SIMD Acceleration of Modular Arithmetic on Contemporary Embedded Platforms

Krishna Chaitanya Pabbuleti, Deepak Hanamant Mane, Avinash Desai, Curt Albert and Patrick Schaumont
Department of Electrical and Computer Engineering
Virginia Polytechnic and State University
Blacksburg, VA 24060
Email: {kriscp4, mdeepak, aviraj, falbert9, schaum}@vt.edu

***Abstract*—Elliptic curve cryptography (ECC) is a public key crypto system popular for embedded implementations because of its shorter key sizes. ECC computations are complex; they involve point additions and doublings on elliptic curves over finite fields. The execution time of ECC is completely dominated by modular multiplications in these fields. In this contribution, we propose vector processing techniques to accelerate modular multiplications in prime fields. We demonstrate implementations for the Venom (NEON) coprocessor in Qualcomm's Scorpion (ARM) CPU, as well as for the SSE2 instruction-set extensions in Intel's Atom CPU. Our implementations, which use NIST-standard prime-field curves, run more than two times faster than the OpenSSL versions of the same ECC operations on the same processor.**

I. Introduction

Handheld computing is an emerging market with increasingly complex and powerful processors to provide wide range of services to the end user. Current processors targeted at the handheld market contain several coprocessors and accelerators which can perform specialized functions, such as signal processing and video acceleration, without burdening the main processor. Information security is a common requirement for those platforms as well. Public key cryptographic algorithms like RSA, DSA and elliptic curve cryptography (ECC) are needed to support key exchange and signature protocols. ECC in particular is popular in embedded context because of the smaller key sizes compared to other public key cryptographic systems. For example, at 128-bit equivalent security, we need 3072-bit public key in RSA, whereas ECC requires only a 256-bit public key.

Our work focuses on efficient implementation of modular arithmetic which forms the basis for ECC, by exploiting the capabilities of SIMD coprocessors in Intel Atom and Qualcomm Snapdragon processors. High-performance implementations are important, also for handheld computing platforms. Indeed, faster cryptography enables higher levels of equivalent security, or higher availability of handheld computing resources for other tasks. Furthermore, new applications such as e-cash and privacy-friendly attributes make extensive use of public-key primitives.

ECC was independently proposed by Neal Koblitz and Victor S. Miller and is based on point algebra of elliptic curves over a finite field. The security of ECC is based on the Elliptic Curve Discrete Logarithm Problem which states that given two points P and Q on the curve such that $Q = k.P$, it is very hard to find k. This operation is the basis for secure protocols for signing and key exchange: the private key k is a very long integer chosen at random and the public key is derived by scalar multiplication of point P with k. The scalar multiplication thus forms the basis of ECC. All points on an elliptic curve form a group. One can either add two different points, or one can double a point (add it to itself). Using point adding and point doubling, a very basic implementation of $k.P$ is shown in Algorithm 1. Each point operation (adding or doubling) requires multiple modular-arithmetic operations in the underlying field. For example, for a Weierstrass curve with Jacobian coordinates (a common choice for NIST curves), one may find 7 modular multiplications and 3 squaring operations for each point doubling, and 12 modular multiplications and 2 modular squarings for point adding [8]. It's easy to see that computing $k.P$ is dominated by modular multiplications. If we assume a 192-bit random k, we may expect to see about 190 point-double and 85 point-add operations, leading to 1362 modular multiplications and 740 modular squarings for *each* $k.P$.

Algorithm 1 Right-to-left binary method for point multiplication [8]

INPUT: $k = (k_{t-1}, ..., k_1, k_0)_2, \quad P \in E(F_q)$.
OUTPUT: kP.
1. $Q \leftarrow \infty$.
2. For i from 0 to t-1 **do**
    2.1 If $k_i = 1$ then $Q \leftarrow Q + P$.
    2.2 $P \leftarrow 2P$.
3. Return(Q).

We propose efficient implementation techniques for modular multiplication using advanced architectural features (and SIMD in particular) on modern SoCs. Yan and others showed that DSP processors are efficient in accelerating modular arithmetic [15], [16]. Morozov later demonstrated how embedded DSP cores in SoC could be used to accelerate modular arithmetic [13]. Similarly, the SSE extension on x86 processors enables efficient big number multiplication [11]. Bernstein et al showed that significant performance gain can be achieved by using NEON SIMD extension [3]. However, his demonstration used a specialized curve called `curve25519`. In this work, we target NIST recommended curves over prime fields. NIST primes are special primes which are of the form $2^m \pm 2^n - \quad ..... \quad - 1$. Reduction is very easy in these fields because of the structure of the prime number. The five 5 NIST primes (P192, P224, P256, P384, P512) are used

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

TABLE I
COMPARISON BETWEEN INTEL AND SNAPDRAGON HARDWARE PLATFORMS

Specification	Intel Atom N2800	Qualcomm Snapdragon APQ8060
Processor Speed	1.86 GHz	1 GHz
Number of Cores	2	2
Instruction Set	x86-64	ARMv7
Instruction Set Extensions	SSE2	Thumb2, Neon, VFPv3
Number of Registers	16 32-bit Registers in 32-bit mode, 16 64-bit Registers in 64-bit mode	16 32 -bit registers (r0-r15)
Number of vector registers	16 128-bit registers (XMM0-XMM15)	16 128-bit registers (q0-q15)
L1 Cache Size	56 KB (24 KB data and 32 KB Instruction) with 2 cycle latency	16 KB data cache and 16 KB instruction cache
L2 Cache Size	1 MB (512 KB per core) with 15 cycle latency	512 KB

for standard curves called `nistp192`, `nistp224`, ..., which require modular operations using 192-bit, 224-bit, ... prime field numbers. Recent work by Kasper demonstrated portable (non-SIMD), 64-bit optimized versions of `nistp224` [17]; our work explores the opportunities offered by SIMD processor extensions.

We implement high performance modular arithmetic on two different hardware platforms supporting SIMD instructions and we compare their performance. We evaluate our implementation on Qualcomm Scorpion and on Intel Atom processor. The Scorpion processor is one of the seven processors integrated in Qualcomm's Snapdragon platform; we will refer to the more common term Snapdragon for the remainder of this paper. We evaluate the performance of `nistp192` and `nistp224` elliptic curves on both the platforms. The same concept could be extended to other standard curves as well.

The major contributions of this work are:

- Implementation of efficient modular multiplication on SIMD architecture for NIST primes.
- Apples-to-apples performance comparison of vectorized modular arithmetic on contemporary embedded platforms, including cycle count performance and analysis of the instruction set.

The remainder of the paper is organized as follows. In the next section we describe the SIMD features of the Qualcomm Snapdragon and the Intel Atom. Section 3 explains different implementation techniques for modular multiplication and optimization for both hardware platforms. Section 4 shows implementation results, including measured performance for P192 and P224 and Section 5 concludes the paper.

II. HARDWARE PLATFORM

We compare the performance of modular arithmetic on two different hardware platforms namely, Qualcomm Snapdragon APQ8060 and Intel Atom N2800. The Snapdragon features a dual CPU (Scorpion) architecture, running up to 1.7 GHz each,

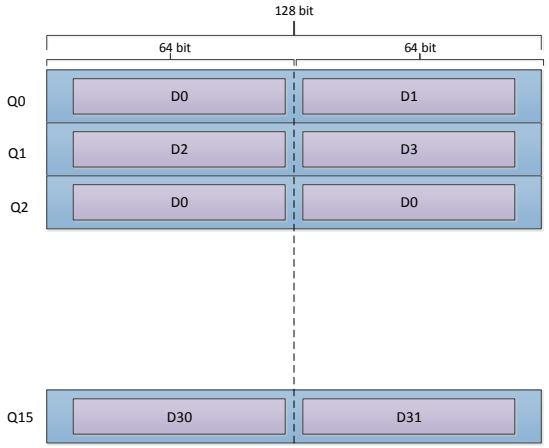

Fig. 1. Registers in NEON

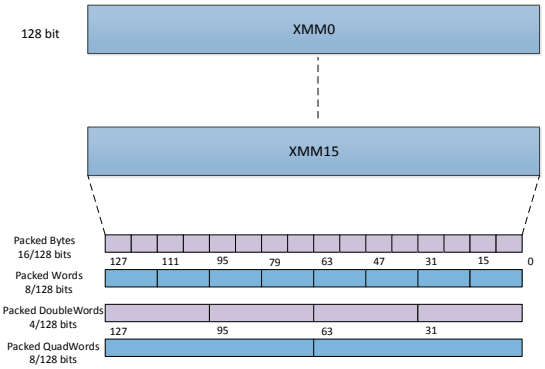

Fig. 2. Fundamental 128-Bit packed SIMD data types in SSE2 [6]

two VeNum (NEON) 128-bit SIMD multimedia coprocessors and combined 512 KByte L2 cache. The Scorpion processor is designed in-house but has many architectural similarities with the ARM Cortex-A8 and ARM Cortex-A9 CPUs. Functionally, Scorpion is an intermediary step between Cortex-A8 and Cortex-A9, supporting some but not all of Cortex-A9's out-of-order instruction execution capabilities. Scorpion-based Snapdragon SOCs implement Cortex-A8 and A9-compliant floating-point and NEON SIMD engines. We specifically target NEON coprocessor for our SIMD optimizations.

The NEON architecture has sixteen 128-bit vector registers, `q0` through `q15` as shown in Figure 1. It also includes thirty-two 64-bit vector registers, `d0` through `d31`, but these registers share physical space with the 128-bit vector registers: `q0` is the concatenation of `d0` and `d1`, `q1` is the concatenation of `d2` and `d3`, and so on. The basic ARM architecture has only sixteen 32-bit registers, `r0` through `r15`. Register `r13` is the stack pointer and register `r15` is the program counter, leaving only fourteen 32-bit registers for general use. An obvious benefit of NEON for cryptography is that it provides much more space in registers, reducing the number of loads and stores from memory. The 128-bit arithmetic unit can perform four 32-bit operations in each cycle. The Cortex A8 NEON microarchitecture has one 128-bit arithmetic unit and one 128-bit NEON load/store unit that runs in parallel with the NEON arithmetic unit.

The second processor we use is the Intel Atom N2800 Pro-

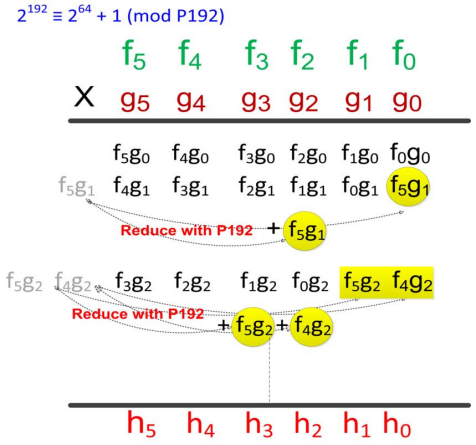

Fig. 3. Modular Multiplication with Reduction

cessor. The Atom platform aims at the power performance of a RISC architecture while still maintaining the x86 instruction set. The Intel Atom N2800 (formerly known as Cedar Trail) contains a dual core 1.86 GHz processor with support for the SSE2 instruction set. The SSE2 instruction set adds sixteen 128-bit SIMD registers to the general purpose x86-64 register set.

The SSE2 instruction set adds support for SIMD instructions through the addition of sixteen 128-bit registers, `XMM0` through `XMM15`. Eight of these were introduced with the original SSE instruction set and eight more were added in the Intel 64 architecture. In the Intel architecture, unlike MIPS, a word is of 16 bits. Figure 2 shows the SIMD data types in SSE2.

III. Implementation

NIST primes are chosen to have a simple structure in terms of powers of two, so that the modulo-operation (also called *reduction*) is easy. In a field based on a prime $P = 2^n \pm 2^m \pm 1$, reduction is based on the relation that $2^n = \mp 2^m \mp 1 (mod P)$.

A. Modular Multiplication in P192

The smallest prime among NIST primes is $P192 = 2^{192} - 2^{64}$ - 1. Any number larger than this prime can be reduced by using the relation $2^{192} = 2^{64} + 1 (mod P192)$. We describe two methods to perform modular multiplication in this field. The first method is a variant of schoolbook multiplication, where each of the partial products is reduced as they are computed. The second method involves reducing one of the multiplicands before computing each of the partial product rows.

1) Schoolbook multiplication with intermediate reduction: First, the number is represented in a polynomial form in radix 32 as below:
$f = f0.2^0 + f1.2^{32} + f2.2^{64} + f3.2^{96} + f4.2^{128} ... + f5.2^{160}$
$g = g0.2^0 + g1.2^{32} + g2.2^{64} + g3.2^{96} + g4.2^{128} ... + g5.2^{160}$

The basis for this approach is schoolbook multiplication as shown in Figure 3. Each row of the partial products is reduced as they are calculated. The partial products whose coefficients are greater than 2^{192} are added back into 2^{64} and 2^0. This is possible because of the relation $2^{192} = 2^{64} + 1$ in P192 prime field. This gives us expressions for terms in polynomial form of the product shown in Listing 1. f and g represent the multiplicands and h represents the reduced product.

```
h0 = f0g0 + f1g5 + f2g4 + f3g3 + f4g2 + f5g1+ f5g5
h1 = f0g1+ f1g0 + f2g5 + f3g4 + f4g3 + f5g2
h2 = f0g2 + f1g1+ f1g5 + f2g0 + f2g4 + f3g3 + f3g5 + f4g2 + f4g4 + f5g1
     + f5g3 + f5g5
h3 = f0g3 + f1g2 + f2g1+ f2g5 + f3g0 + f3g4 + f4g3 + f4g5 + f5g2 + f5g4
h4 = f0g4 + f1g3 + f2g2 + f3g1+ f3g5 + f4g0 + f4g4 + f5g3 + f5g5
h5 = f0g5 + f1g4 + f2g3 + f3g2 + f4g1+ f4g5 + f5g0 + f5g4
```

Listing 1. Expressions for terms in polynomial form of the product

It can be seen in the expressions of Listing 1 that we are adding multiple 64-bit values, which may result in overflow. In order to avoid this overflow, operands are represented using a redundant representation. These redundant representations have some slack at the MSB side, so that the carry does not overflow. For example, we have chosen radix-24 to represent a number. So, the partial products are of size 48 bits. Thus there are 16 bits of slack to accommodate the overflow. Each number is represented in polynomial form in radix-24 as in Listing 2.

```
f = f0.2^0 + f1.2^24 + f2.2^48 + f3.2^72 + f4.2^96 + f5.2^120 + f6.2^144
    + f7.2^168
g = g0.2^0 + g1.2^24 + g2.2^48 + g3.2^72 + g4.2^96 + g5.2^120 + g6.2^144
    + g7.2^168
```

Listing 2. Polynomial Form

We now multiply f and g term by term in the same fashion as we do above using schoolbook multiplication, and reduce the partial products. The powers greater than 2^{192} are added back into 2^{64} and 2^0, reduced and arranged in terms of their power as shown in Listing 3.

```
2^0   : f0g0 + f1g7 + f2g6 + f3g5 + f4g4 + f5g3 + f6g2 + f7g1
2^16  : f7g7
2^24  : f0g1 + f1g0 + f2g7 + f3g6 + f4g5 + f5g4 + f6g3 + f7g2
2^48  : f0g2 + f1g1 + f2g0 + f3g7 + f4g6 + f5g5 + f6g4 + f7g3
2^64  : f1g7 + f2g6 + f3g5 + f4g4 + f5g3 + f6g2 + f7g1
2^72  : f0g3 + f1g2 + f2g1 + f3g0 + f4g7 + f5g6 + f6g5 + f7g4
.
.
```

Listing 3. Reduction of Partial Products

It can be observed that there are powers which are not present as terms in the original polynomial. For example, 2^{16} and 2^{64} are not present in the original polynomial, but are present in the result. So, these have to be split and added into the nearest powers. For example, the lower 8 bits of 2^{16} are shifted 16 places and added into 2^0 and the upper 16 bits are added in 2^{24}. After eliminating all the non-aligned powers, carries are propagated and final reduced result is produced. The partial products f0g0, f0g1, ..., f7g7 are computed on SIMD coprocessor, two at a time. The accumulation, shifting and carry propagation are all carried on the main processor. SIMD coprocessor is very efficient in performing multiplications, but performs poorly for other operations like shifting and carry propagation. So, partial products are computed on SIMD coprocessor, whereas the subsequent operations are carried out on main processor.

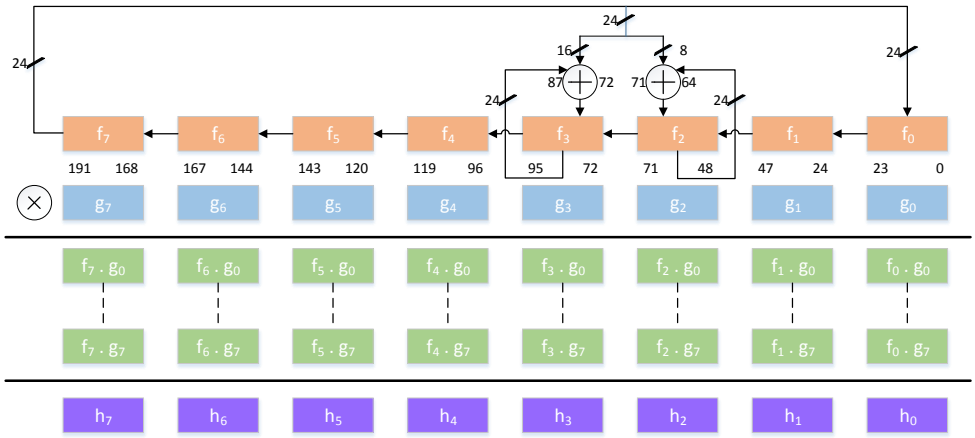

Fig. 4. P192 Implementation

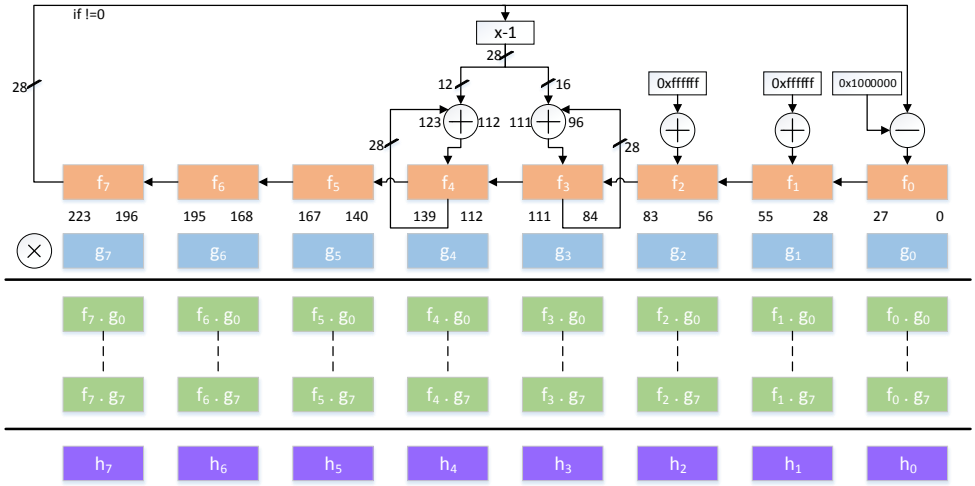

Fig. 5. P224 Implementation

2) Multiplicand reduction method: Instead of the shifting and reducing the partial products, one of the multiplicands is reduced after calculating one row of partial product. Mane discusses an FPGA implementation of modular multiplication in which one of the operands is reduced as the multiplication progresses [12]. We have adapted this technique on SIMD platforms by reducing the number of reduction stages. We have also eliminated intermediate carry propagation by using redundant representation of operands. The operands f and g are represented in redundant representation in radix-24 in the same way as discussed in the previous method. First row of partial products are obtained by multiplying f with g0. Now, f is shifted left by 24 bits, the 24 bits beyond 2^{192} are added to 2^0 and 2^{64}, which produces reduced multiplicand. This shifting and adding back is based on the relation $2^{192} = 2^{64} + 1$ in P192 prime field, which produces the reduced result. It should be noted that, because of the redundant representation, lower 8 bits of 2^{64} are in f2 and remaining 16 bits are present in f3. So, lower 8 bits of f7 are added in upper 8 bits of f2 and upper 16 bits of f7 are added in lower 16 bits of f3. We do not need to worry about carry because of the slack present due to redundant representation. Now, the second row of partial products is calculated by multiplying reduced f with g1 and added to the previous partial products. Similarly, all partial product rows are computed by shifting f and reducing it, followed by a multiplication with g. In the end, carries are propagated and any trailing bits beyond 2^{192} are again reduced by adding them to 2^0 and 2^{64}.

B. Modular Multiplication in P224

The next bigger prime in NIST primes is $P224 = 2^{224} - 2^{96} + 1$. Any number larger than this prime can be reduced by using the relation $2^{224} = 2^{96} - 1(mod P192)$. Results of P192 indicate that the multiplicand reduction method is significantly faster than the schoolbook multiplication with intermediate reduction method. Therefore, we have implemented only multiplicand reduction method for P224.

1) Multiplicand reduction method: In the same way as above, one of the multiplicands is shifted and reduced after calculating one row of partial product. The numbers f and g are represented in redundant representation in radix-28 as in Listing 4

f = f0.2^0 + f1.2^{28} + f2.2^{56} + f3.2^{84} + f4.2^{112} + f5.2^{140} + f6.2^{168} + f7.2^{196}
g = g0.2^0 + g1.2^{28} + g2.2^{56} + g3.2^{84} + g4.2^{112} + g5.2^{140} + g6.2^{168} + g7.2^{196}

Listing 4. Redundant representation in Radix-28

First row of partial products are obtained by multiplying f with g0. Now, f is shifted left by 28 bits, the 28 bits beyond 2^{224} are added to 2^{96} and subtracted from 2^0, which produces the reduced result. In the same way as above, because of the redundant representation, the lower 16 bits of 2^{96} are in f3 and the remaining 12 bits are present in f4. The lower 16 bits of f7 are added in upper 16 bits of f3 and the upper 12 bits of f7 are added in lower 12 bits of f4. When f is shifted left, f0 is zero and we need to subtract f7 from f0. We need to get borrow from neighbor, but the neighbor is not guaranteed to be a non-zero number. The borrow is taken from f7 which is being added to f3 and f4. Because we have taken the borrow from non-neighbor, we have to add 0xffffffff to the members between f3 and f0, i.e., f1 and f2. This is best illustrated in the Figure 5.

C. Platform Specific Optimization

This section explains specific implementation details for NEON and SSE2 vector coprocessors. NEON can do two multiplications or two multiply and accumulate operations at a time. Moreover, we can take advantage of multiply and accumulate instruction. As shown in Figures 4 and 5, we first multiply f0 and f1 with g0 in one cycle and obtain partial products f0g0 and f1g0. In the next cycle, we multiply f2 and f3 with g0 and obtain f2g0 and f3g0 and so on. After multiplying all the limbs of f with g0, f is shifted left by 24-bits and reduced. It should be noted that this reduction is performed on main processor because it is more efficient in handling these operations compared to Neon coprocessor. In the next step, f0 and f1 are multiplied with g1 and the results are added to previously calculated f0g0 and f1g0, using multiply accumulate instruction. This gives additions for free and also saves registers, as there is no need to save the intermediate partial products. In the end, the partial products are moved to main processor, carry propagation is done and final product is calculated.

In SSE2, the modular multiplication is implemented as follows. We use `PMULUDQ` to perform two multiplication operations simultaneously. `PMULUDQ` can perform multiplication operation on two of the four 32-bit values read from memory. We can use `PSHUFD` to rearrange the 32-bit values to allow us to use all of the data read in from memory before writing it back or reading in new data. `PADDQ` is used to accumulate the 64-bit partial products produced in the multiplicand reduction algorithm. `PSHUFD` instruction is used to order the result so that it can be properly written back to memory, 128 bits at a time. The downside of the SSE2 instructions is that they overwrite one of the operands. When you call an instruction on two operands, it calculates the result and stores it in one of the operand registers overwriting the operand. In order to preserve the operand, we have store it in a different register beforehand which adds overhead. Moreover, SSE2 doesn't have multiply and accumulate instruction, so the intermediate results have to be saved and added to the partial product, adding further overhead.

IV. Results

Table II shows the performance for the two proposed approaches on the Qualcomm Snapdragon and Intel Atom for prime field P192. The code is compiled using gcc version 4.4.5 on ARM and gcc version 4.6.3 on Intel Atom. As our modular multiplication is coded in assembly, we expect that the difference in compiler versions has minimal impact. Clock cycles are measured by reading counter registers from Performance Monitoring Unit inside CP15 coprocessor of ARM. On Intel, clock cycles are measured using `RDTSC` instruction. Schoolbook multiplication is significantly slower than the multiplicand reduction method, because the former method involves several shifts and adds which adds a lot of overhead. Atom takes more cycles than ARM in all the cases. Similar trend can be noticed in the numbers obtained from eBACS benchmark in Table IV [4]. Vectorising the multiplications increases the performance by almost 30% in all the cases. Both our methods are faster than the modular multiplication using GMP multiprecision library. Table III shows clock cycles comparison for multiplicand reduction method on the ARM and Atom for prime field P224. We did not implement schoolbook multiplication for P224 because results from P192 show that multiplicand reduction is already faster. Vectorising the modular multiplication gives a 40% improvement in the performance on ARM whereas gives only 22% improvement on Intel Atom. This may be attributed to overheads present in SSE2 like non-availability of multiply-accumulate instruction and result replacing one of the operands.

TABLE II
Cycles for modular multiplication for P192

Algorithm	Snapdragon	Snapdragon +NEON	Atom	Atom +SSE2
Intermediate Reduction Method	1116	858	1282	842
Multiplicand Reduction Method	574	404	864	685
GMP	1786	-	2122	-

TABLE III
Cycles for modular multiplication for P224

Algorithm	Snapdragon	Snapdragon +NEON	Atom	Atom +SSE2
Multiplicand Reduction Method	685	405	1083	836
GMP	1858	-	2286	-

Table IV shows the comparison of point multiplication on various embedded platforms against point multiplication using our modular multiplication techniques. The point multiplication is performed in projective coordinate system in all the implementations given in the table. Our implementation is faster than the other generic implementations of comparable security level. However, specialized implementations like `curve25519` on Snapdragon by Bernstein [3] performs better than our implementation. As expected, Atom takes more number of cycles than Snapdragon because the modular multiplication on Atom takes more number of cycles. A similar trend is found in the cycle counts obtained from the eBACS benchmark suite, where Atom takes 1.5 to 1.7 times more number of cycles than Snapdragon.

Figure 6 shows the data from Table IV in a manner to appreciate the merit of SIMD vectorization. The figure shows the number of Point Multiplications completed per million cycles of the target architecture, for different curve sizes and different architectures. The vectorized implementations (SSE2

TABLE IV
SCALAR MULTIPLICATION PERFORMANCE FOR VARIOUS EMBEDDED PLATFORMS

Platform	Curve and Implementation	Cycles
TI OMAP 3530(ARM A8 + C64x + DSP) [13]	secp160r1	1,054K
TI OMAP 3530(ARM A8 + C64x + DSP) [13]	secp224r1	2,175K
Qualcomm Snapdragon	Ed25519 [4]	3,295K
Intel N280	Ed25519 [4]	5,774K
Qualcomm Snapdragon	P192 ebacs_OpenSSL	3,143K
Qualcomm Snapdragon	P224 ebacs_OpenSSL	3,996K
Intel N280	P192 ebacs_OpenSSL	4,973K
Intel N280	P224 ebacs_OpenSSL	6,349K
Qualcomm Snapdragon with NEON	Curve25519 [3]	511K
Qualcomm Snapdragon	P192 with Multiplicand Reduction	**1,591K**
Qualcomm Snapdragon with NEON	P192 with Multiplicand Reduction	**1,243K**
Intel N280	P192 with Multiplicand Reduction	**2,390K**
Intel N280 with SSE2	P192 with Multiplicand Reduction	**2,115K**

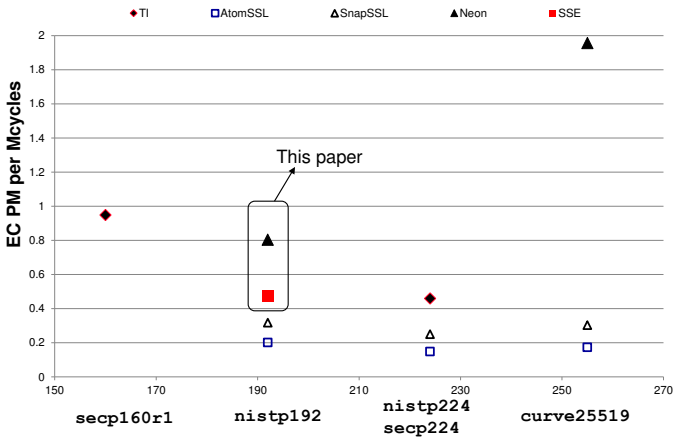

Fig. 6. Number of Point Mult per million Cycles

on Atom, NEON on Snapdragon, DSP64x on OMAP) are shown as solid symbols; the default ones are shown as open symbols.

V. CONCLUSION

In this paper we demonstrated how SIMD extensions could be exploited to accelerate the underlying modular arithmetic. We implemented and compared the modular multiplication on two different embedded computing platforms. A generalized modular multiplication for NIST primes and its implementation on SIMD platforms is presented. Our results show that accelerating basic cryptographic operations on mobile platforms can have large impact on the performance of secure mobile applications. Our future work includes the system integration of elliptic curve cryptography based protocols on mobile platforms. Since Intel N2800 runs at higher frequency, time taken for scalar multiplication on both platforms is almost same. But it might be interesting to study the power consumption for modular multiplication on both platforms.

ACKNOWLEDGEMENTS

This work was supported in part through a NIST grant 60NANB10D004 and NSF grant 0855095.

REFERENCES

[1] D. F. Aranha and C. P. L. Gouvêa, "RELIC an Efficient LIbrary for Cryptography", *available online at* http://code.google.com/p/relic-toolkit/

[2] ARM Limited, "Cortex-A8 technical reference manual", revision r3p2, 2010. *available online at* http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/index.html

[3] D. J. Bernstein and P. Schwabe, "NEON crypto," in *Proceedings of the 14th international conference on Cryptographic Hardware and Embedded Systems CHES'12*, 2012, pp. 320-339.

[4] D. J. Bernstein and T. Lange, "eBACS: ECRYPT Benchmarking of Cryptographic Systems", http://bench.cr.yp.to.

[5] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records." in *9th international conference on theory and practice in public-key cryptography, PKC 2006*, New York, NY, USA, 2006, pp. 207-228.

[6] Dan Garcia, "Great Ideas in Computer Architecture SIMD II," Oct. 2012. *available online at* http://www-inst.eecs.berkeley.edu/~cs61c/fa12/lectures/18LecFa12DLPII.pdf

[7] GNU Multiprecision Library *available online at* http://gmplib.org/

[8] D. Hankerson, A. Menezes, S. Vanstone, "Guide to Elliptic Curve Cryptography", Springer 2004.

[9] Intel,"Evaluation Platforms", *available online at* http://www.intel.com/p/en_US/embedded/designcenter/tools/seed-board-program?iid=1032# expired

[10] Intel, "Intel Atom Processor N2800", *available online at* http://ark.intel.com/products/58917/Intel-Atom-Processor-N2800-(1M-Cache-1_86GHz)#infosectionpackagespecifications

[11] Intel, Using Streaming SIMD Extensions (SSE2) to perform Big Multiplications, *available online at* http://software.intel.com/sites/default/files/m/c/c/7/4/8/24960-40809_w_big_mul.pdf

[12] S. Mane, L. Judge and P. Schaumont, "An Integrated Prime-Field ECDLP Hardware Accelerator with High-Performance Modular Arithmetic Units," in *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2011, pp.198-203.

[13] S. Morozov, C. Tergino and P. Schaumont, "System integration of Elliptic Curve Cryptography on an OMAP platform," in *Proceedings of the 9th IEEE Symposium on Application Specific Processors SASP'11*, 2011, pp. 52-57.

[14] NSA, Mathematical routines for the NIST prime elliptic curves, *available online at* http://www.nsa.gov/ia/_files/nist-routines.pdf

[15] C. Tergino, "Efficient binary field multiplication on a VLIW DSP", 2009, Master's Thesis Virginia Tech - 06222009-150103.

[16] H. Yan, Z. J. Shi and Y. Fei, "Efficient implementation of elliptic curve cryptography on DSP for underwater sensor networks", in *Proceedings of the 7th Workshop on Optimizations for DSP and Embedded Systems (ODES-7)*, 2009.

[17] E. Kasper, "Fast Elliptic Curve Cryptography in OpenSSL", in *Financial Cryptography and Data Security: FC 2011 Workshops, RLCPS and WECSR*, 2011.