Fast GPGPU-Based Elliptic Curve Scalar Multiplication

Eric M. Mahé and Jean-Marie Chauvet

MassiveRand
{eric.mahe,jmc}@massiverand.com
http://www.massiverand.com
62, ave. Pierre Grenier, 92100 Boulogne-Billancourt, France

Abstract. This paper presents a fast implementation to compute the scalar multiplication of elliptic curve points based on a "General-Purpose computing on Graphics Processing Units" (GPGPU) approach. A GPU implementation using Dan Bernstein's Curve25519, an elliptic curve over a 255-bit prime field complying with the new 128-bit security level, computes the scalar multiplication in less than a microsecond on AMD's R9 290X GPU. The presented methods and implementation considerations can be applied to any parallel architecture.

Keywords: Cryptography, GPU, Random Bit Generator, ECC, Curve25519, Ed25519, OpenCL

1 Motivations

In the current controversial context caused by the disclosure to the press of classified details of several top-secret United States and British government mass surveillance programs by former NSA contractor Edward Snowden [21], issues of data privacy, anonymity, forward secrecy and deniability have raised to public prominence. Providing high-speed cryptographic performance, leveraging alternate uses of state-of-the-art yet ubiquitous computing platforms, could contribute to generalize symmetric cryptography. Namely, should high-speed ECC be available on the current generation of low-cost devices, streaming key exchange protocols and on-the-fly one-time pad encryption may be broadly deployed.

2 Faster GPGPU ECC Scalar Multiplication

2.1 Timings

Previous reports implementing ECC schemes on GPUs [7, 26, 11, 4] have explored elliptic curve scalar multiplication algorithms for parallel hardware architecture, mostly in view of asymmetric cryptography. Performance improvements were demonstrated by dividing the elliptic curve arithmetic over multiple threads or

by dividing single finite field operations over the available resources. GPU architectures have evolved however, and in the direction of providing ever increasing acceleration of general purpose computing capabilities. Meanwhile programming models and development libraries such as NVIDIA's CUDA [23] and OpenCL [25] were maturing quickly.

Curve25519 [5], originally introduced by Bernstein, has now a well documented track record of reference implementations available on many platforms [1, 19, 9, 13, 10, 16]. It is generally recognized as one of the fastest practical implementation for the Diffie-Hellman key exchange protocol [20]. This work thus focuses on Curve25519, and uses several standard implementations of NIST-recommended curves as comparison data points.

Selected reference implementations of scalar multiplication on Curve25519 were ported to OpenCL: the NaCL reference implementation [9], the so-called *donna* variants for 32-bit and 64-bit architectures [1]. For comparison we also developed alternate OpenCL implementations based on optimizations suggested for SIMD architectures [27, 17, 15]. In all cases we ended up retaining the implementation where one complete scalar multiplication is computed per thread as also done in [4], which we call the *GPGPU approach*; only best performances are shown.

Table 1. Execution times in seconds for 1,048,576 scalar multiplications on Curve25519

Variant	i7 Quad-Core 3770	GTX Titan	HD 6870	R9 290X
OpenCL 32-bit	42.6	4.2	2.26	0.7
OpenCL 64-bit	84.3	2	2.27	n/a

Table 1 compares execution time in seconds of the best OpenCL kernels: on a multicore CPU, the *i7-3770* at 3.5GHz with 4 cores, 8 threads; and on a range of GPUs, the *GeForce GTX Titan* GPU, with 6GB, 2,688 processors, 4,494 GFLOPS; the *Radeon HD 6870*, with 1 GB GDDR5 memory, 1,120 processors and 2,000 GFLOPS; and the recently released *R9 290X* GPU, 4 GB GDDR5, 2,816 processors and 5,600 GFLOPS. The OpenCL 32-bit implementation uses the 32-bit scalar type of OpenCL 1.2; the OpenCL 64-bit implementation uses 64-bit instructions in the i7 CPU implementation and 64-bit scalar type support when available in the OpenCL driver for GPU. On a R9 290X, the OpenCL 32-bit implementation performs 1,778,000 scalar multiplications per second, i.e. one scalar multiplication in 563 nanoseconds.

2.2 Porting to OpenCL

Very little adaptation was required to move code from the reference implementations to the OpenCL 1.2 environment.

Integer types used here in the *floodyberry* implementation were mapped to OpenCL types as shown in Table 2. There were no particular difficulty as The

Table 2. Mapping scalar types from single-core reference donna implementation in [1] to OpenCL

Types in `curve25519-donna-32bit`	Built-in scalar types in OpenCL (API types for application)
`int32_t`	`int` (`cl_int`)
`uint32_t`	`unsigned int` (`cl_uint`)
`uint64_t`	`unsigned long` (`cl_ulong`)
`uint8_t`	`unsigned char` (`cl_uchar`)

OpenCL C programming language is based on the ISO/IEC 9899:1999 C language specification (the C99 specification).

As a second step, the original reference implementation of field element and elliptic point functions were moved into the kernel program file and declared as `inline` functions rather than as `DONNA_INLINE static`. This uses more memory but removes the context switches and stack operations associated with regular function calls in the kernel. Private memory and registers were used throughout, as the GPGPU approach, with a single thread for each scalar multiplication, does not need to share data between work items. Furthermore as the number of registers may vary from a multicore architecture to another, the OpenCL compiler pushing intermediate variables to slower local memory in case of register spilling, care was taken to optimize the number of private variables used in these ported functions. A simple instance of reducing register use is presented in Table 5. Similarly, as much as possible, loops were unrolled as comparison operations are expensive.

Listing 1.1. OpenCL kernel for base point scalar multiplication

```
#define WG_SIZE 256
#define KEY_SIZE 32

__kernel __attribute((reqd_work_group_size(WG_SIZE,1,1)))
void mr_scalar_mult(
                     __global u8 *g_pk,
                     __global u8 *g_sk)
{
        unsigned int tx = get_local_id  (0);
        unsigned int g  = get_group_id (0);
        unsigned int d  = get_local_size(0);

        __local uint8_t mypublic[KEY_SIZE*WG_SIZE];
        size_t i, offset;

        offset = tx*KEY_SIZE;
        for(i = 0; i < KEY_SIZE; i++)
        {
```

```
                mypublic[offset+i] = g_sk[offset + i + (g * d
                    * KEY_SIZE)];
        }
        mypublic[offset+0]  &= 248;
        mypublic[offset+31] &= 127;
        mypublic[offset+31] |= 64;

        curve25519_scalarmult(mypublic+offset);

        for(i = 0; i < KEY_SIZE; i++)
        {
                g_pk[offset + i + (g * d * KEY_SIZE)] =
                    mypublic[offset+i];
        }
}
```

Finally OpenCL kernels were created for scalar multiplication of the base point and for scalar multiplication of a non-base point as required for the Diffie-Hellman key exchange protocol. Arrays of scalars and arrays of elliptic curve points are passed to the kernel in `global` memory by the driving C++ program; results are also returned in `global` memory to the driving program. Listing 1.1 shows the OpenCL kernel which, given as an input the global memory array `g_sk`, containing $4096 * 256$ 256-bit random secret integers, performs $1,048,576$ base point scalar multiplications on Curve25519 on as many threads organized as 4096 work groups of 256 threads, returning the million plus points x coordinates in the global memory `g_pk`.

The timings presented include enqueuing OpenCL buffers from the driving C program and moving global data to local memory and registers in the OpenCL kernels.

2.3 Timings of NIST Curves

As comparison data points, several NIST curves OpenCL kernels were developed using the same GPGPU approach mentioned earlier. Table 3 shows fastest OpenCL kernel execution time for scalar multiplication over several NIST recommended elliptic curves [14], both on an i7 multicore CPU and the GTX Titan GPU. OpenCL kernels for the NIST curves were ported without modifications, beyond proper type mapping explained above, from the reference implementations in the micro-ECC library [22]. Execution times in both tables may be compared against execution time of the original reference implementations, in C and C++, running on the CPU without the benefits of parallelization introduced by OpenCL.

On the i7 Quad-Core, standard implementation benefit from Streaming SIMD Extensions 2 (SSE2), one of the Intel SIMD processor supplementary instruction sets, which, combined with the 64-bit variants of the reference implementations, provides faster execution times. Measured programs were built with Visual Studio 2008 and run on Microsoft Windows 7 Professional, using AMD OpenCL 1.2

Table 3. Execution times in seconds for 1,048,576 scalar multiplications on alternate elliptic curves

NIST EC	i7 Quad-Core 3770 32-bit	GTX Titan 32-bit
secp128r1	42.7	2.0
secp192r1	120.9	6.4
secp256r1	235.5	16.1
secp384r1	653.3	67.9

Table 4. Execution times in seconds for 1,048,576 scalar multiplications on the i7 CPU, standard non-OpenCL implementations

Curve	32-bit	64-bit and SSE2
Curve25519 (donna)	211.5	57
secp256r1	533	282

drivers Catalyst 13.11, NVIDIA ForceWare 331.82 and Intel OpenCL SDK 2013 R2.

3 Streaming Diffie-Hellman Exchange Protocols

In this section we briefly present the use of the fast GPGPU scalar multiplication implementation for state-of-the-art Elliptic Curve Diffie-Hellman key exchange (ECDH) primitives for security level of approximately 128 bits. The choice of the well-known Curve25519 and the GPGPU approach to OpenCL programming reflects the motivations presented earlier:

- Very fast, constant time execution. Every Diffie-Hellman exchange basically consumes four scalar multiplications, so faster performance of scalar multiplication $P \rightarrow [s]P$ is critical as it applies to a high throughput stream of both varying points P and scalars s in our motivating scenarios. Alternate optimizations afforded e.g. by pre-computations [20] are less relevant here since we are interested in the censorship circumvention context where keys are changing with each communication message.
- Compact keys and messages. Bernstein's Curve25519, which is based on an efficient, uniform differential addition chain applied to a well-chosen pair of curve and twist. Curve25519 and its twist are presented as Montgomery models. These models not only provide highly efficient group operations, but they are optimized for x-coordinate-only operations [5]. Importantly enough, typical Diffie-Hellman functions may be compromised if public keys are not validated [3]; Curve25519's design, in contrast, accepts every 32-byte string as a public key.

These desirable characteristics of Curve25519, with the reported GPU acceleration of elliptic curve operations, were thus packaged as an experimental plug-in for a popular Instant Messaging client, PSI+, a cross-platform Jabber/XMPP

[24] client[1]. In this preliminary implementation, the elliptic curve Diffie-Hellman key exchange protocol is performed prior to sending each message and the shared secret key is used as a one-time pad to encrypt and decrypt each individual XMPP message. Practically since millions of keys per seconds are available from the GPU, the Diffie-Hellman exchange is synchronized with the XMPP presence messages so that enough available shared secrets are immediately available when beginning a conversation.

There is an extensive literature analyzing the speed of various implementations of diverse Diffie-Hellman functions for various conjectured security levels. Most of these results are now found in the comprehensive ECRYPT Benchmarking of Cryptographic Systems [8]. Comparison with related art is not straightforward, however, since CPU cycles hardly compare to GPU kernels parallel threads and different GPU platforms may be employed, all with different architectural characteristics and performance capabilities. In addition, the rapid pace of progress of manufacturers may render experimental results obsolete quite quickly.

In [12], the authors report on GPU accelerated version of the LSB Invariant scalar point multiplication for binary elliptic curves implemented using the CUDA programming language. By varying coordinate systems and parallelization factors, timings measured ranged from 9.545 ms to 190.203995 ms for precomputations and from 10.363000 ms to 173.121002 ms for actual scalar point multiplications in $GF(2163)$ on an NVIDIA GTX 285 graphics card. The paper [6] reports on the advancement of the project to break the Certicom ECC2K-130 challenge: to compute an elliptic-curve discrete logarithm on a Koblitz curve over $\mathbb{F}_{2^{131}}$. The comprehensive optimization of the ECC2K-130 computations for GPU resulted in the execution of 320 million $\mathbb{F}_{2^{131}}$ field multiplications per second on a NVIDIA GTX 295 graphics card. In [2] a RNS base representation was proposed to perform elliptic curve point arithmetic, parallelizing each RNS channel on a GPU thread. The authors' results suggested a maximum throughput of 9990 scalar multiplications per second and minimum latency of 24.3 ms for a 224-bit underlying field, on an NVIDIA 285 GTX GPU. In [11] the author obtained a throughput of 290 000 operations per second, on a GTX 580, a GPU card from 2010 which has only 512 cores.

4 Conclusions

The simple GPGPU approach, throughput-oriented parallelization, to porting elliptic points arithmetic to multicore and parallel GPU architectures delivered encouraging timing results on modern graphic cards and multicore architectures. In the context of serving a high-throughput stream of keys behind a Web server, or in an autonomous instant messaging client, additional work is required towards a goal of even lower latency in such modern parallel architectures.

[1] PSI+ Project at url `http://psi-plus.com/`

References

1. Andrew M. Floodyberry. ed25519-donna. GitHub repository, March 2012.
2. Samuel Antao, Jean-Claude Bajard, and Leonel Sousa. Elliptic Curve point multiplication on GPUs. In François Charot, Frank Hannig, Jürgen Teich, and Christophe Wolinski, editors, *ASAP*, pages 192–199. IEEE, 2010.
3. Adrian Antipa, Daniel Brown, Alfred Menezes, RenÃ© Struik, and Scott Vanstone. Validation of elliptic curve public keys. In YvoG. Desmedt, editor, *Public Key Cryptography PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 211–223. Springer Berlin Heidelberg, 2002.
4. D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang. The billion-mulmod-per-second PC. In *Workshop SHARCS 2009: Special-purpose Hardware for Attacking Cryptographic Systems*, 2009.
5. Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Berlin Heidelberg, 2006.
6. Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. ECC2K-130 on NVIDIA GPUs. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology – INDOCRYPT 2010*, volume 6498 of *Lecture Notes in Computer Science*, pages 328–346. Springer-Verlag Berlin Heidelberg, 2010. Document ID: 1957e89d79c5a898b6ef308dc10b0446, `http://cryptojedi.org/papers/\#gpuev1l`.
7. Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. ECM on Graphics Cards. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2009.
8. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. accessed M 16, 2014.
9. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Proceedings of the 2nd international conference on Cryptology and Information Security in Latin America*, LATINCRYPT'12, pages 159–176, Berlin, Heidelberg, 2012. Springer-Verlag.
10. DanielJ. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.
11. Joppe W. Bos. Low-Latency Elliptic Curve Scalar Multiplication. *International Journal of Parallel Programming*, 40(5):532–550, 2012.
12. A.E. Cohen and K.K. Parhi. GPU accelerated elliptic curve cryptography in GF (2m). In *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pages 57–60. IEEE, 2010.
13. Frank Denis. libsodium, P(ortable|ackageable) NaCl-based crypto library. GitHub repository, 2013.
14. Patrick Gallagher and Cita Furlani. FIPS Pub 186-3 Federal Information Processing Standards publication digital signature standard (DSS), 2009.
15. Pascal Giorgi, Laurent Imbert, and Thomas Izard. Optimizing elliptic curve scalar multiplication for small scalars. In *Proc. SPIE*, volume 7444, pages 74440N–74440N–10, 2009.

16. Michael Hutter and Peter Schwabe. NaCl on 8-Bit AVR Microcontrollers. In Amr Youssef, Abderrahmane Nitaj, and AboulElla Hassanien, editors, *Progress in Cryptology AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2013.
17. Tetsuya Izu and Tsuyoshi Takagi. Fast Elliptic Curve Multiplications with SIMD Operations. *IEICE Transactions*, 87-A(1):85–93, 2004.
18. Tanja Lange and Dan Bernstein. Security dangers of the NIST curves. International State of the Art in Cryptography - Security Worksfop, Athens, May 2013.
19. Adam Langley. A collection of implementations of Curve25519, an elliptic curve Diffie Hellman primitive, 2008.
20. Adam Langley. Faster Curve25519 with precomputation. Blog post: Imperial Violet, May 2013.
21. Jeff Larson, Nicole Perlroth, and Scott Shane. Revealed: The NSA's Secret Campaign to Crack, Undermine Internet Security. *Pro Publica*, 2013.
22. Ken MacKay. micro-ECC: A small ECDH implementation for 32-bit microcontrollers. GitHub repository, Jan 2013.
23. NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2.
24. P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011.
25. John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, May 2010.
26. Robert Szerwinski and Tim Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 79–99. Springer, 2008.
27. W. Trei. Efficient Modular Arithmetic for SIMD Devices. *ArXiv e-prints*, October 2013.

Table 5. Simple instance of reducing the number of intermediate variables in the kernel program to avoid register spilling, after loop unrolling.

Original code in `curve25519-donna-32bit`

```
DONNA_INLINE static void
curve25519_swap_conditional(bignum25519 a,
                            bignum25519 b,
                            uint32_t iswap) {
const uint32_t swap = (uint32_t)(-(int32_t)iswap);
uint32_t x0,x1,x2,x3,x4,x5,x6,x7,x8,x9;

x0 = swap & (a[0] ^ b[0]); a[0] ^= x0; b[0] ^= x0;
x1 = swap & (a[1] ^ b[1]); a[1] ^= x1; b[1] ^= x1;
x2 = swap & (a[2] ^ b[2]); a[2] ^= x2; b[2] ^= x2;
x3 = swap & (a[3] ^ b[3]); a[3] ^= x3; b[3] ^= x3;
x4 = swap & (a[4] ^ b[4]); a[4] ^= x4; b[4] ^= x4;
x5 = swap & (a[5] ^ b[5]); a[5] ^= x5; b[5] ^= x5;
x6 = swap & (a[6] ^ b[6]); a[6] ^= x6; b[6] ^= x6;
x7 = swap & (a[7] ^ b[7]); a[7] ^= x7; b[7] ^= x7;
x8 = swap & (a[8] ^ b[8]); a[8] ^= x8; b[8] ^= x8;
x9 = swap & (a[9] ^ b[9]); a[9] ^= x9; b[9] ^= x9;
}
```

Adapted code in OpenCL kernel file

```
__inline void
curve25519_swap_conditional(bignum25519 a,
                            bignum25519 b,
                            uint32_t iswap) {
const uint32_t swap = (uint32_t)(-(int32_t)iswap);
uint32_t x0;

x0 = swap & (a[0] ^ b[0]); a[0] ^= x0; b[0] ^= x0;
x0 = swap & (a[1] ^ b[1]); a[1] ^= x0; b[1] ^= x0;
x0 = swap & (a[2] ^ b[2]); a[2] ^= x0; b[2] ^= x0;
x0 = swap & (a[3] ^ b[3]); a[3] ^= x0; b[3] ^= x0;
x0 = swap & (a[4] ^ b[4]); a[4] ^= x0; b[4] ^= x0;
x0 = swap & (a[5] ^ b[5]); a[5] ^= x0; b[5] ^= x0;
x0 = swap & (a[6] ^ b[6]); a[6] ^= x0; b[6] ^= x0;
x0 = swap & (a[7] ^ b[7]); a[7] ^= x0; b[7] ^= x0;
x0 = swap & (a[8] ^ b[8]); a[8] ^= x0; b[8] ^= x0;
x0 = swap & (a[9] ^ b[9]); a[9] ^= x0; b[9] ^= x0;
}
```