HPC on the Intel Xeon Phi: Homomorphic Word Searching

Paulo Martins(✉) and Leonel Sousa

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal
paulo.sergio@netcabo.pt, las@inesc-id.pt

Abstract. In this paper, the suitability of implementing parallel homomorphic word searching on Intel Xeon Phi coprocessors is evaluated for the first time. Homomorphic encryption allows to produce a cryptogram that encrypts the result of applying some values to any function, even when the input values are encrypted and without access to the private-key. For example, it is possible to search if any word of a set of encrypted words matches a plaintext reference word and generate a new cryptogram that encrypts the amount of matches. In this paper it is shown that this operation is about 834 times faster by using a system with 4 Intel Xeon Phi coprocessors 5110P attached to an Intel Xeon CPU E5-2630 v2, when compared with an implementation on a single core of the Xeon CPU.

Keywords: Intel Xeon Phi · Homomorphic encryption · Homomorphic word searching

1 Introduction

The use of embedded systems is becoming ubiquitous, as more sensors and actuators are incorporated into everyday electronics and on the general infrastructure. Since these devices often have limited computational resources, it would be beneficial to offload parts of their computation to a third party. However, the processed data may be private, which means that the third party should not have access to it. Cryptography enables the encryption of data, such that access to it is impossible without the usage of a specific key. In particular, with public-key cryptography, every user produces a pair of keys: one is public, and should be widely distributed, while the other is private. Someone with access to the public-key may produce a cryptogram by applying the encryption algorithm to a plaintext. This cryptogram cannot be decrypted by anyone but the owner of the corresponding private-key. Homomorphic Encryption (HE), in turn, allows one to operate on ciphered data [2]. With this approach, one can produce an encryption of the output of an arbitrary function from the encrypted inputs, and without access to the deciphering key.

P. Martins—This work was supported by national funds through FCT (Fundação para a Ciência e a Tecnologia) with reference UID/CEC/50021/2013 and under the Ph.D. grant with reference SFRH/BD/103791/2014.

I. Dutra et al. (Eds.): VECPAR 2016, LNCS 10150, pp. 75–88, 2017.
DOI: 10.1007/978-3-319-61982-8_9

With homomorphic encryption, processors with limited resources may offload the processing of sensitive data without compromising the privacy of that data. This may be useful for instance, for advertising companies that wish to select ads based on the browsing history or keywords in e-mails of potential customers; or to analyze medical data of sensors connected to a patient without breaching the patient's privacy; or for companies wishing to offload the computation of the purchasing patterns of its customers. It is therefore of interest to investigate how current servers may execute these procedures in an efficient and scalable way. Some platforms to provide High Performance Computing (HPC) in server settings include various Field-Programmable Gate-Array (FPGA) solutions, the use of Graphics Processing Units (GPUs) for general purpose processing, among others. However, these solutions typically lead to high development and maintenance costs.

As an alternative, Intel developed the Many Integrated Core (MIC) architecture [5]. Intel MIC is a many core coprocessor architecture supported on a modified version of the P54C design, used on the original Pentium. These cores can be very power efficient on current semiconductor process architectures due to short pipelines and low frequency operations. Also, the modified cores enable the use of many of the programming models that most developers are already accustomed to, such as OpenMP, OpenCL, Message Parsing Interface (MPI), Cilk/Cilk Plus, and specialized versions of Fortran, C++ and math libraries. This is twofold important. First, there is a large amount of code already being deployed with these tools, that can be readily executed on the Intel Xeon Phi. Second, it eases the process of porting applications to the new architecture. We consider, therefore, that this architecture is one of the most suitable for servers with heterogeneous workloads, and investigate for the first time how well it is adapted to homomorphic word searching.

The rest of the paper is organized as follows. In Sect. 2, we give an introduction to HE, and describe how it can be applied to perform word searching. Afterwards, in Sect. 3, procedures and algorithms are proposed for the Intel Xeon Phi architecture. The performance of the proposed parallel algorithms is evaluated in Sect. 4, compared with related work in Sect. 5, and finally conclusions are drawn in Sect. 6.

2 Homomorphic Encryption

HE can be metaphorically explained by the jewelry shop problem [3], whose solution is represented in Fig. 1. Alice, a shop owner, wanted her workers to assemble precious materials, such as gold and diamonds, into intricately designed rings and necklaces. However, she distrusted her workers, and thus did not want the workers to come in direct contact with the materials, since she was afraid they might steal them. In order to solve this problem, Alice used a transparent impenetrable glovebox. She would then open the box, and store the raw materials inside. Afterwards, she would lock the box using a key to which only she had access. This process embodies the encryption procedure. As shown in Fig. 1,

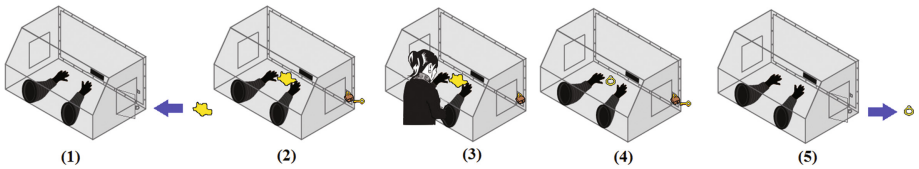

Fig. 1. A piece of gold is locked inside a glovebox, so that a worker may transform it into a ring. The ring is later removed when the glovebox is unlocked

the workers could use the gloves to assemble the rings and necklaces. In this situation, the gloves represent the homomorphism of an HE scheme. Since the box was impenetrable they could not have access to the precious materials; in a similar manner to how a server is not be able to access the encrypted values it processes homomorphically. When the piece was finished, Alice could open the box, and retrieve the result, which is mirrored as the decryption operation in the case of a cryptosystem.

Fully Homomorphic Encryption (FHE) was first uncovered in 2009 [2]. In contrast to previous Somewhat Homomorphic Encryption (SHE) schemes that only enabled a subset of all possible operations on cryptograms, with FHE it is possible to arbitrarily process encrypted data. In this paper, the cryptosystem described in [6] will be focused on. This cryptosystem is a leveled FHE scheme: with it, it is possible to evaluate arbitrary functions of encrypted data; one only needs to specify the maximum "size" of functions beforehand and the size of the generated public-key depends on this value. Arithmetic is performed in $R_q = \frac{\mathbb{Z}/(q\mathbb{Z})[x]}{\Phi(x)}$, with $\Phi(x) = x^n + 1$ and n a power of two. In this ring, two elements are congruent (i.e. equivalent) if their difference is a multiple of $\Phi(x)$. Similarly, two elements $a(x)$ and $b(x)$ are congruent if the difference between all corresponding coefficients a_i and b_i is a multiple k_i of q. An example of operations in this ring for $n = 2$ and $q = 2$ is as follows:

$$\begin{aligned} (x+1) + x &= 2x + 1 \equiv \quad 1 \\ (x+1) \times x &= x^2 + x \equiv x + 1 \end{aligned} \tag{1}$$

where $\equiv$ is used to denote congruency. In the first case, if we consider the coefficients of x, one can see that $2 - 0 = 2$ is a multiple of $q = 2$, and therefore the congruency is valid. In the second equation, we can see that $x^2 - (-1)$ is a multiple of $\Phi(x) = x^2 + 1$, hence $x^2 + x \equiv x - 1$. Since $(-1) - 1$ is also a multiple of $q = 2$, the second congruency is valid in this ring. Typically, the elements of R_q with the smallest polynomial degrees, and with the smallest non-negative coefficients are used as the representatives for the congruency classes. With this representation, addition and multiplication of polynomials is followed by the computation of the remainder of the division by $\Phi(x)$, and afterwards by the computation of the remainder of the division of the coefficients by q.

In this cryptosystem, the secret-key corresponds to a vector $s_{2\times 1} = (1, -t)^T \in R_q^2$, where $t \leftarrow D_{R_q,\sigma_k}$ is a polynomial drawn from a "narrow" distribution [6], namely a Gaussian distribution. In order to produce the corresponding

public-key, $a \leftarrow R_q$ is drawn uniformly from R_q, $e \leftarrow D_{R_q,\sigma_k}$ is produced, and $b = at + e$ is computed. The public-key corresponds to $A_{1\times 2} = (b, a)$. Note that

$$A_{1\times 2} \times s_{2\times 1} = e \tag{2}$$

where e is a "small" polynomial. A cryptogram $C_{N\times 2}$, with $N = 2l$ and $l = \lceil \log q \rceil$, encrypting a value $\mu \in R_q$ is a matrix such that, for a small *error*:

$$C_{N\times 2} \times s_{2\times 1} = \mu \begin{bmatrix} 1 & 0 \\ 2 & 0 \\ \vdots & \vdots \\ 2^{l-1} & 0 \\ 0 & 1 \\ 0 & 2 \\ \vdots & \vdots \\ 0 & 2^{l-1} \end{bmatrix} \times s_{2\times 1} + error \tag{3}$$

If we add two cryptograms $C_{N\times 2}$ and $D_{N\times 2}$, the resulting cryptogram retains the format described in Eq. (3). Homomorphic multiplication, in contrast, is more complex. To multiply two ciphertexts $C_{N\times 2}$ and $D_{N\times 2}$, one computes $BD(C_{N\times 2}) \times D_{N\times 2}$. The $BD(C_{N\times 2})$ function expands each entry of the matrix across l columns performing bit decomposition. In concrete, each element $c_{i,j}$ is decomposed into $c_{i,j}[k]$ for $k \in \{0, \ldots, l-1\}$, such that $c_{i,j} = \sum_{k=0}^{l-1} c_{i,j}[k]2^k$, where the $c_{i,j}[k]$ are polynomials with coefficients either 0 or 1, producing a matrix:

$$\begin{pmatrix} c_{0,0}[0] & \ldots & c_{0,0}[l-1] & c_{0,1}[0] & \ldots & c_{0,1}[l-1] \\ c_{1,0}[0] & \ldots & c_{1,0}[l-1] & c_{1,1}[0] & \ldots & c_{1,1}[l-1] \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ c_{N-1,0}[0] & \ldots & c_{N-1,0}[l-1] & c_{N-1,1}[0] & \ldots & c_{N-1,1}[l-1] \end{pmatrix} \tag{4}$$

It can be proved that the result of $BD(C_{N\times 2}) \times D_{N\times 2}$ retains the format in (3), but μ now takes the value of the product of the original plaintexts. If the term *error* remains small enough, the result can still be deciphered.

As more operations are performed, *error* grows, and as such the number of operations that can be applied are limited by the homomorphic capacity of the cryptosystem. In the particular case of homomorphic multiplication, noise growth is asymmetric, i.e. if matrices $C_{N\times 2}$ and $D_{N\times 2}$ are swapped in the expression $BD(C_{N\times 2}) \times D_{N\times 2}$, the final *error* will not necessarily be the same. It is best to use the ciphertext with the largest error as $C_{N\times 2}$ [6].

2.1 Ring Arithmetic

The parameters for the cryptosystem described in [6] enable efficient arithmetic over R_q. In particular, the value of n was set to $n = 1024$ and q to $q = \texttt{0x7FFE0001}$ in hexadecimal. This leads to $l = 31$ and $N = 62$. Reduction

modulo q after multiplications is achieved by noting that the following congruence is valid (both side of the expression have the same remainder modulo q):

$$2^{31} \equiv 2^{17} - 1 \bmod q \tag{5}$$

since $q = 2^{31} - 2^{17} + 1$. Thus, the value $z \in \{0, \ldots, (q-1)^2\}$ of the product of two polynomial coefficients can be rewritten as $z = z_1 2^{31} + z_0$, and the following congruence can be applied:

$$z \equiv z_1 2^{17} + z_0 - z_1 \tag{6}$$

This latter congruence is iteratively employed as depicted in Algorithm 1 until $z \in \{0, \ldots, 2^{31} - 1\}$. Afterwards, a conditional subtraction by q when $z \in \{q, \ldots, 2^{31} - 1\}$ suffices to ensure that z is in $\{0, \ldots, q-1\}$. When adding or subtracting two polynomial coefficients, a subtraction or an addition by q suffices to bring the result z back to $\{0, \ldots, q-1\}$ when $z \geq q$ or $z < 0$, respectively.

Algorithm 1. Modular reduction in $\mathbb{Z}/(q\mathbb{Z})$

Require: $z \in \{0, \ldots, q^2 - 2q + 1\}$
Ensure: $z \in \{0, \ldots, q-1\}$
  while $z \geq 2^{31}$ **do**
    $z_1 = z >> 31$
    $z_0 = z\&(2^{31} - 1)$
    $z = z_1 2^{17} + (z_0 - z_1)$
  end while
  if $z \geq q$ **then**
    $z = z - q$
  end if
  return z

Addition of polynomials in R_q is performed by adding the corresponding polynomial coefficients in $\mathbb{Z}/(q\mathbb{Z})$ with Single Instruction Multiple Data (SIMD) instructions. Multiplication of two polynomials in $\frac{\mathbb{Z}/(q\mathbb{Z})[\dot{x}]}{\dot{x}^n - 1}$ is equivalent to a cyclic convolution of n points: if $u(\dot{x}) = \sum_{i=0}^{n-1} u_i \dot{x}^i$, $u(\dot{x})\dot{x}^k \equiv \sum_{i=0}^{n-1} u_{n-k+i \bmod n} \dot{x}^i \bmod \dot{x}^n - 1$, thus $z(\dot{x}) = u(\dot{x}) \times v(\dot{x}) \equiv \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} u_{n-j+i \bmod n} v_j \dot{x}^i \bmod \dot{x}^n - 1$. This operation is equivalent to multiplying the coefficients of the Fast Fourier Transform (FFT) over $\mathbb{Z}/(q\mathbb{Z})$ of the two polynomials, which results in an algorithm with lower complexity. By noting that if $\eta^n \equiv -1 (\bmod q)$, and $\dot{x} = \eta x$, then

$$\dot{x}^n - 1 = (\eta x)^n - 1 \equiv -(x^n + 1) \equiv 0 (\bmod x^n + 1) \tag{7}$$

This means that if the change of variable $\dot{x} = \eta x$ is applied, operations modulo $\dot{x}^n - 1$ will be converted to operations modulo $x^n + 1$ when the variable is changed back to $x = \eta^{-1}\dot{x}$.

Thus, to multiply two polynomials $a(x)$ and $b(x)$ in R_q, one first computes $u(\dot{x}) \equiv \sum_{i=0}^{n-1} \underbrace{a_i \eta^{-i}}_{u_i} \dot{x}^i$ and $v(\dot{x}) \equiv \sum_{i=0}^{n-1} \underbrace{b_i \eta^{-i}}_{v_i} \dot{x}^i$. Afterwards, one applies a FFT over $\mathbb{Z}/(q\mathbb{Z})$ to u and v, and multiplies the resulting transforms coefficient-wise. To get the final result, an inverse FFT has to be applied, and a final change of variable to return the polynomials from $\dot{x}$ to x.

2.2 Homomorphic Word Matching

In this work, the previous scheme was applied to homomorphically perform word matching. One can imagine a server where e-mails are stored in encrypted format. The senders of e-mails should encrypt the words of those e-mails by applying Algorithm 2 to the set of words in the e-mail. It should be noted that since a hash function is used to conceal the words lengths it is not possible to obtain the plaintext words back from the cryptograms. For practical implementations, the sender would have to cipher the e-mail twice, once where all the words are encrypted with this algorithm, and another time where the e-mail is encrypted as a whole with a "reversible" encryption.

Algorithm 2. Encryption of a list of words to be searched

Require: List of words to be searched, *input_list*
Ensure: List of encrypted words, *output_list*
  output_list = {}
  for all *word* **in** *input_list* **do**
    encrypted_bit_list = {}
    a = Hash(*word*)
    for all bit a_i **in** a **do**
      c_i = Encrypt(a_i)
      encrypted_bit_list = *encrypted_bit_list* $\cup \{c_i\}$
    end for
    output_list = *output_list* $\cup$ {*encrypted_bit_list*}
  end for
  return *output_list*

The e-mail client could then issue word searching queries. We assume, for simplicity, that the queried word is provided in the clear. The e-mail server would iterate through all encrypted words, and apply Algorithm 3 to each of them and the plaintext queried word. In this algorithm, one starts by computing the hash of the word to be searched, producing a. Afterwards, the value of Encrypt($\prod_i a_i$ XNOR b_i) is homomorphically computed, where b_i is the i^{th} bit of the hash of the encrypted word. When processing the cryptograms, the term i in the product has the value of c_i if $a_i = 1$, and Encrypt(1) $- c_i$ otherwise, where c_i is the encryption of b_i. This operation produces a cryptogram that encrypts 1 when the two words are the same, or 0 otherwise. It should be noted

that due to the asymmetric noise growth, it is best to compute the product linearly, instead of using a logarithmic tree. I.e., it is best to keep a product "accumulator" and multiply all terms by this accumulator sequentially. This allows one to always choose the accumulator of the product as the operand $C_{N\times 2}$ in $BD(C_{N\times 2}) \times D_{N\times 2}$, leading to a slower growth of the *error* term. Moreover, when one wants to perform a search over a set of encrypted words, one can apply Algorithm 3 to each of these words, and afterwards add the results to get the encrypted value of the number of matches. The server could then transfer this result back to the client, without ever having access to the amount of matches. The client could afterwards decipher the result.

Algorithm 3. Matching a plaintext word with an encrypted word

Require: Encrypted bits c_i of the hash of $word_1$
Require: Plaintext $word_2$
Ensure: Cryptogram *match* encrypts 1 if there was a match, and 0 otherwise
  $match = \text{Encrypt}(1)$
  $a = \text{Hash}(word_2)$
  for all bit a_i **in** a **do**
    if $a_i = 1$ **then**
      $d_i = c_i$
    else
      $d_i = \text{Encrypt}(1) - c_i$
    end if
    $match = BD(match) \times d_i$
  end for
  return *match*

In this work, only the more burdensome Algorithm 3 was parallelized and accelerated using Xeon Phis. The parallel implementation of this algorithm will be explained in detail in the following section.

3 Parallel Algorithms

The targeted system provided 4 Xeon Phi Knights Corner coprocessors [5]. Each coprocessor features 61 cores operating at 1.053 GHz, interconnected via a 512-bit bidirectional ring, as shown in Fig. 2. Since the Xeon Phi coprocessor runs an Operating System (OS) inside, one of the cores will typically be dedicated to answering hardware/software requests like interrupts. As such, there are 60 usable cores, each supporting four-way hyperthreading, and thus 240 hardware threads are available. The cores can run at turbo modes, increasing the frequency of operation, if the power envelope allows.

Each core has two 32 kB L1 individual caches, for data and instructions, and a 512 kB L2 cache. The L2 caches are kept fully coherent by a global-distributed tag directory. The performance of the architecture is boosted with the vector

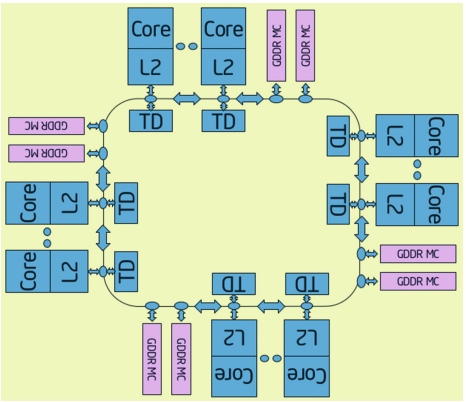

Fig. 2. Knights Corner schematic [11]

processing unit, which enables the processing of 512-bit registers. Each of the Xeon Phi cores has 32×512-bit SIMD registers. At each clock cycle, up to two instructions of a single thread are executed at each core. However, the two instructions follow two architecturally different pipelines, and therefore only one vector instruction can be executed at each cycle. The hardware cannot issue instructions back to back from the same thread in the core, and therefore at least two threads are necessary to reach full utilization of a core. Running 3 or 4 threads allows to hide more periods of latency, such as wrong instruction prefetches.

Since 512-bit SIMD instructions are available, 16 coefficients of a polynomial $f(x)$ can be processed in parallel, as each coefficient was represented with 32 bits. When performing reductions after additions or subtractions, comparison with q or 0 was implemented with the instruction `vpcmpud`, which produces a mask that indicates which lanes are greater or equal than q, or less than 0. This mask was used to prefix operations `vpsubd` and `vpaddd` that respectively subtract or add q to the lanes of the source register whose corresponding mask bit is 1. Furthermore, the repeated application of congruence (6) to reduce modular multiplications, as shown in Algorithm 1 was implemented with SIMD after unrolling the loop for when z initially had the value of $z = (q-1)^2$, so as to avoid divergent code on parallel operations.

Addition of polynomials in R_q was implemented by adding the corresponding polynomial coefficients over $\mathbb{Z}/(q\mathbb{Z})$ with SIMD instructions. Polynomial multiplications were implemented using changes of variable and FFTs. FFTs are decomposed into epochs, the number of which depends on the used radix r. In particular, each FFT consists of $\log_r n$ epochs, and in each epoch n/r computations, denominated butterflies, are performed. Using higher radices allows one to improve data locality. For the considered parameters, a radix-4 FFT was implemented, since $1024 = 4^5$. It is not possible to use higher radices (except for radix-1024, which is prohibitively large), since one cannot write 1024 as a

power of a larger integer. Moreover, 16 butterflies were processed in parallel using SIMD extensions to speed up the FFT computation.

The homomorphic multiplication $E_{N\times 2} = BD(C_{N\times 2}) \times D_{N\times 2}$ proceeded in three steps:

- In step (i), multiple threads processed the $D_{N\times 2}$ matrix. It consists on changing the variable of the polynomials from x to $\dot{x}$, and afterwards applying the FFT, producing a new matrix $\hat{U}_{N\times 2} = FFT(CONV(C_{N\times 2}))$ (where $CONV$ denotes the change in variable).
- In step (ii), the matrix multiplication operation was processed in blocks. Each of the 240 threads in a Xeon Phi coprocessor was given an identifier-pair (id_x, id_y), with $id_x \in \{0,\ldots,15\}$ and $id_y \in \{0,\ldots,14\}$. By denoting $\hat{V}_{N\times N} = FFT(CONV(BD(C_{N\times 2})))$, $x_i = \left\lfloor \frac{id_x \times N}{16} \right\rfloor$, $x_f = \left\lfloor \frac{(id_x+1)\times N}{16} \right\rfloor$, $y_i = \left\lfloor \frac{id_y \times N}{15} \right\rfloor$, and $y_f = \left\lfloor \frac{(id_y+1)\times N}{15} \right\rfloor$, each thread performed the following operation:

$$\hat{W}^{(id_x,id_y)} = \begin{pmatrix} \hat{v}_{y_i,x_i} & \cdots & \hat{v}_{y_i,x_f-1} \\ \vdots & \ddots & \vdots \\ \hat{v}_{y_f-1,x_i} & \cdots & \hat{v}_{y_f-1,x_f-1} \end{pmatrix} \times \begin{pmatrix} \hat{u}_{x_i,0} & \hat{u}_{x_i,1} \\ \cdots & \cdots \\ \hat{u}_{x_f-1,0} & \hat{u}_{x_f-1,1} \end{pmatrix} \tag{8}$$

  where the values of $\hat{v}_{i,j}$ are produced from the matrix $C_{N\times 2}$ as they are needed so as to reduce the memory requisites. Then, Algorithm 4 is used to add the blocks $\hat{W}^{(id_x,id_y)}$ with equal id_y to produce the matrix $\hat{W}_{N\times 2}$. In particular, this algorithm implements a logarithmic tree structure to add the intermediary results of the matrix: the base of the tree contains all intermediary values before running the algorithm, and the levels of the tree are processed from the base to the root. Each level corresponds to a time instant where the nodes are processed in parallel. In each of these nodes the values from its children are added, and therefore after the root is reached all intermediary results have been accumulated.
- In step (iii), an Inverse FFT (IFFT) is applied to the polynomials in $\hat{W}$, and $W = IFFT(\hat{W})$ is converted to $E_{N\times 2} = BD(C_{N\times 2}) \times D_{N\times 2}$, by changing the variable from $\dot{x}$ to x. Thread parallelism was used to process multiple polynomials simultaneously.

Since steps (i) and (iii) did not fully utilize the computational power of the Xeon Phi coprocessor, because there was not enough parallelism, several matrices were processed in parallel in these steps; which corresponds to searching on several words in parallel. Therefore, to compute s matrix multiplications, step (i) is first applied to s matrices in parallel, then step (ii) is repeated s times (once for each matrix multiplication), and finally step (iii) is applied s times in parallel to get the s results.

A modified version of Algorithm 3 was then implemented on the Xeon Phi. This modification corresponds to the comparison of the plaintext word with s encrypted words simultaneously, using the proposed matrix multiplication algorithm. Furthermore, the subtraction featured in the algorithm was accelerated

Algorithm 4. Logarithmic addition tree

Require: $\hat{W}^{(id_x,id_y)}, \forall id_x \in \{0,\ldots,15\}, \forall id_y \in \{0,\ldots,14\}$
Ensure: $\hat{W} = (\sum_{id_x} \hat{W}^{(id_x,0)}, \ldots, \sum_{id_x} \hat{W}^{(id_x,14)})$
  $\hat{W}_{N\times 2} = 0$
  The following code, until the return statement, is executed by all threads
  for $hop = 2$; $hop \leq 16$; $hop = hop \times 2$ **do**
    Thread barrier
    if id_x **is a multiple of** hop **then**
      if $hop = 16$ **then**
        for $i = y_i$; $i < y_f$; $i = i + 1$ **do**
          $\hat{W}_{i,0} = \hat{W}_{i-y_i,0}^{(id_x,id_y)} + \hat{W}_{i-y_i,0}^{(id_x+hop/2,id_y)}$
          $\hat{W}_{i,1} = \hat{W}_{i-y_i,1}^{(id_x,id_y)} + \hat{W}_{i-y_i,1}^{(id_x+hop/2,id_y)}$
        end for
      else
        $\hat{W}^{(id_x,id_y)} = \hat{W}^{(id_x,id_y)} + \hat{W}^{(id_x+hop/2,id_y)}$
      end if
    end if
  end for
  Thread barrier
  return $\hat{W}$

using multi-threading and SIMD extensions. When using a system with k Xeon Phis, the modified algorithm can be processed k times in parallel, and ks matches are homomorphically tested at the same time. Thus, when performing a search of a plaintext word over a set of encrypted words, the set was broken into smaller sets of ks encrypted words, and the Xeon Phis processed the sets in sequence. Afterwards, the host Central Processing Unit (CPU) adds the encrypted matches to create a cryptogram that when decrypted indicates how many encrypted words are equal to the plaintext.

4 Experimental Results

The proposed parallel algorithm was implemented on a system with an Intel Xeon CPU E5-2630 v2, operating at a frequency of 2.6 GHz, connected to 4 Intel Xeon Phi coprocessors 5110P, running at 1.053 GHz. The code was compiled with `icc` 16.0.1, using the optimization flag $-O2$. Computation was offloaded to the Xeon Phi coprocessors through `icc` pragmas, and the code on the Xeon Phi coprocessors was parallelized with OpenMP and SIMD intrinsics.

In order to find the optimal value for s (the number of matrix multiplications processed at a time by each Xeon Phi coprocessor), the homomorphic word searching algorithm was run on the 4 Xeon Phi coprocessors and timed for different values of s. In particular, a plaintext word was compared with sets of over 90 words for $s \in \{2, 4, 6, 8, 10, 12\}$. The relative execution time per encrypted word of the code offloaded to the Xeon Phi coprocessors can be found in Fig. 3. One can see that the relative search time per word decreases from 0.154 s for

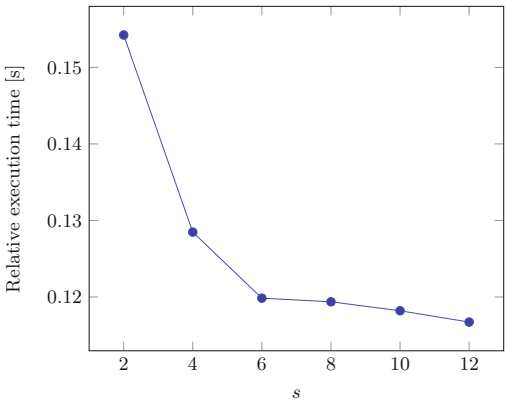

Fig. 3. Relative execution time per encrypted word as a function of the number of matrix multiplications (s) processed at a time by each Xeon Phi coprocessor

Table 1. Total execution time for homomorphic keyword searching

Number of encrypted words	Sequential execution [s]	Parallel execution [s]	Speedup
24	2662.6	3.6	743.9
48	5585.6	6.0	933.2
72	8418.6	10.5	803.2
96	11735	13.7	856.0

$s = 2$ until 0.119 s for $s = 6$, and stabilizes around that value for larger values of s. Therefore, the value of $s = 6$ was chosen for evaluating the performance of the parallel algorithms and obtaining the results presented next.

A sequential baseline version of Algorithm 3 was also implemented on a single core of the Xeon processor. Both the parallel implementation running on the Xeon Phi coprocessors and the sequential version running on the Xeon core were executed to perform a word search over sets of 24, 48, 72 and 96 encrypted words. The execution times of the word searching algorithm can be found in Table 1. There is a significant improvement in performance when executing the algorithm on the Xeon Phi coprocessors. In particular, an average speedup of 834 was obtained. By computing the obtained efficiency as the ratio between the speed-up and the product of the number of hardware threads and the number of SIMD lanes, and by taking into account the different frequencies of the Xeon processor and the Xeon Phi accelerator, one gets a value of 13.3%. This is accounted for by the fact that the vector unit is not exploited by all the instructions; that overhead is introduced when exploiting SIMD parallelism – for instance, in Algorithm 1 the while loop is unrolled for the worst case when exploiting SIMD extensions; and that there is also overhead associated with multi-threading parallelism – for

example, when Algorithm 4 is executed, the underlying cache-coherence protocol introduces delay in every iteration, so that data sharing is possible.

Finally, the Intel Xeon CPU E5-2630 v2 processor supports up to 12 hardware threads, with SIMD instructions of 8×32-bits lanes. Hence, even if 100% efficiency were obtained, with the parallelization of the sequential Xeon homomorphic word searching operation, one would require a cluster 18 processors to beat the performance of the 4 Xeon Phi coprocessors.

5 Related Work

There has been work in the literature on how to port scientific applications to the Xeon Phi, such as the lattice Boltzmann code [13], the Monte Carlo tree search [9], the Rodinia benchmark [10] and sparse matrix multiplications [14] with very satisfactory performance. Common concerns among these works include the division of the work-load in a balanced way among the large amount of threads of the Xeon Phi coprocessor, an effective vectorization of code, and also of how to best distribute data in memory. While some works focus on the computation of the FFT [7,8], they are supported on the complex plane instead of finite fields, since they target telecommunications protocols, and hence their performance is not directly comparable with the FFT implementation presented in this work. Furthermore, long integer operations are optimized in [1] for the Xeon Phi coprocessor, with a special focus on vectorization. These operations are used to implement the Rivest-Shamir-Adleman (RSA) cryptosystem [12]. While textbook RSA is homomorphically multiplicative, since the multiplication of two cryptograms results in an encryption of the multiplication of the underlying plaintexts, textbook RSA is not considered safe, and this feature is not exploited in [1].

The described cryptosystem was supported on an earlier scheme [4], also based on matrix operations. However, the latter did not exploit ring arithmetic, which arguably degrades performance, and hence, as far as we know, there are no practical implementations. The cryptosystem proposed in [6] was also implemented therein using an Intel Core-i7 5930 K and a NVIDIA GeForce GTX980 as an accelerator. An homomorphic word searching procedure took a relative time of about 20 ms per encrypted word. This result cannot be directly compared with the results obtained herein (see Fig. 3) since the main objective of this work was to evaluate the improvement of performance one could get with widely deployed programming tools, such as OpenMP, that are available on the Xeon Phis, and provide more manageable code development. The GTX980 GPU is organized according to a different architecture, and targets a more strict range of applications, which does not allow a direct comparison with the results obtained for the Xeon Phi. Furthermore, the GTX980 GPU has 2048 CUDA cores, whereas the 4 Xeon Phis feature a total of 960 hardware threads, and therefore it is possible to exploit a larger level of parallelism with the GTX980. Also, the GTX980 runs at a slightly higher frequency (1.126 GHz) than the Xeon Phis (1.053 Ghz).

6 Conclusion

A large amount of embedded systems are currently being deployed in the market, either in the form of consumer electronics or household appliances and in the general infrastructure. Considering that they have limited computational resources, more and more computation will start being offloaded to central servers. Since this data may be private, it is expected that homomorphic encryption will become increasingly important, because it allows for the processing of encrypted data. In this work, the performance of homomorphic encryption is significantly enhanced with the use of Xeon Phi coprocessors. This enhancement is achieved by exploiting the fact that the considered cryptosystem relies on matrix multiplication over a specific ring, which is a burdensome operation with a large level of parallelism. In concrete, a speedup of about 834 was obtained for an homomorphic word searching procedure.

Furthermore, the Xeon Phi architecture has several advantages when compared with other HPC systems. It is more flexible than GPU architectures, supporting parallel divergent code more efficiently. It provides more manageable and less time consuming tools for code development than FPGAs. Finally, it supports multiple programming paradigms (such as OpenMP, and MPI) that are widely deployed, and for which large codebases already exist.

References

1. Chang, C., Yao, S., Yu, D.: Vectorized big integer operations for cryptosystems on the Intel MIC architecture. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), pp. 194–203, December 2015
2. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University, Stanford, CA, USA (2009)
3. Gentry, C.: Computing arbitrary functions of encrypted data. Commun. ACM **53**(3), 97–105 (2010). http://doi.acm.org/10.1145/1666420.1666444
4. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. Cryptology ePrint Archive, Report 2013/340 (2013). http://eprint.iacr.org/2013/340
5. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2013)
6. Khedr, A., Gulak, G., Vaikuntanathan, V.: Shield: scalable homomorphic implementation of encrypted data-classifiers. Cryptology ePrint Archive, Report 2014/838 (2014). http://eprint.iacr.org/
7. Khelifi, M., Massicotte, D., Savaria, Y.: Parallel independent FFT implementation on intel processors and Xeon phi for LTE and OFDM systems. In: Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC), pp. 1–4, October 2015
8. Khelifi, M., Massicotte, D., Savaria, Y.: Towards efficient and concurrent FFTs implementation on Intel Xeon/MIC clusters for LTE and HPC. In: 2016 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2611–2614, May 2016

9. Mirsoleimani, S.A., Plaat, A., Van Den Herik, J., Vermaseren, J.: Scaling Monte Carlo tree search on Intel Xeon phi. In: 21st International Conference on Parallel and Distributed Systems (ICPADS), 2015, pp. 666–673, December 2015
10. Misra, G., Kurkure, N., Das, A., Valmiki, M., Das, S., Gupta, A.: Evaluation of rodinia codes on Intel Xeon Phi. In: 2013 4th International Conference on Intelligent Systems, Modelling and Simulation, pp. 415–419, January 2013
11. Reinders, J.: An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors, November 2012. https://software.intel.com/en-us/mic-developer, Intel Corporation
12. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978). http://doi.acm.org/10.1145/359340.359342
13. Rosales, C.: Porting to the Intel Xeon Phi: opportunities and challenges. In: 2013 Extreme Scaling Workshop (XSW 2013), pp. 1–7, August 2013
14. Saule, E., Kaya, K., Çatalyürek, Ü.V.: Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013. LNCS, vol. 8384, pp. 559–570. Springer, Heidelberg (2014). doi:10.1007/978-3-642-55224-3_52