Exploiting SIMD instructions in current processors to improve classical string algorithms*

Susana Ladra[1], Óscar Pedreira[1], Jose Duato[2], and Nieves R. Brisaboa[1]

[1] Database Laboratory. Universidade da Coruña, Spain.
{sladra|opedreira|brisaboa}@udc.es
[2] Parallel Architectures Group, Universitat Politècnica de València, Spain.
jduato@disca.upv.es

Abstract. Current processors include instruction set extensions especially designed for improving the performance of media, imaging, and 3D workloads. These instructions are rarely considered when implementing practical solutions for algorithms and compressed data structures, mostly because they are not directly generated by the compiler. In this paper, we proclaim their benefits and encourage their use, as they are an unused asset included in almost all general-purpose computers. As a proof of concept, we perform an experimental evaluation by straightforwardly including some of these complex instructions in basic string algorithms used for indexing and search, obtaining significant speedups. This opens a new interesting line of research: designing new algorithms and data structures by taking into account the existence of these sets of instructions, in order to achieve significant speedups at no extra cost.

1 Introduction

The amount of digitally available information has grown at an exponential rate during the last years, both on the Internet as within particular organizations. Efficient information processing has attracted great research effort from different areas. Algorithms and data structures that obtain efficient representations and analysis of large databases can be combined with different approaches from computer architecture, as, among others, hardware-aware implementations that exploit particular features of the hardware. For example, many algorithms have been adapted to exploit the architecture of GPUs [21], FPGAs [22], or general-purpose CPUs that provide instructions included towards improving the performance of particular application domains.

In this paper we explore how the recent SIMD (*Single Instruction Multiple Data*) included in general-purpose Intel/AMD processors can be used to improve the performance of applications of text indexing and searching.

* This work was supported by the Spanish MICINN, Plan E funds, under Grant TIN2009-14475-C04-01 (third author) and TIN2009-14560-C03-02, CDTI CEN-20091048, and Xunta de Galicia grants 2010/17 and 10SIN028E (first, second and fourth authors).

The instruction set of most general-purpose processors includes SIMD (*Single Instruction Multiple Data*) instructions thought to improve performance and power consumption in particular application areas, such as multimedia or graphics processing. Although these SIMD instructions work at a small scale when compared with special-purpose SIMD processors, the performance of algorithm implementations using them can be significantly improved. The SIMD support has evolved as processors did, considering new application areas under its scope, such as text/string processing and complex search algorithms.

Despite the benefits these facilities can bring to data management, they have been rarely used or evaluated in the existing literature [23, 19, 2, 18, 17]. The situation is similar to what happened 30 years ago in the RISC vs. CISC debate, in which complex instructions that were not used by compilers/programmers were finally removed from the instruction set of processors. However, there is a difference in the case of the SIMD instructions included in general-purpose processors. Although compilers are not able to directly use most of them by themselves, programmers can easily invoke them from high-level languages by using built-in libraries included with the compilers.

In this work we address the improvement of algorithms for text/string processing using the SIMD instructions included in the Intel SSE4.2 (*Streaming SIMD Extensions*) specification. We present case studies and experimental results that show how much text/string algorithms can benefit from the SIMD extensions.

MMX was the first Intel SIMD extension for general-purpose processors. The instructions included in the MMX specification were specially designed for improving the performance of multimedia and 3D workloads. Following MMX, new extensions appeared with the names of SSE, SSE2, SSE3, and SSE4, which include new promising instructions for text search and processing [15]. These extensions have been supported by the new generations of Intel and AMD processors, and kept in each new generation for compatibility reasons.

A first advantage of the use of SIMD instructions is that it does not imply significant additional programming cost, since the use of instructions is easy from high-level languages with the use of high-level libraries. In addition, this approach does not introduce any overhead, unlike other hardware-aware optimizations, such as GPU processing (data have to be moved to/from the device) or thread processing (threads have to be created and maintained). Moreover, this optimization does not prevent other improvements such as parallel processing, since the task can be distributed into several machines without changing the original algorithm and the sequential part processed in each node can make use of SIMD instructions. The use of this type of instructions not only improves performance but also power consumption, an important factor nowadays for companies with intensive data processing.

Portability can be seen as a drawback of this approach. However, these instructions are supposed to be kept in future processors, as it happened with previous SSE extensions, which were included in every new generation of processors [15].

As a proof of concept, in this paper we present three case studies that show how SIMD instructions can be used to improve the performance of compact data structures and algorithms for indexing and searching in strings. Particularly, we show how the *rank* and *select* operations can be implemented using these instructions, both in sequences of bits and bytes. The performance of many algorithms and data structures for text indexing and searching directly depends on the efficient implementation of rank and select, since these operations are the main component of the computational cost. We also apply this approach to the classical Horspool search algorithm [8]. We present some experiments showing that the use of this set of instructions is simple and that the results obtained are extremely competitive.

The rest of the paper is structured as follows: the next section introduces the Intel SSE4.2 instruction set and the most important instructions for the scope of this paper. Section 3 presents three case studies we have developed with the corresponding experimental evaluations. Finally, Section 4 presents the conclusions of our work.

2 Streaming SIMD Extensions 4 (SSE4)

As we have introduced in the previous section, SSE4.2 [15, 1] is the latest SIMD instruction set extension from the SSE (*Streaming SIMD Extensions*) family. It was introduced in general-purpose processors by Intel, extending the previous SSE extensions, namely SSE, SSE2, and SSE3. The original SSE included a set of new instructions that operated mainly on floating point data, with the target of increasing performance of digital signal and graphics processing. Following extensions SSE2 and SSE3 included new instructions that also considered cache-control instructions and instructions targeting efficient 3D processing. These instructions were introduced in Intel processors and in parallel supported in AMD processors, so they are present in the vast majority of general-purpose processors.

SSE4 introduced 54 new instructions, divided in the SSE4.1 (47 instructions) and SSE4.2 (7 instructions) subsets. The instructions in SSE4.2 provide string and text processing capabilities that can be used to enhance the performance of text-based applications as searching, indexing, or compression. In this section we briefly describe the instructions that we use in the rest of the paper:

- `POPCOUNT`: it counts the number of bits set to 1 in a word of 16, 32, or 64 bits. It is an important addition since most data structures and algorithms for text searching and indexing rely on the use of binary sequences and counting bits is a very common and intensively used operation (counting the number of bits set to 1 up to a given position in a bitmap is called a *rank* operation).
- `PCMPESTRI`: the name of the instruction stands for *Packed Compare Explicit Length Strings, Return Index*, and it compares two strings of at most 16 bytes. The instruction returns the position of the first different character of the two strings being compared. Several variants of this instruction exist,

depending on whether the length of the strings is specified by the user and whether the result is an index or a bit mask indicating in which positions the bytes are equal (1) and in which are not (0). The instructions `PCMPISTRI`, `PCMPESTRM`, and `PCMPISTRM` correspond to these variants of `PCMPESTRI`. The comparison to run or the format of the return value can be configured with a 7-bit mode.

In this paper we used these instructions through `gcc`. The use of SSE4.1 and SSE4.2 is supported in the `gcc` compiler (version 4.3) through built-in functions and code generation by specifying the options `-msse4.1` and `-msse4.2` when compiling. The programmer can invoke a high-level function with the name of the instruction.

3 Experimental evaluation

In this section, we will show the performance of basic string algorithms when using instructions included in the SSE4.2 extension. We first describe the machines used in the experiments, and then we present three different case studies to demonstrate the efficiency of the use of this set of instructions.

For the experiments in this paper we used is an Intel® Core™ i5 CPU 750 @ 2.67GHz (4 cores) with 16 GB RAM. It ran Ubuntu GNU/Linux with kernel version 2.6.38-11-generic (64 bits). We compiled with gcc version 4.5.2 and the option `-O3` and `-msse4.2` to enable SSE4.2 extensions. In some experiments we also used an Intel®Xeon® -E5520@2.26GHz with 72 GB DDR3@800MHz RAM. It ran Ubuntu 9.10 (kernel 2.6.31-19-server), using gcc version 4.4.1. If no further specifications are made, the machine used in the experiments is the Intel i5.

3.1 *Rank* and *select* over bit strings

Bit strings are frequently used in numerous scenarios, including succinct data structures, which represent data, such as sets, trees, hash tables, graphs or texts, using as little space as possible while retaining its original functionality.

Given a sequence of bits $B_{1,n}$, we define three basic operations:

- $rank_b(B, i)$ counts the number of times the bit b appears in B up to position i. If no specification is made, *rank* stands for $rank_1$ from now on.
- $select_b(B, j)$ returns the position of the $j - th$ appearance of bit b in B. Analogously to *rank*, *select* stands for $select_1$ if no bit specification is made.
- $access(B, i)$ returns whether position i of sequence B contains 0 or 1.

Several strategies have been developed to efficiently compute *rank* and *select* when dealing with binary sequences. They are usually based on building auxiliary structures that lead to a more efficient management of the sequence, such as a two-level directory proposed by Jacobson [9]. One practical solution, we denote by `GGMN`, was proposed by González et al. [6], where precomputed *popcount* tables are used [11]. Popcounting consists of counting how many bits are set in a bit

array. By using tables where this counting is already computed for small arrays of 8 bits, *rank* and *select* operations can be efficiently solved with a parameterizable space overhead. `GGMN` builds a one-level directory structure over the bit sequence consisting of blocks of size k that store $rank_1(B, p)$ for every p multiple of k. To compute $rank_1(B, i)$ it first obtains the stored number of times the bit 1 appears before the block containing the position i and then it requires a sequential scan to count all the set bits in the block up to position i. This sequential scan uses a *popcount* procedure over each 32-bit sequence, which computes the number of bits set to 1 in a 32-bit integer x by performing:

$$tab[(x\texttt{>>}0)\&\texttt{0xff}] + tab[(x\texttt{>>}8)\&\texttt{0xff}] + tab[(x\texttt{>>}16)\&\texttt{0xff}] + tab[(x\texttt{>>}24)\&\texttt{0xff}],$$

where *tab* is the precomputed *popcount* table that contains the number of 1s in each different byte value. The space overhead is $1/k$, thus this solution offers an interesting space/time tradeoff. We will use this solution with $k = 20$ such that just 5% of extra space is needed, while still computing *rank* and *select* efficiently.

For the experimental evaluation of this case study, we just replace the *popcount* procedure of the practical solution with the `POPCOUNT` SSE4.2 instruction. To call `POPCOUNT` SSE4.2 instruction we just need to use the built-in function *__builtin_popcount* or use *_mm_popcnt_u32* and *_mm_popcnt_u64* included in *nmmintrin.h* library.

We evaluate the original implementation of `GGMN` with a straightforward replacement of the popcount procedure, `GGMN with SSE4.2` in three different scenarios that require *rank* operations over bit strings:

1. *Scenario 1*: We evaluate the performance of *rank* operation by computing the number of bits up to all the positions of a bitmap of length 1,000,000,000 in random order. The bitmap was generated by setting up bits at random positions. We compute the average time to compute a *rank* operation over the bit string.
2. *Scenario 2*: We use `GGMN with SSE4.2` in the original implementation of a web graph compression method, *RPGraph* [5], which represents a Web graph based on the Re-Pair compression method. We compute the time to rebuild a graph from its compressed representation and measure average CPU user time per neighbor retrieved.[3]
3. *Scenario 3*: We use `GGMN with SSE4.2` in an implementation [20] of the compressed suffix tree proposed by Sadakane [16]. We measure the average time to compute the longest common substring for each pair of sequences from a set of 100 DNA sequences, whose average read length is 470.29.[4]

Table 1 shows the results obtained for the three scenarios. The second and third column indicate the time per operation (which is scenario-dependant, as

[3] We use the source code at http://webgraphs.recoded.cl/index.php?section=rpgraph to represent the graph EU with the parameters indicated in the example of use.

[4] We use the source code at http://www.cs.helsinki.fi/group/suds/cst. The sets of DNA sequences were generated by a 454 Sequencing System from one individual Melitaea cinxia (a butterfly).

Table 1: Experimental evaluation of popcount SSE4.2 instruction

Scenario	`GGMN` Time	`GGMN with SSE4.2` Time	Speedup ratio	% time *rank* oper.	SSE4.2 instruction used
1	0.133 ns	0.068 ns	1.95	100.00%	*_mm_popcnt_u32*
2	0.150 μs	0.129 μs	1.16	37.60%	*_mm_popcnt_u32*
3	2.893 ms	2.299 ms	1.26	31.30%	*_mm_popcnt_u64*

we explained above). The fourth column shows the speedup ratio obtained (we divide the second column by the third column). The fourth column of the table indicates the percentage of time consumed by the *rank* operation when no SSE4.2 instructions are used. This value was computed with `gprof` profiler. The last column shows the SSE4.2 instruction used.

As we can observe, we can easily accelerate the practical implementation of algorithms that require *rank* operations by directly using popcount SSE4.2 instruction. Depending on the percentage of *rank* operations required by the algorithms that operate over the data structure, the speedup obtained can vary up to 2, if the popcount procedure is intensively used. We also obtain a greater speedup when replacing a 64-bit popcount procedure with the 64-bit SSE4.2 instruction, as in Scenario 3.

Notice that `GGMN` solution is parameterizable, k being the size of the blocks in the data structure. If we vary k, the block size of the auxiliary data structure for *rank* and *select*, we obtain a space/time tradeoff. Hence, we can not only accelerate `GGMN` by using SSE4.2 instructions. We can instead improve the space required. For example, in Scenario 1, by using SSE4.2 instructions we can achieve the same time performance as with the original implementation while reducing the extra space required from 5% ($k = 20$) to 1.25% ($k = 80$). In addition, when we use a higher k value, the speedup obtained is greater. Effectively, since blocks are larger, the number of calls to the popcount procedure is also higher. Hence, the speedup obtained with the replacement of the procedure by the popcount SSE4.2 instruction is greater (we can obtain speedups greater than 3).

3.2 *Rank*, *select* and *access* over byte strings

Rank, *select* and *access* operations can be extended to arbitrary sequences S with an alphabet Σ of size σ. In this case, given a sequence of symbols $S = S_1 S_2 \dots S_n$ and a symbol $s \in \Sigma$, $rank_s(S, i)$ returns the number of times the symbol s appears in the sequence up to position i, that is, in $S[1, i]$; $select_s(S, j)$ returns the position of S containing the j-th occurrence of the symbol s; and $access(S, i)$ returns the i-th symbol of sequence S, that is, S_i. It may be a necessary operation, since S is commonly represented in a compact way.

For some scenarios, the strategies used with binary sequences can be efficiently adapted to the general case. A simple generalization of Jacobson's idea has been proven successfully for byte strings [4]. We will use this solution, which we call "Sequential+Blocks", to prove the performance of SSE4.2 instructions in

this scenario. It consists of representing the original byte sequence in plain form and using an auxiliary data structure to support byte-wise rank/select operations. Given a sequence of bytes $B[1,n]$, it builds a two-level directory structure, dividing the sequence into sb superblocks and each superblock into b blocks of size $n/(sb \cdot b)$. The first level stores the number of occurrences of each byte from the beginning of the sequence to the start of each superblock. The second level stores the number of occurrences of each byte up to the start of each block from the beginning of the superblock it belongs to. The second-level values cannot be larger than $sb \cdot b$, and hence can be represented with fewer bits. Thus, $rank_{b_i}(B,j)$ is obtained by counting the number of occurrences of b_i from the beginning of the last block before j up to the position j, and adding to that the values stored in the corresponding block and superblock for byte b_i. To compute $select_{b_i}(B,j)$ we binary search for the first stored value x such that $rank_{b_i}(B,x) = j$. We first binary search the values stored in the superblocks, then those in the blocks inside the right superblock, and finally complete the search with a sequential scanning in the right block. This structure answers *rank* in time $O(n/(sb \cdot b))$ and *select* in time $O(\log sb + \log b + n/(sb \cdot b))$.

In this section we slightly modify the algorithm of the sequential solution to incorporate SSE4.2 instructions, since we can count the number of occurrences of a byte value inside a block in a more efficient way with the instructions `PCMPESTRM` and `POPCOUNT`. The complete source code of this sequential scan for *rank* operation is described in Code 1.1. The modification of the sequential scan of a block for *select* operation is analogous. As it can be observed in the included pseudocode, we process the sequence in blocks of 16 bytes, which produces a significative speedup as we report in the experimental evaluation of this section.

Code 1.1: Sequential *rank* in a block using SSE4.2 instructions

```
#include <nmmintrin.h>
#include <emmintrin.h>

const int mode =  _SIDD_UBYTE_OPS | _SIDD_CMP_EQUAL_EACH | _SIDD_BIT_MASK;

uint seqRank(uint* vector, byte searchedByte, uint position){
   register uint i, cont = 0;
   __m128i patt, window, returnValue;
   byte *c1, patt_code[16];
   uint d = position>>4, r = position & 0xf;
   for(i=0;i<16;i++)
      patt_code[i]=searchedByte;
   long long * pat_array = (long long *)patt_code;
   patt = _mm_set_epi64x(pat_array[1],pat_array[0]);
   long long * text_array = (long long *)vector;
   for(i=0;i<d;i++){
      window = _mm_set_epi64x(text_array[1],text_array[0]);
      returnValue = _mm_cmpestrm(patt, 16,window, 16, mode);
      cont+=_mm_popcnt_u32(_mm_extract_epi32(returnValue,0));
      text_array += 2;
   }
   window = _mm_set_epi64x(text_array[1],text_array[0]);
   returnValue = _mm_cmpestrm(patt, r,window, r,  mode);
   cont+=_mm_popcnt_u32(_mm_extract_epi32(returnValue,0))+r-16;
   return cont;
}
```

Impact of the SSE4.2 alternative on rank and select over byte strings. In order to test the efficiency of this new practical implementation of the sequential solution, which we call "Sequential+Blocks with SSE4.2", we ran some experiments to compute *rank* and *select* operations over 3 byte sequences[5]. We denote them `Bytemap 1`, `Bytemap 2`, and `Bytemap 3`. Figure 1a shows the frequency distribution of all the byte values on those byte sequences, whose sizes (in bytes) are 228,707,250; 834,670; and 65,536 respectively.

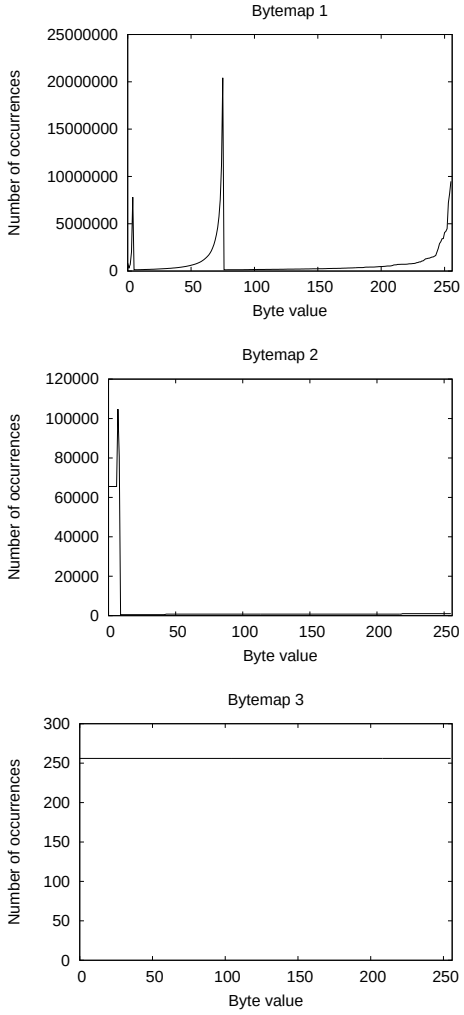
In addition to the sequential solution (without SSE4.2 instructions and with SSE4.2 instructions), we include in the comparison three different approaches using wavelet trees, which are some of the most competitive solutions to solve rank/select operations over arbitrary sequences and transform the problem to solve binary rank/select operations. We include a balanced binary wavelet tree with no bitmap compression [7], denoted by "binary WT", a Huffman-shaped wavelet tree [12], denoted by "huff-shaped WT" and a balanced binary wavelet tree using Raman et al. solution for the rank/select operation over the binary sequences at each level [14], denoted by "binary WT + RRR".[6]

We used several configuration of parameters to obtain a space/time tradeoff. The space usage is shown as the space required by the whole representation of the byte sequence and extra structures in main memory as a percentage of the size of the original byte sequence. We computed the average time to perform *rank*, *select* and *access* operations over the byte sequences. More precisely, the *rank* time was measured by computing *rank* operations of random byte values over all the positions of the sequence in random order; *select* time was measured by searching random occurrences of random byte values, then computing the average time; and *access* time was computed by obtaining the byte value at all the positions of the sequence in random order. Times are shown in μs per operation.

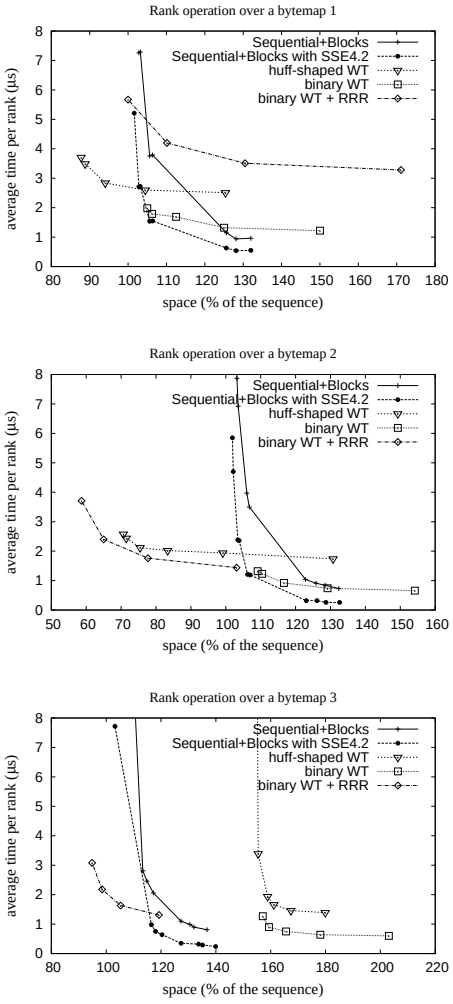
Subfigures 1b to 2b illustrate the behavior of the different techniques to answer *rank*, *select* and *access* operations. As it can observed from the figures, we can significantly improve the sequential implementation by including SSE4.2 instructions in the sequential scan of the blocks, where the times obtained with "Sequential+Blocks with SSE4.2" are around three times faster than the times obtained with "Sequential+Blocks". This improvement makes the sequential solution a more attractive solution when we require to compute *rank* and *select*, also considering that this solution can directly solve *access* operation, which is not trivially solved with the wavelet-tree-based alternative techniques, as illustrated in Subfigure 2b. Notice that this may not be a completely fair comparison, since "binary WT + RRR", "huff-shaped WT" and "binary WT" methods could also be accelerated by using SSE4.2 instructions. However, it just makes evident

[5] The byte strings correspond to three byte sequence from a tree-shaped indexing data structure, Wavelet Trees on Bytecodes (WTBC) using Plain Huffman encoding over the ALL corpus, as described in [4]. More precisely, we extracted the byte sequence from the leftmost node at each level of the WTBC.

[6] We use the implementations of the Compact Data Structures Library (libcds) available at http://libcds.recoded.cl/

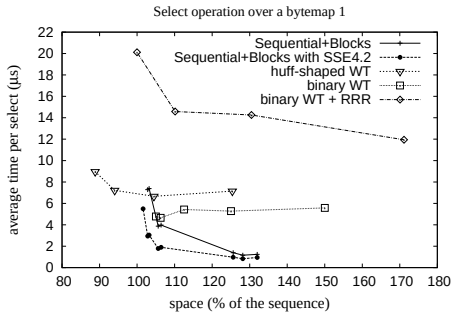


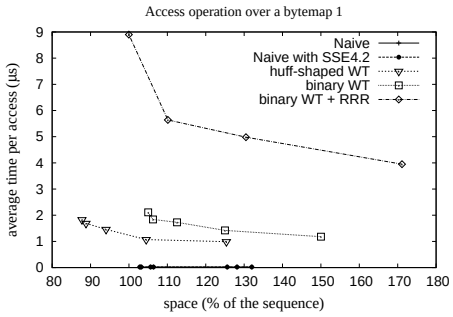
(a) Frequency distribution of the byte values for three different bytemaps.

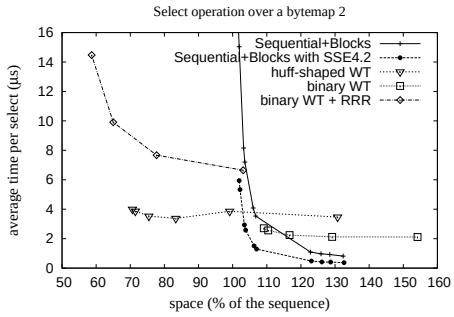


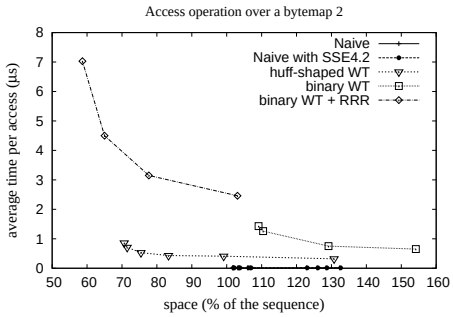
(b) Space/time tradeoff for *rank* operation over the three byte sequences using different techniques.

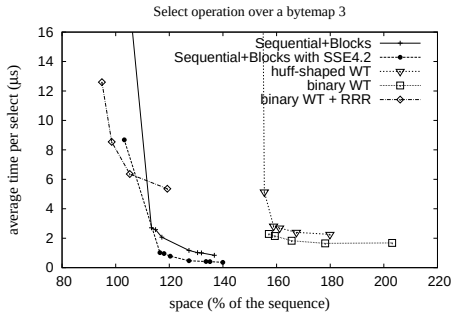
Fig. 1: Experimental results for *rank* operation over different byte strings



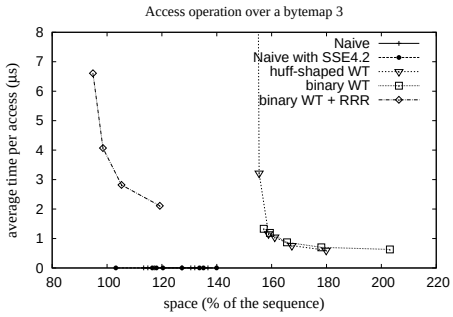








(a) Space/time tradeoff for random *select* operations over the three byte sequences using different techniques.



(b) Space/time tradeoff for *access* operation over the three byte sequences using different techniques.

Fig. 2: Experimental results for *select* and *access* over different byte strings

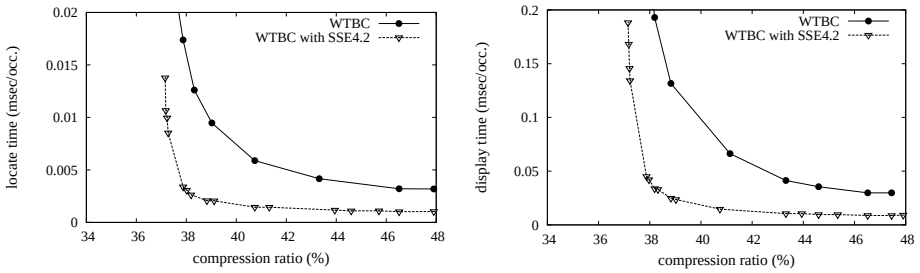


Fig. 3: Influence of SSE4.2 instructions on the WTBC data structure.

that an a priori sequential solution can be more efficient than a theoretically better one if we accelerate its practical implementation with some algorithm engineering.

Impact of the SSE4.2 alternative on a compressed data structure. We compare now the performance of the new practical implementation of "Sequential+Blocks with SSE4.2" on the WTBC data structure [4], which is a word-oriented self-index for natural language text. It is based on a byte-wise wavelet tree, thus it requires to compute *rank*, *select* and *access* operations over byte strings in an efficient way. Its original implementation uses the sequential solution to support *rank* and *select*. We ran these experiments on the Intel Xeon, and we computed *locate* and *display* operations searching for a set of 429 queries over the INEX 2009 Wikipedia Dataset[7] without XML tags. It consists of 8.76GiB of plain text (1.8×10^9 words), and a vocabulary of 14.88 million words.

As we can see in Figure 3, the speedup obtained in *rank* and *select* operations for the isolated byte strings is practically the same when evaluating the overall performance of the whole data structure. Thus, using SSE4.2 instructions makes WTBC almost three times faster. This is due to the intensive use of *rank* and *select* operations during the overall navigation of the WTBC data structure.

3.3 Searches over character strings

In this section, we will show how SIMD instructions can also improve the efficiency of classical pattern matching search algorithms in strings. Particularly, we focus on the well-known Horspool string search algorithm [8], which is a simplified version of the Boyer-Moore algorithm [3]. This algorithm is frequently found in indexing structures such as block-addressing inverted indexes for natural language texts [13, 10].

The idea of these algorithms is to traverse the string skipping some bytes during the search, so it is not necessary to compare the whole string against the

[7] http://www.mpi-inf.mpg.de/departments/d5/software/inex.

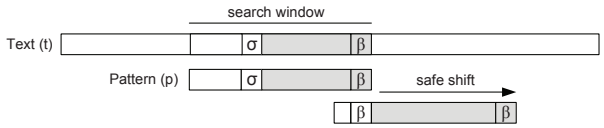


Fig. 4: Horspool algorithm

search pattern. The Horspool algorithm uses a search window with the size of the search pattern that will be moved along the text during the search from left to right (see Figure 4). For each position of the search window, the algorithm carries out two steps. The first one is comparison, that is, the search window is compared with the pattern from right to left until an occurrence of the pattern is found or the comparison fails at a given character.

The next step is the shift of the window to its next position. The shift should be as large as possible but safe, that is, without missing a potential occurrence of the pattern in the string. If β is the last character of the search window, the shift distance is given by the distance from the last occurrence of β in the pattern to the end of the pattern. If the last character of the search window does not occur in the search pattern, the shift distance is the size of the pattern. The shift distance is not computed in each step, but it is precomputed for each character present in the search pattern before starting the traversal of the text.

The application of SIMD instructions in the Horspool algorithm is straightforward. The comparison of the search window with the pattern can be carried out with the `PCMPESTRI` instruction, so the comparison is not done one character at a time, but up to sixteen characters at a time. The fact that the comparison is performed from right to left does not affect the use of this instruction[8]. Note that if the search pattern has less than sixteen characters, each comparison of the search window is carried with one CPU instruction.

Table 2 shows the results we obtained when comparing a sequential implementation and a SIMD-based implementation of Horspool algorithm. In order to compare the two implementations we used the first 1 GB of the INEX corpus used in the previous section. We searched for 200 patterns with both implementations and computed the average time per pattern. The average number of characters per pattern is 9.125. We ran the same experiments in two different machines, the Intel i5 and the Intel Xeon described at the beginning of this section, using in both machines the executable file built with gcc version 4.5.2 in the Intel i5. We observe that we can accelerate Horspool algorithm by including SSE4.2 instructions, and the speedup obtained is practically the same for both machines, which shows that the use of SSE4.2 is portable to different machines.

[8] We use `PCMPESTRI` instruction with mode: `_SIDD_UBYTE_OPS` | `_SIDD_CMP_EQUAL_EACH` | `_SIDD_MOST_SIGNIFICANT` | `_SIDD_NEGATIVE_POLARITY`.

Table 2: Experimental evaluation of SSE4.2-based Horspool search

Setting	Sequential Time (s)	SSE4.2 Time (s)	speedup ratio
Intel i5	0.879	0.772	1.139
Intel Xeon	1.236	1.086	1.138

4 Conclusions

The case studies and experimental evaluations we have presented in this paper reveal how algorithms for text/string processing can benefit from hardware-aware implementations that exploit the SSE4.2 SIMD instructions included in recent general-purpose processors. This is a remarkable result because the use of these basic string algorithms is ubiquitous in many sorts of applications, specially for indexing and searching. In the paper we just prove that we can obtain significant speedups by the straightforward use of SSE4.2 instructions in our practical implementations. We plan as future work to propose new algorithms and data structures taking into account the existence of these sets of instruction extensions during the design process.

The use of the SSE4.2 does not imply any additional computational cost when compared with other optimizations based on particular hardware features, such as GPU programming or using threads. In addition, its use is orthogonal to the use of other optimization techniques, such as parallelism or GPUs, since SSE4 extensions focus on improving the efficiency of the sequential part of the algorithm. Future work includes analyzing under which conditions the usage of each approach, or even their combination, is more convenient.

Portability can be seen as a potential drawback of this approach, although the instructions included under the SSE4.2 extension are supposed to be kept in future generations of processors by various vendors, as it has happened with previous instruction set extensions of the SSE family.

References

1. Intel SSE4 Programming Reference, July 2007. Reference Number: D91561-003.
2. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weiman. Optimal packed string matching. In *Procs. FSTTCS*, pages 423–432, 2011.
3. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM (CACM)*, 20(10):762–772, 1977.
4. N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Procs. SIGIR*, pages 139–146, 2008.
5. F. Claude and G. Navarro. Fast and compact web graph representations. *ACM TWEB*, 4:16:1–16:31, September 2010.
6. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Procs. WEA*, pages 27–38, 2005.
7. R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *Procs. SODA*, pages 841–850, 2003.

8. R. N. Horspool. Practical fast searching in strings. *Software: Practice and Experience (SPE)*, 10(6):501–506, 1980.
9. G. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1989.
10. U. Manber and S. Wu. Glimpse: a tool to search through entire file systems. In *Procs. of USENIX Winter Technical Conf.*, pages 4–4, 1994.
11. I. Munro. Tables. In *Procs. FSTTCS*, LNCS 1180, pages 37–42, 1996.
12. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
13. G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Inf. Retr.*, 3(1):49–77, 2000.
14. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Procs. SODA*, pages 233–242, 2002.
15. Ramanathan R.M. Extending the world's most popular processor architecture. Technical report, Intel Corporation, 2006.
16. K. Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41:589–607, 2007.
17. B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. In *Procs. of ADMS*, 2011.
18. G. Shi, M. Li, and M. Lipasti. Accelerating search and recognition workloads with sse 4.2 string and text processing instructions. In *Procs. ISPASS*, ISPASS '11, pages 145–153, 2011.
19. A. Stepanov, A. Gangolli, D. Rose, R. Ernst, and P. Oberoi. Simd-based decoding of posting lists. In *Procs. CIKM*, pages 317–326, New York, NY, USA, 2011. ACM.
20. N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. Engineering a compressed suffix tree implementation. *ACM JEA*, 14:2:4.2–2:4.23, 2010.
21. L. Wang, M. Huang, V. K. Narayana, and T. El-Ghazawi. Scaling scientific applications on clusters of hybrid multicore/gpu nodes. In *Procs. of ACM CF*, pages 6:1–6:10. ACM Press, 2011.
22. L. Woods. Fast data analytics with fpgas. In *Procs. of ICDE Workshops*, pages 296–299. IEEE Press, 2011.
23. J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Procs. of SIGMOD*, SIGMOD '02, pages 145–156. ACM, 2002.