HPC on the Intel Xeon Phi: Homomorphic Word Searching

Paulo Martins [**] and Leonel Sousa

Instituto Superior Técnico, Universidade de Lisboa
INESC-ID
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

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 [1]. 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.

With HE, companies are capable of outsourcing the processing of sensitive data while ensuring the privacy of the data. For instance, it is possible to delegate the statistical analysis of sensitive data collected by sensors to an untrusted third party. 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

[**] 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.

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 Xeon Phi coprocessor featuring a large amount of cores [2]. These cores are very power efficient due to short pipelines and low frequency operations. Also, they enable the use of many of the programming models that most developers are already accustomed to, such as OpenMP, and Message Parsing Interface (MPI). 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 Section 2, we give an introduction to HE, and describe how it can be applied to perform word searching. Afterwards, in Section 3, procedures and algorithms are proposed for the Intel Xeon Phi architecture. The performance of the proposed parallel algorithms is evaluated in Section 4, and finally conclusions are drawn in Section 5.

2 Homomorphic Encryption

The dichotomy of either keeping data encrypted or being processable was overcome in 2009, by the uncovering of Fully Homomorphic Encryption (FHE) [1]. 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 [3] 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. Roughly, elements $f(x), g(x) \in R_q$ are represented as polynomials, and after each addition, subtraction or multiplication, one computes the remainder of the division of the polynomial by $\Phi(x)$, and of the remainder of the polynomial coefficients by q [5], e.g. $u(x) = [f(x) \pm g(x)]_q$ and $v(x) = [f(x) \times g(x) \bmod \Phi_n(x)]_q$. We designate the processes of computing the remainders of the division "reduction".

In this cryptosystem, the public-key corresponds to a matrix $A \in R_q^{1\times 2}$ and the private-key is a matrix $s \in R_q^{2\times 1}$ such that

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

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*:

$$C_{N\times 2} \times s_{2\times 1} = \mu \begin{pmatrix} 1\ 2 \dots 2^{l-1}\ 0\ 0 \dots\ \ 0 \\ 0\ 0 \dots\ \ 0\ \ 1\ 2 \dots 2^{l-1} \end{pmatrix}^T \times s_{2\times 1} + error \tag{2}$$

If we add two cryptograms $C_{N\times 2}$ and $D_{N\times 2}$, the resulting cryptogram retains the format described in equation 2. 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, and produces an $N \times N$ matrix. 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. The *error* term grows as homomorphic operations are performed. The threshold for the size of *error* dictates how many operations can be performed in the encrypted domain. If this threshold is reached, decryption is no longer possible.

The parameters for the cryptosystem described in [3] enable efficient arithmetic over R_q. In particular, the value of n was set to $n = 1024$ and q to $q =$ `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): $2^{31} \equiv 2^{17} - 1 \bmod q$. 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:

$$z \equiv z_1 2^{17} + z_0 - z_1 \tag{3}$$

This latter congruence is iteratively employed 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.

Multiplications modulo $\Phi_n(x)$ over $\mathbb{Z}/(q\mathbb{Z})$ can be implemented by applying the following processing steps [4]: *i*) performing a change of variable of the polynomials, *ii*) applying a Fast Fourier Transform (FFT) over $\mathbb{Z}/(q\mathbb{Z})$ to both of the multiplication operands, *iii*) multiplying the coefficients of the transforms, *iv*) performing an Inverse FFT (IFFT) of the result, *v*) and finally reverting the change of variable. In particular, if $\eta^n \equiv -1(\mathrm{mod} q)$, and the change of variable $\dot{x} = \eta x$ is performed, then $\dot{x}^n - 1 = (\eta x)^n - 1 \equiv -(x^n + 1) \equiv 0(\bmod x^n + 1)$, and operations modulo $\dot{x}^n - 1$ will be equivalent to modulo $x^n + 1$ after reverting the change in variable. Furthermore, multiplication of the coefficients of the FFTs will produce the FFT of the convolution of the coefficients of the transformed polynomials. This convolution is equivalent to performing a multiplication modulo $\dot{x}^n - 1$.

2.1 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 1 to the set of words in the e-mail. It should be noted that since

Algorithm 1 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
    $encrypted\_bit\_list = \{\}$, $a = \text{Hash}(word)$
    for all bit $a_i$ in $a$ do
      $c_i = \text{Encrypt}(a_i)$, $encrypted\_bit\_list = encrypted\_bit\_list \cup \{c_i\}$
    end for
    $output\_list = output\_list \cup \{encrypted\_bit\_list\}$
  end for
  return $output\_list$
```

a hash function is used to conceal the word 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.

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 2 to each of them and the plaintext queried word. This algorithm computes a sequence of homomorphic multiplications that ultimately result on a cryptogram ciphering 1 if the two words are the same or 0 otherwise. Afterwards, these cryptograms are added 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. In this work, only the more burdensome Algorithm 2 was parallelized and accelerated using Xeon Phis. The parallel implementation of this algorithm will be explained in detail in the following section.

Algorithm 2 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 $match$
```

3 Parallel Algorithms

The targeted system provided 4 Xeon Phi Knights Corner coprocessors [2]. Each coprocessor features 60 cores operating at 1.053 GHz, interconnected via a 512-bit bidirectional ring. Each core has two 32kB L1 individual caches, for data and instructions, and a 512kB L2 cache. The L2 caches are kept fully coherent by a global-distributed tag directory. A core can hold up to 4 hardware threads at any time, and at each clock cycle, up to two instructions of a single thread are executed. However, the two instructions follow two architecturally different pipelines, and therefore only one vector instruction can be executed at each cycle. The performance of the architecture is boosted with a vector processing unit, which enables the processing of 512-bit registers.

Since 512-bit Single Instruction Multiple Data (SIMD) instructions are available, 8 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 (3) to reduce modular multiplications 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. Using higher radices allows one to improve data locality. For the considered parameters, a radix-4 FFT was implemented. Each FFT consists of $\log_r n$ epochs, and in each epoch n/r computations, denominated butterflies, are performed. Using SIMD extensions, it was possible to process 8 butterflies in parallel.

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 was processed in blocks. Each of the 240 threads in a Xeon Phi was given an identifier (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:

$$\hat{W}^{(id_x,id_y)} = \begin{pmatrix} \hat{v}_{y_i,x_i} & \cdots & \hat{v}_{y_i,x_f-1} \\ \vdots & \ddots & \vdots \\ \hat{v}_{y_f-1,x_i} & \cdots & \hat{v}_{y_f-1,x_f-1} \end{pmatrix} \times \begin{pmatrix} \hat{u}_{x_i,0} & \hat{u}_{x_i,1} \\ \cdots & \cdots \\ \hat{u}_{x_f-1,0} & \hat{u}_{x_f-1,1} \end{pmatrix} \qquad (4)$$

Then, Algorithm 3 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.

Algorithm 3 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
          $\hat{W}_{i,0} = \hat{W}^{(id_x,id_y)}_{i-y_i,0} + \hat{W}^{(id_x+hop/2,id_y)}_{i-y_i,0}, \hat{W}_{i,1} = \hat{W}^{(id_x,id_y)}_{i-y_i,1} + \hat{W}^{(id_x+hop/2,id_y)}_{i-y_i,1}$
        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 $\hat{W}$
```

In step $iii)$, an 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 2 was then implemented on the Xeon Phi coprocessor: a plaintext word is compared with s encrypted words simultaneously, using the proposed matrix multiplication algorithm. 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.

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.

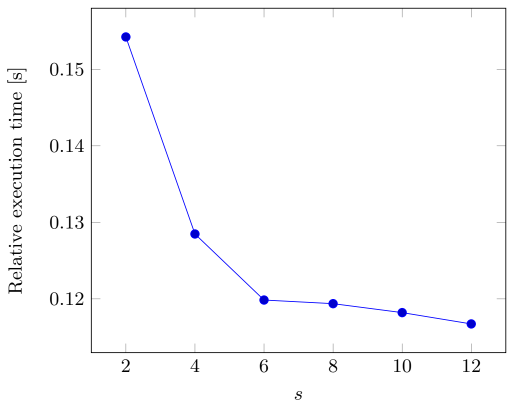

Fig. 1. Relative execution time per encrypted word as a function of the number of matrix multiplications (s) processed at a time by each Xeon Phi coprocessor

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 Figure 1. One can see that the relative search time per word decreases from 0.154 s for $s = 2$ 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.

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

Table 1. Total execution time for homomorphic keyword searching

A sequential baseline version of Algorithm 2 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.

The described cryptosystem was also implemented in [3] using an Intel Core-i7 5930K 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 Figure 1) 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. Furthermore, 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.

5 Conclusion

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. It should be noted that the Xeon Phi code development paradigm is more flexible than other solutions such as GPUs, fitting a wider range of applications, and therefore may be more suitable to server settings with heterogeneous workloads.

References

1. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford University, Stanford, CA, USA (2009)
2. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2013)
3. 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/
4. Pöppelmann, T., Güneysu, T.: Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In: Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings. pp. 139–158 (2012)
5. Wang, X., Xu, G., Wang, M., Meng, X.: Mathematical Foundations of Public Key Cryptography, vol. 1. CRC Press, Boca Raton, Florida, USA, 1 edn. (10 2015)