sort(G_{MIC}, G_{CPU})$ ▷ **(4)**

V. Experimental Results

A. Experimental platform and performance evaluation

All tests have been performed on a Xeon server running CentOS equipped with:

- Two Intel Xeon CPU E5-2670 8-core 2.60GHz CPUs with hyper-threading enabled.
- 32 GB main memory.
- A single 60-core Xeon Phi coprocessor card (4 hardware threads per core, 240 hardware threads overall) with 5GB dedicated memory.

We have used the Intel's ICC compiler (version 14.0.2.144) with the *-O3* optimization level by default. Auto-vectorization has been enabled with the *-vec* compiler flag.

The experiments used for assessing performance are similar to previous work [4], [14], [18]. We have evaluated our application by searching 20 query protein sequences against the Swiss-Prot database (release 2013_11)[1]. This database comprises 192480382 amino acids in 541561 sequences with the longest sequence containing 35213 amino acids. The queries has also been extracted 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. BLOSUM62 has been used as scoring matrix, and gap insertion and extension penalties have been set to 10 and 2, respectively. Performance results are in GCUPS .

B. Performance results on the Intel Xeon

Figure 4 shows the performance on the Intel Xeon for the different approaches under evaluation with increasing number of OpenMP threads. Without enabling vectorization (denoted as *no-vec* in the figure), our implementation hardly improves performance. Vectorization does not only improve performance significantly, but it also allows our codes to scale with the number of threads. The hand-tuned versions based on AVX intrinsics (denoted as *intrinsic)* outperform the guided vectorization counterparts (denoted as *simd)*. Overall, the Sequence profile (denoted as *SP*) performs better than the Query profile (denoted as *QP*). *SP* achieves almost linear speedup, reaching 30.4 GCUPS with 32 OpenMP threads.

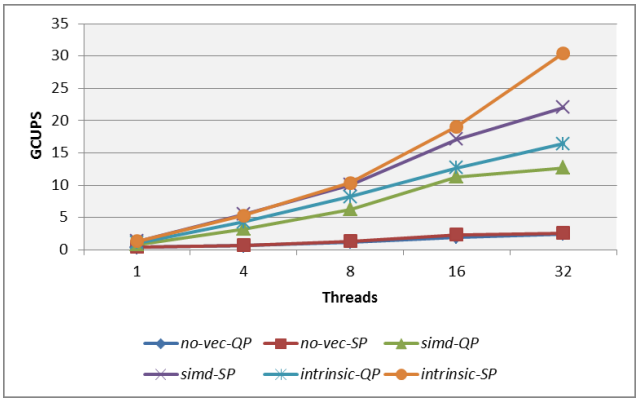

Fig. 4. Scalability on the Intel Xeon.

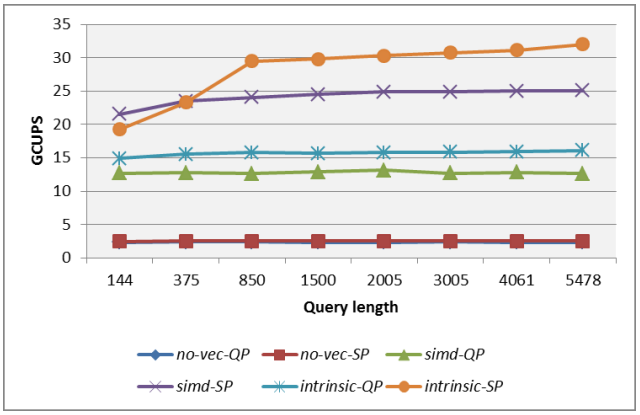

Fig. 5. Performance on the Intel Xeon with queries of varying length.

Figure 5 illustrates the performance with queries of varying length using 32 OpenMP threads. Most implementations have a rather flat performance curve. This is the expected behavior since our implementation exploits inter-task parallelism. However, we observe a gradual loss in performance for shorter queries, which is more noticeable for the hand-tuned *SP* version (performance drops from 32 GCUPS to just 19.3 GCUPS). Indeed, for query lengths shorter than 375 residues, *QP* outperforms *SP*. A similar behavior has been observed in previous research for the Xeon Phi [6]. This can be explained by the additional overhead incurred by the construction of the *SP* profile. As mention above, these profiles cannot be constructed in the pre-processing stage, and this overhead cannot be offset for shorter queries. This observation also suggests us the implementation of an adaptive version that dynamically sets either the *SP* or the *QP* profile based on the length of the queries, which should be known in advance.

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

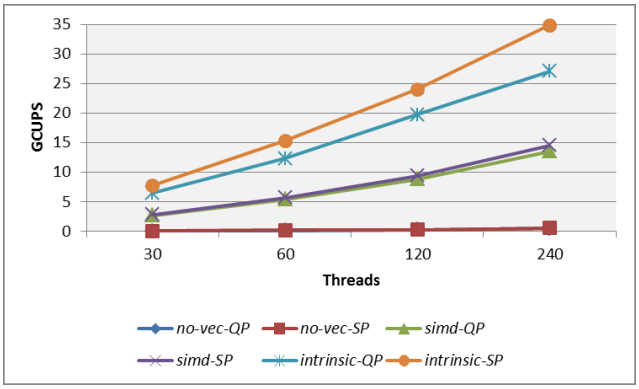

Fig. 6. Scalability on the Intel Xeon Phi.

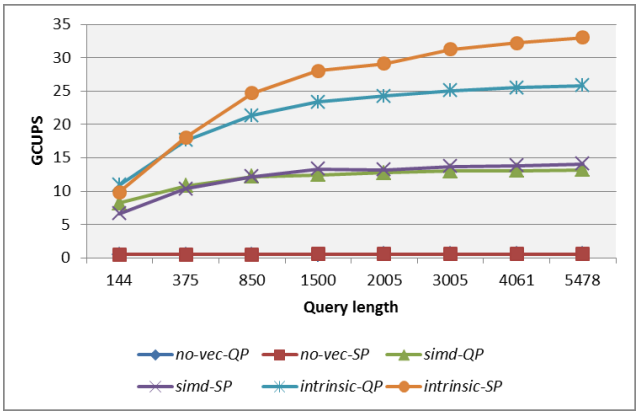

Fig. 7. Performance on the Intel Xeon Phi with queries of varying length.

C. Performance results on the Intel Xeon Phi

Figure 6 shows the performance on the Intel Xeon Phi for the different approaches under evaluation with increasing number of threads (from 30 to 240 hardware threads). Again, without enabling vectorization, our implementation hardly improves performance. The hand-tuned codes based on MIC intrinsics also outperforms guided vectorization counterparts. However in this case, the performance gap between both approaches is much higher (around 2×). Indeed, even the hand-tuned *QP* is able to outperform *SP* with automatic vectorization. We also observe that the hand-tuned *SP* achieves the maximum performance (34.9 GCUPS with 240 threads), scaling relatively well with the number of hardware threads. In contrast, the guided auto-vectorization counterpart only reaches 14.5 GCUPS.

Figure 7 illustrates the performance with queries of varying length using 240 hardware threads. There is a noticeable drop in performance with queries shorter than about 850 residues. Again, *QP* outperforms *SP* for short queries due to the additional overhead incurred by the *SP* profile construction, also suggesting an adaptive implementation.

Finally, Figure 8 analyzes the impact on performance of the blocking optimizations used in our codes with queries of varying length. In these test we have used 240 hardware threads. Blocking achieves a consistent 3× performance improvement

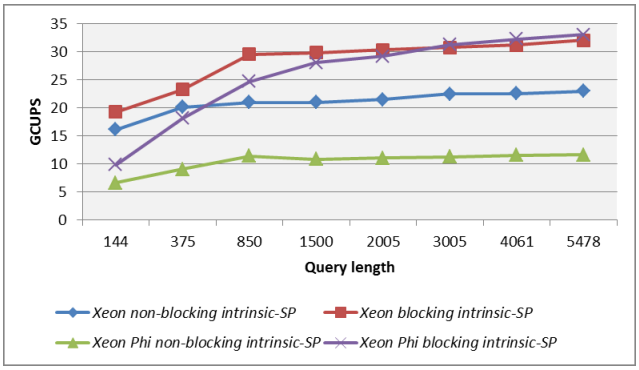

Fig. 8. Performance impact of the blocking optimizations with queries of varying length.

for query lengths greater than 850 residues, but there is also an important gain even for the shorter query lengths. As a reference, we have also shown the performance on the Intel Xeon (using 32 OpenMP). Xeon also benefits from blocking, but the improvements are lower. This is the expected behavior since the Xeon has a larger Cache than the Phi.

Another interesting insight shown in Figure 8 is that our Xeon implementation is able to outperform the Phi counterpart for shorter query lengths. However, we should highlight that there is still room for additional performance optimizations in our codes and the Phi will benefit more from them. Indeed, this is a penalty from using a common baseline code with very few low-level optimizations. However, highly hand optimized codes, such as the recently presented SWAPHI [6], are able to achieve up to 58.8 GCUPS on a single Phi card. Achieving such impressive performance, but relying on compiler technology as much as possible is still one of our goals for future research.

D. Performance results of the heterogeneous implementation

Our evaluation concludes with the analysis of the heterogeneous implementation. The key to achieve higher performance is to balance the workload between both processors. Figure 9 analyses this for best hand-tuned *SP* implementation. We are using all the computational resources and we have varied the percentage of the sequence pairs that are aligned on the Phi. The maximum performance is achieved with a static 45% Xeon 55% Xeon-Phi distribution, in other words, assigning 20% more job to the Phi. Despite using a relatively simple workload distribution scheme, the overall maximum performance is 62.6 GCUPS. This is close to the sum of the individual performances achieved by the Xeon and the Phi (30.4 and 34.9 GCUPS respectively). Therefore, the additional overhead caused by the sequence distribution of our heterogeneous implementation is almost negligible. Overall, we are able to compete with the results achieve by SWAPHI on a single Phi, but at the expense of using all the computational resources of our heterogeneous server.

VI. Conclusions and Ideas for Future Research

In this paper, we have presented an evaluation of a heterogeneous SW database search algorithm. In our implementation,

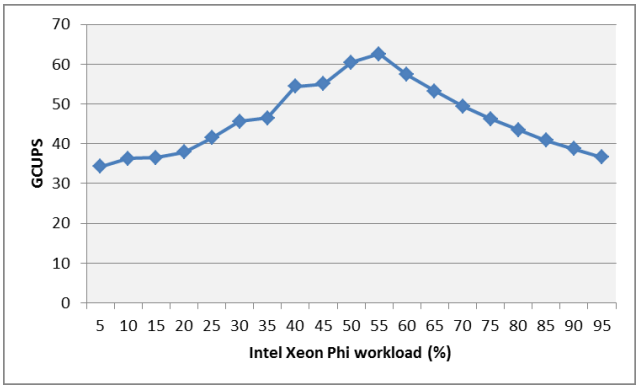

Fig. 9. Performance of the hybrid implementation varying the percentage of sequence pairs that are aligned on the Phi.

Xeon Cores and a Xeon Phi coprocessor run collaboratively for computing multiple SW alignments in parallel. A static distribution is able to balance the Xeon and Xeon Phi running times, although we need to know in advance the relative performance of both processors to apply it effectively. For our best hand-tuned implementations, assigning around 20% more work load to the Phi gets the optimal balance and achieves the maximum performance (more than 60 GCUPS using all the computational resources of our server). Nevertheless, we are also analyzing other approaches that distribute both queries and reference sequences dynamically since they allow us to have a finer control of the computational resources.

Highly hand optimized codes, such as the recently presented SWAPHI [6], are able to achieve just slightly lower performance on a single Phi. Reaching such impressive performance, but relying on compiler technology as much as possible is still one of the main targets of our future research. Our previous experience about SIMD tuning encourages us to still pursue such a goal [19].

Finally, we would like to analyze the opportunities for saving energy consumption that heterogeneous implementation allows. Energy is the product of time and power and since the heterogeneous implementation is able to reduce time, it could also save power. However, this is just intuition. Modern architectures integrate different power knobs than have a tremendous impact on power-performance ratios. Overall, we do not have yet a define answer to that question since this is not a simple analysis and it requires additional infrastructure. Fortunately, we have recently built a prototype to analyze such power-performance tradeoffs [20] and we plan to investigate this issue as part of our future research.

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, "Computación eficiente del alineamiento de secuencias de adn sobre cluster de multicores," Master's thesis, Universidad Nacional de La Plata, Argentina. Available online at: http://hdl.handle.net/10915/27737, 2013.

[16] N. Alachiotis, S. A. Berger, and A. Stamatakis, "Coupling simd and simt architectures to boost performance of a phylogeny-aware alignment kernel." *BMC Bioinformatics*, vol. 13, p. 196, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/bmcbi/bmcbi13.html#AlachiotisBS12

[17] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T.-W. Lam, "Soap3: Ultra-fast gpu-based parallel alignment tool for short reads," *Bioinformatics*, 2012. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/early/2012/01/28/bioinformatics.bts061.abstract

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

[19] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado, "2-d wavelet transform enhancement on general- purpose microprocessors: Memory hierarchy and simd parallelism exploitation," in *High Performance Computing HiPC 2002*, ser. Lecture Notes in Computer Science, S. Sahni, V. Prasanna, and U. Shukla, Eds. Springer Berlin Heidelberg, 2002, vol. 2552, pp. 9–21. [Online]. Available: http://dx.doi.org/10.1007/3-540-36265-7_2

[20] F. D. Igual, L. M. Jara, J. I. Gomez-Perez, L. Piñuel, and M. Prieto-Matías, "A power measurement environment for pcie accelerators," *Computer Science - Research and Development*, pp. 1–10, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00450-014-0266-8