2014 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)

Investigating Large Integer Arithmetic on Intel Xeon Phi SIMD Extensions

Anastasis Keliris
Electrical and Computer Engineering
New York University Abu Dhabi
E-mail: anastasis.keliris@nyu.edu

Michail Maniatakos
Electrical and Computer Engineering
New York University Abu Dhabi
E-mail: michail.maniatakos@nyu.edu

***Abstract*—In this paper, we investigate the Single Instruction Multiple Data (SIMD) extensions of the Intel Xeon Phi, and we analyze the benefits and limitations of its unique architecture in large integer arithmetic applications. The first generation Intel Xeon Phi, named *Knights Corner*, is a state-of-the-art many core coprocessor with up to 61 cores. Each Knights Corner core has 512-bit SIMD vectors, which are supported by a unique instruction set architecture, incompatible with existing SIMD schemes (SSE, AVX etc.). The second generation Intel Xeon Phi, *Knights Landing*, will include the standard 512-bit AVX512 SIMD extensions. We implement a multiplication scheme for large integers, which leverages the aforementioned SIMD extensions. Using this multiplication scheme, we discuss the SIMD extensions' performance for both Intel Xeon Phi generations. Preliminary results indicate that the Knights Corner SIMD speedup of large integer multiplication is limited by the absence of specific instructions that typically appear in common SIMD architectures. The emulation on Knights Landing (which includes backwards-compatible AVX512 extensions), on the contrary, shows that large integer multiplication can indeed benefit by the presence of 512-bit vectors, for commonly used 1024- and 2048-bit operands, compared to publicly available large arithmetic libraries.**

***Index Terms*—large integer multiplication, Intel Xeon Phi, SIMD, AVX**

I. Introduction

Secure and reliable information communication is essential in multifarious domains, such as financial transactions, private sector services and government and military applications [1]. The information itself, as well as information exchange, can be secured with the use of cryptography-based information security schemes, ensuring authentication, integrity, confidentiality and non-repudiation [2]. As performance is a requirement of prime importance for practical applicability of these schemes, the underlying cryptographic primitives which enable information security must be as efficient as possible, enabling seamless integration of protection schemes with applications [3].

Since the introduction of public-key cryptography [4], there is a growing interest in efficient implementations of large integer arithmetic. Modular multiplication and modular exponentiation are typical examples of computationally intensive building blocks, appearing in widely used public-key cryptographic schemes, such as RSA [5], El-Gamal [6] and Elliptic Curve Cryptography (ECC) [7]. The performance of applications that use public-key cryptography is therefore strongly linked with the efficiency of modular multiplication algorithms. In addition, as public-key cryptography key sizes increase for ensuring higher cryptographic security [8], the need for efficient, optimized implementations of large integer arithmetic is of immediate practical significance.

Most software implementations of large integer multiplication up to date, are based on scalar instructions [9]. In this paper, we aim to leverage the wider vectors offered by the Intel Xeon Phi Single Instruction Multiple Data (SIMD) instruction set extensions, to efficiently perform large integer multiplication. SIMD instructions perform the same operation on multiple data points simultaneously and are present on many modern computer architectures [10]. CPU's that support SIMD extensions include the x86 platform with Intel's SSE, AVX and AVX2 extensions [11] and ARM with the Neon extensions [12].

In this paper we target the Intel Xeon Phi, because of its wider, 512-bit SIMD vectors. Specifically, we investigate two Xeon Phi generations: i) *Knights Corner (KNC)*, the first Many Integrated Core Architecture (MIC) platform by Intel. It has up to 61 cores and is mainly used in high performance computing. Each core consists of a scalar and a vector unit [13], with the latter offering 512-bit wide SIMD registers. KNC includes a unique SIMD extension set, incompatible with existing SIMD schemes. ii) *Knights Landing (KNL)*, the second generation MIC, which will include 512-bit wide SIMD vectors supported by the AVX512 extensions [14].

The main contributions of this paper are:

- An analysis of the benefits and limitations offered by the unique SIMD architecture of KNC.
- Performance evaluation of large integer arithmetic operations on 512-bit wide SIMD vectors, using the unique SIMD instruction set architecture (ISA) for KNC and the AVX512 extensions for KNL.

The remainder of the paper is organized as follows: In the following section we discuss related work on software implementations of large integer and modular arithmetic, as well as preliminary studies on the Intel Xeon Phi. Section III provides an overview of the Intel Xeon Phi's architecture. The methodology used for implementing vector-based multiplication is described in Section IV, while experimental setup and implementation challenges are presented in Section V. Section VI presents the experimental results, followed by conclusions and future directions in Section VII.

II. Related Work

The use of SIMD instructions to accelerate large integer operations is currently in the spotlight, due to the recent wider

978-1-4799-4972-4/14/$31.00 ©2014 IEEE

SIMD architectures which enable simultaneous operations on larger data sets. Fast multiplication of large integers requires efficient carry propagation handling. In order to achieve this, the use of a redundant radix representation of large integers has been proposed [15] and has been implemented for state-of-the-art processors [16], [9], [17], [18].

In [19], Bernstein used the redundant radix representation to design and implement a specialized high-speed high-security elliptic curve: `curve25519`. This specialized curve was used for demonstration of significant speedups of high security cryptography, using ARM Cortex A8's NEON vector instruction set [17]. The vector extensions supported by the Cell Broadband Engine, also known as Synergistic Processor Elements, are used in [18] to perform fast *Montgomery Multiplication* using a radix-2^{16} representation. Montgomery multiplication [20] is a very efficient modular multiplication algorithm, that replaces division with a series of additions and divisions by a power of 2. The implementation achieved a speedup of up to 2.47X in comparison to a non-SIMD implementation on the same processor.

The SIMD extensions of Qualcomm's Snapdragon and Intel's Atom processors for modular arithmetic for ECC were studied in [16]. Different redundant radix representations, according to the elliptic curve under analysis, were used, and a comparison between the NEON and SSE2 vector instructions for modular multiplication was presented. The authors demonstrated speed-ups of more than 2X compared to the scalar modular multiplication algorithm used by the GNU Multiple Precision Arithmetic Library [21].

In [9], Gueron and Krasnov demonstrated that in order to outperform scalar implementations of modular multiplication using SIMD instructions and redundant radix representation, the architecture must support vector registers that are *at least 162 bits wide*. This number is derived from comparing the number of additions and multiplications required by the scalar and redundant radix representations for a radix-2^{29} representation. The AVX2 instruction set extension satisfies this requirement since it uses 256-bit wide vector registers. In addition, the authors investigated an efficient software implementation of modular exponentiation targeting the RSA cryptographic algorithm. A *Non-Reduced Montgomery Multiplication* was used in combination with a radix-2^{29} representation. This implementation used the AVX2 instruction set and achieved a 51% speedup for RSA2048 compared with the scalar non-vectorized implementation.

General purpose GPUs (GPGPU) have been also studied for large integer arithmetic and cryptography applications [22]. In [23], the author proposed an algorithm for Montgomery exponentiation that used a radix-2^{24} representation, to exploit the parallelism of GPUs. This work presents important speedup when multiple exponentiations are to be calculated at the same time, but shows worse performance for a single operation. GPUs suffer from the same limitations as SIMD approaches: GPUs are designed to handle floating-point data very efficiently but not integers. According to [22], poor overall performance is achieved when trying to use GPUs for symmetric encryption algorithms, due to lack of support for certain operations.

The authors of [24] explored the SIMD instruction set extensions of the first generation Intel Xeon Phi. The authors analyzed and compared the Intel SIMD architectures of Intel Xeon processors and the Intel Xeon Phi for single-precision floating point operations. The results showed that effective SIMD utilization leads to significant performance gains, up to 2X, compared to Intel Xeon processors that support legacy 256-bit AVX2 extensions. Although this work focuses on floating-point operations and not large integer arithmetic, it provides extensive information on considerations for using Intel's KNC unique instruction set architecture.

The KNC performance factors and limitations were extensively studied in [25] through micro-benchmarking. In particular, the authors measured the latency and throughput of the Vector Processing Unit. The results indicate that the latency is 4 and 6 cycles for vector arithmetic and vector permutation instructions respectively. The theoretical instruction throughput of 1 tera floating-point operations per second (TFLOPS) can be achieved in scenarios where 240 threads of the coprocessor are used to perform fused multiply-add instructions on floating-point operands. The authors also argue that great effort is required from programmers for achieving maximum performance gains. An extensive overview of the Intel Xeon Phi KNC is presented in [26]. This best practice guide also provides useful information about programming models used for achieving optimal performance, utilizing the unique architecture of the coprocessor.

III. Hardware Platform: The Intel Xeon Phi

The Intel Xeon Phi is the brand name for Intel's processor family of Many Integrated Core (MIC) architecture. The first generation Intel Xeon Phi is known as Knights Corner (KNC), while the second generation is code-named Knights Landing (KNL)[1]. In this section, we discuss i) the unique characteristics of KNC, which render it incompatible with previous Intel SIMD extensions, and ii) the published features of KNL and the AVX512 architecture.

A. Knights Corner

The KNC coprocessor offers up to 61 cores, each fetching and decoding instructions from four hardware threads. Each core is simplified, in the sense that their scalar unit is an in-order architecture based on the Intel Pentium processor family. The cores have been stripped out of expensive features, such as branch prediction and out-of-order execution, and operate at 1.1GHz. Each core of the KNC is based on the x86 instruction set architecture and provides support of 64-bit addressing [13].

The cores have 32KB 8-way associative L1 data cache and instruction cache and an interface with the Core Ring Interface (CRI)/L2 cache. Through the CRI, all the cores are connected to a bidirectional ring interconnect. 512-bit SIMD vectors are supported by the Vector Processing Unit. Each core

[1]Under development by Intel at the time of this publication.

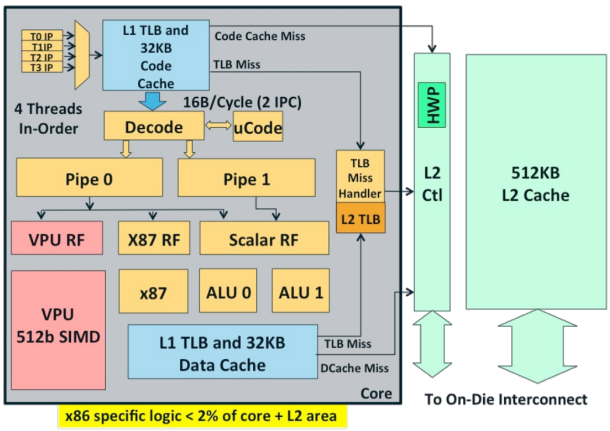

Fig. 1. Architectural overview of an Intel Xeon Phi core [27].

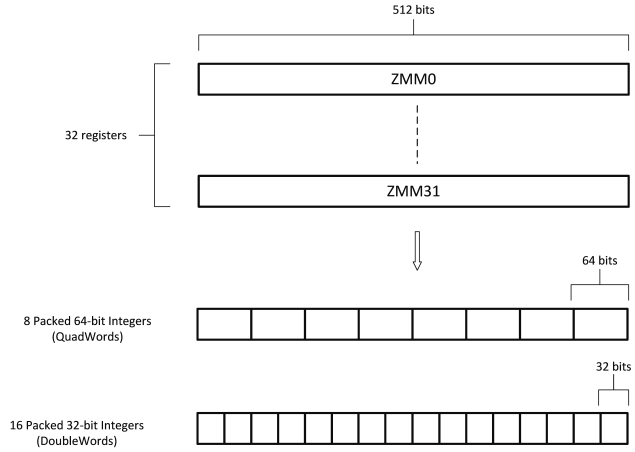

Fig. 2. Vector registers on KNC Intel Xeon Phi and supported packed integer data types [28].

can execute 2 instructions per cycle, one on the V-pipe and one on the U-pipe. Vector instructions can only be executed on the U-pipe. A high level architectural overview of a KNC core is shown in Figure 1. The U-pipe corresponds to Pipe 0 and the V-pipe to Pipe 1 in Figure 1.

The coprocessor connects to the host system via PCI-express. It can be accessed from the host as a network node, through a virtualized TCP/IP stack. A Linux micro operating system that runs on the coprocessor provides traditional UNIX tools and support for the main Intel developer tools. Two models of operation are supported: native and offloading. The first explicitly runs applications on the coprocessor, while the latter runs on the host and offloads sections of the code for execution on the coprocessor. We target the native compilation model, since we are mainly interested in evaluating the SIMD extensions performance.

The Vector Processing Unit: Each core of the KNC Intel Xeon Phi includes a Vector Processing Unit (VPU). The VPU includes 32 512-bit wide vector registers `zmm0 – zmm31` along with 8 16-bit wide mask registers. The vector registers are supported by a new 512-bit SIMD ISA [28] which is an intermediate between AVX2 and the upcoming AVX512. Previous SIMD extensions such as SSE4.1, SSE4.2 and AVX are not supported by the KNC. The packed native data types that are supported by the VPU are:

- *DoubleWords* - 32-bit integers
- *QuadWords* - 64-bit integers
- 32-bit single precision floating points
- 64-bit double precision floating points

The vector registers and the integer data types supported are shown in Figure 2. In this work, we focus on the packed 64-bit integers for efficiently performing the redundant radix multiplication. Data alignment is important for improving performance when using vector instructions [29]. For best performance on the Intel Xeon Phi, data must be 64-byte aligned, unlike previous SIMD extensions [25]. For comparison, AVX requires 32-byte alignment and SSE 16-byte data alignment.

B. Knights Landing

The second generation of Intel's MIC architecture, code named Knights Landing is expected in 2015. It will target the 14nm manufacturing process and support the AVX512 extensions [14]. One of the main differences with KNC is that it will be available both as a standalone many-core CPU (not requiring a separate Xeon host), and as a PCI-express coprocessor. The AVX512 extensions offer 512-bit wide registers and full compatibility with previous Intel SIMD extensions [30]. In this work, we have used the Intel Software Development Emulator to run our SIMD multiplication scheme and test the performance of the AVX512 extensions, as it is still under development and its final specifications are not published.

C. Important SIMD instructions for large integer arithmetic

The important vector instructions we use in our implementation, for both KNC and KNL, are briefly explained below:

VPSHUFD - KNC and KNL
: Permutes 32-bit blocks of an INT32 vector.

VPBROADCASTQ - KNC and KNL
: Copy a 64-bit memory operand to all elements of a vector register.

VPMULHUD - KNC
: Multiplies two UINT32 vectors and stores high result in destination vector register.

VPMULLD - KNC
: Multiplies two INT32 vectors and stores low result in destination vector.

VPMULUDQ - KNL
: Multiplies the low UINT32 parts from each packed 64-bit source vectors element, and stores the UINT64 result in destination vector.

IV. Multiplication Methodology

For the evaluation of the SIMD extensions of the Intel Xeon Phi, we implement a variation of the carry-less multiplication presented in [9]. The proposed multiplication uses redundant

Algorithm 1: Calculate $C = A \times B$ in radix 2^m repr.

Input: $A = \sum_{i=0}^{n-1} A_i \times 2^{m \times i}, B = \sum_{i=0}^{n-1} B_i \times 2^{m \times i}$
$n =$ number of digits in radix 2^m representation
Output: $C = A \times B = \sum_{i=0}^{2n-1} C_i \times 2^{m \times i}$

```
k ← 0
for i = 0 to n − 1 do
    for j = 0 to n − 1 do
        C_{i+j} ← C_{i+j} + A_i × B_j
    end for
    k ← k + 1
    if k ≥ 2^{64−2×radix} then
        cleanup
        k ← 0
    end if
end for
cleanup
```

radix representation in order to avoid carry propagation in each intermediate step. For comparison purposes, as shown in Algorithm 1, we modify the modular exponentiation algorithm originally demonstrated in [9], to support unsigned integer multiplication without carry propagation on radix 2^m operand digits.

A. Radix 2^m representation

An n-bit integer can be represented in any radix-2^m representation for $m > 1$. The number of digits required for conversion from a normal representation of an n-bit integer, to a radix-2^m representation is $ceil(n/m)$. An n-bit integer which is converted from radix-2^{64} to radix-2^m representation becomes $A = \sum_{i=0}^{k-1} A_i \times 2^{m \times i}$, where $0 \leq A_i < m$. This representation is unique. An extensive analysis of radix-2^m representations appears in [9].

An example of an 128-bit integer converted to radix-2^{29} is presented below. In this example, we can observe that 5 digits are required for the radix-2^{29} representation as expected, since $n = 128$, $m = 29$ and $ceil(128/29) = 5$. For brevity, we don't present the higher 32 bits of each packed 64-bit vector register element in the radix-2^{29} representation, as all 32 bits are equal to zero.

Example: Conversion of 128-bit integer to radix-2^{29}:

128-bit integer in regular representation:

$$cee4fccab563954d444db98ca87aadbd$$

Radix-2^{29} representation:

$$00000cee|09f9956a|18e55351|026dcc65|087aadbd$$

The immediate gain of this representation is that we can perform carry independent operations on the vector registers, *without* dealing with carry propagation after each intermediate result. In particular, we can avoid dealing with the carry until after the multiplication is completed, or after a specific number of accumulations. If we store the radix-2^m integers in 64-bit vectors, then m bits are required for every "digit". After a multiplication step of the algorithm, each packed UINT64 vector register has $2 \times m$ bits at most. This allows for $64 - 2 \times m$ bits slack for accumulations without overflow in each register, meaning we can perform $2^{64-2\times m}$ accumulations before having to perform correction to avoid overflow. This is the exact role of the variable k in Algorithm 1: At every step of the outer loop, the variable is incremented by one and a cleanup procedure is initiated only if the maximum number of permitted additions, according to m, is reached.

After multiplying two digits, the destination register will include $2 \times m$ significant bits. This representation is not a radix-2^m representation, but a *redundant-radix* representation. As the name implies, this representation is not unique. To convert a redundant radix representation to a radix-2^m representation, the m lower bits of a digit are kept and the $64 - m$ bits are carried and accumulated with the next most significant digit. This procedure is repeated from the least significant digit to the most significant digit. As one can easily observe, the cleanup procedure consists mostly of masking, shifting and permutation operations. Because shifting and permutation operations are costly on vectors and have a 6 cycle latency on the KNC [25], we try to perform the cleanup procedure as infrequently as possible.

V. Experimental Setup

For the investigation of Intel Xeon Phi SIMD extensions performance, we implement the algorithm of Section IV in C language with inline assembly and test it on Intel's Software Developer Emulator.

A. Multiplication algorithm specifics

The operand sizes that we study in this paper are 256, 512, 1024, 2048 and 4096 bits. The first three operand sizes are typical operand sizes used in cryptography. We also include 2048 and 4096 bit operands in our analysis, since future extensions of cryptographic algorithms, with higher security requirements, will require increased key lengths.

The large integers we use as operands are generated with a pseudo-random number generator. The *seed* we use for this generator is a function of the host system's time. We choose pseudo-random integers because we aim at a generic model for multiplication and not a model that relies on integers with specific characteristics. We manually inserted trivial test cases, such as maximum and minimum number supported, for correctness proof of our implementation.

We select $m = 29$ for the radix-2^m representation in our implementation, similarly to the implementation of [9]. By using a radix-2^{29} representation, the result of multiplication of two 29-bit words will be at most 58-bits, leaving 6 bits available for carry propagation. We can thus perform $2^{64-58} = 64$ operations of the outer loop (accumulations) before having to deal with potential overflow. We explored the use of other

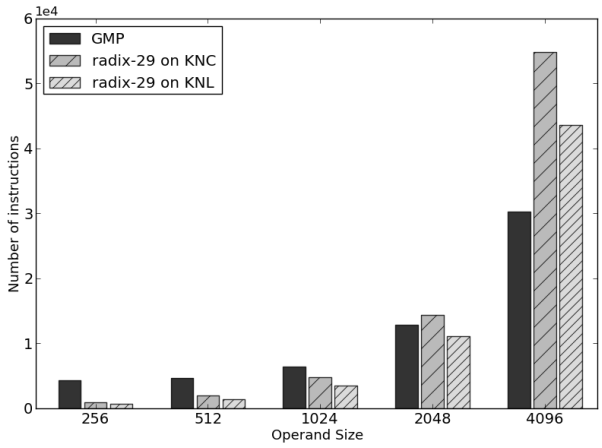

Fig. 3. Instruction count comparison for multiplication on KNC and KNL using GMP and the radix-2^{29} implementation.

radices, in particular $m = 28$ and $m = 30$. For smaller radices, each integer requires more digits and thus more multiplications and accumulations, while for larger radices the cleanup procedure has to be performed more frequently. Our study indicated that the best performance, for the operand sizes we study, was achieved when $m = 29$ confirming the results of [9].

B. Platform details

We perform a comparison with the GNU Multiple Precision Arithmetic Library (GMP) version 5.1.3 [21] for evaluating the performance of Intel Xeon Phi SIMD extensions. The host machine we use for cross-compiling the executable, is an Intel Xeon E5-2609 CPU operating at 2.4GHz with 64GB RAM and runs CentOS 6.5 operating system. The compiler we use is the Intel C compiler `icc` version 14.0.2. Our implementation is cross-compiled for the KNC and the KNL architecture for native execution, with use of the appropriate flags in `icc`. We emulate a KNL core with AVX512 extension support, using Intel Software Development Emulator (SDE) version 6.22 and measure the performance in instructions executed by analyzing the instruction trace of the execution.

C. Implementation challenges

Since we are emulating execution, the accuracy of the performance assessment is limited, since system specifics are not available. For example, we do not have information about behavior of mechanisms such as caches, pipeline and forwarding. Therefore, in this study, we measure in a deterministic way the total instructions executed with the help of Intel's Software Developer Emulator. We then compare instructions required for a multiplication using the GMP method and the same multiplication using the radix-2^{29} method. Future work will include validation of our experiments on the actual Intel Xeon Phi hardware.

VI. Experimental Results

In Figure 3 we present the instruction count comparison of our large integer multiplication implementation and the GMP method. The instruction count for a KNL core with AVX512 support and a KNC core with VPU support is compared with the instruction count GMP achieves for the same multiplication. In our multiplication implementation, instructions are executed both on the scalar unit and the vector unit of the coprocessor, while GMP only uses the scalar unit of the processors.

The results of our experiments are summarized in Table I. We can observe from Table I and Figure 3 that for smaller operand sizes, up to 2048 bits for KNL and up to 1024 bits for KNC, the radix-2^{29} method outperforms the GMP method. For the typical operand size of 1024 bits, the speedup of the radix-2^{29} algorithm is significant: With our implementation, we achieve a 1.85X speedup on KNL and 1.36 on KNC. A reason for the increased instructions required by GMP to perform multiplication on the smaller operands, is the fact that GMP has to perform internal calls to the library's functions. This adds an overhead that is not negligible for the smaller operand sizes. When larger operands are multiplied, this overhead becomes insignificant.

Multiplication with GMP outperforms our implementation for the large operand sizes of 2048 and 4096 bits. There are three reasons for that: i) First, as operand size increases, more digits are required for the radix-2^{29} representation, and therefore more steps of Algorithm 1 have to be performed. ii) For large operands, GMP uses more advanced algorithms to perform the multiplication, such as the Karatsuba algorithm [31] and the Toom-Cook algorithm [32], which result in better performance. iii) Finally, we note that for the two larger operand sizes, 2048-bits and 4096-bits, the cleanup procedure in our multiplication implementation, is invoked 1 and 2 times respectively, when selecting $m = 29$. This happens because for 2048 and 4096 bit operands, 71 and 142 digits are required respectively, for representing each operand. Cleanup is required since the accumulations exceed the $2^{64-58} = 64$ threshold, adding an additional performance penalty.

The KNC coprocessor is always worse in comparison to KNL. This happens because KNC does not provide support for existing SIMD extensions, incorporating, instead, a fundamentally different SIMD ISA (extensively discussed in Section III). Specifically, an important integer multiplication instruction – `PMULUDQ` – is not supported. The KNC uses the `VPMULLD` and `VPMULHUD` instructions, in its place, generating the lowest and highest part of the result individually. Therefore, instead of one multiplication instruction in the case of KNL, the KNC coprocessor needs two (not including the overhead for operand merging), justifying the performance degradation.

VII. Conclusion - Future work

In this paper, we present a study of the Intel Xeon Phi's SIMD extensions for use in large integer arithmetic. For this purpose, we implement a radix-2^m multiplication algorithm and compare our results for the first and second generation

TABLE I
INSTRUCTION COUNT RESULTS

Operand Size (bits)	**GMP**	**Radix-2^{29} multiplication algorithm (instructions)**			
		KNC	**Ratio**	**KNL**	**Ratio**
256	4264	879	4.85	676	6.31
512	4652	1923	2.42	1405	3.31
1024	6452	4749	1.36	3493	1.85
2048	12841	14388	0.89	11055	1.16
4096	30247	54805	0.55	43657	0.69

of Intel Xeon Phi, Knights Corner and Knights Landing respectively, with the GMP library multiplication. Our results show that a large integer multiplication can potentially benefit from the wide SIMD vectors of the Intel Xeon Phi, given specific constraints. For the KNC, this translates to the lack of specific integer vector instructions in the coprocessor's ISA. Additionally, for both platforms we study, large operands of 2048 and 4096 bits perform worse than GMP, because of the extra cost that is induced due to the cleanup process required for these sizes.

With regards to future directions, more advanced multiplication algorithms, such as the Karatsuba [31] and Toom-Cook methods [32], or a combination of both, will be implemented for the vector unit of the Xeon Phi. Additionally, we will perform a validation of our results on actual hardware, comparing actual cycle count and execution time, whenever the coprocessors become available.

REFERENCES

[1] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[2] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*, CRC press, 1996.

[3] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography engineering: design principles and practical applications*, John Wiley & Sons, 2012.

[4] W. Diffie and M. E. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644–654, 1976.

[5] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[6] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in Cryptology*. Springer, 1985, pp. 10–18.

[7] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[8] NIST, "Digital Signature Standard, FIPS PUB 186-4, National Institute of Standards and Technology," 2013.

[9] S. Gueron and V. Krasnov, "Software implementation of modular exponentiation, using advanced vector instructions architectures," in *Arithmetic of Finite Fields*, pp. 119–135. Springer, 2012.

[10] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, Newnes, 2013.

[11] Intel, "Intel64 and IA-32 Architectures Software Developer Manuals," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2014, [Online; accessed 20-Mar-2014].

[12] A. Cortex, "A9 technical reference manual, 2012," .

[13] Intel, "Intel Xeon Phi Coprocessor Developers Quick Start Guide," http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide, 2013, [Online; accessed 20-Mar-2014].

[14] J. Reinders, "AVX-512 instructions," http://software.intel.com/en-us/blogs/2013/avx-512-instructions/, 2013, [Online; accessed 20-Mar-2014].

[15] S. Moore, "Using streaming SIMD extensions (SSE2) to perform big multiplications. Application Note AP-941, Intel Corporation, 2000. Version 2.0," *Order*, , no. 248606-001.

[16] K. C. Pabbuleti, D. H. Mane, A. Desai, C. Albert, and P. Schaumont, "SIMD acceleration of modular arithmetic on contemporary embedded platforms," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–6.

[17] D. J. Bernstein and P. Schwabe, "NEON crypto," in *Cryptographic Hardware and Embedded Systems–CHES 2012*, pp. 320–339. Springer, 2012.

[18] J. W. Bos and M. E. Kaihara, "Montgomery multiplication on the cell," in *Parallel Processing and Applied Mathematics*, pp. 477–485. Springer, 2010.

[19] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records," in *Public Key Cryptography-PKC 2006*, pp. 207–228. Springer, 2006.

[20] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[21] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.1.3 edition, 2013, http://gmplib.org/.

[22] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck, "Cryptographics: Secret key cryptography using graphics cards," in *Topics in Cryptology–CT-RSA 2005*, pp. 334–350. Springer, 2005.

[23] S. Fleissner, "Gpu-accelerated montgomery exponentiation," in *Computational Science–ICCS 2007*, pp. 213–220. Springer, 2007.

[24] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. Jarvis, "Exploring SIMD for molecular dynamics, Using Intel Xeon Processors and Intel Xeon Phi coprocessors," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 1085–1097.

[25] J. Fang, A. L. Varbanescu, H. Sips, L. Zhang, Y. Che, and C. Xu, "An empirical study of Intel Xeon Phi," *arXiv preprint arXiv:1310.5842*, 2013.

[26] M. Barth, K. Sweden, M. Byckling, C. Finland, N. Ilieva, N. Bulgaria, S. Saarinen, M. Schliephake, V. Weinberg, and L. Germany, "Best practice guide Intel Xeon Phi v1.1," .

[27] J. Reinders, "An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors," http://download-software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf, 2012, [Online; accessed 20-Mar-2014].

[28] Intel, "Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual ," http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf, 2012, [Online; accessed 20-Mar-2014].

[29] A. E. Eichenberger, P. Wu, and K. O'brien, "Vectorization for simd architectures with alignment constraints," in *ACM SIGPLAN Notices*. ACM, 2004, vol. 39, pp. 82–93.

[30] Intel, "Intel Instruction Set Architecture Extensions," http://download-software.intel.com/sites/default/files/managed/68/8b/319433-019.pdf, 2014, [Online; accessed 20-Mar-2014].

[31] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, 1963, vol. 7, p. 595.

[32] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," in *Soviet Mathematics Doklady*, 1963, vol. 3, pp. 714–716.