Speed up over the Rainbow

Nir Drucker[1,2] and Shay Gueron[1,2]

[1]University of Haifa, Israel, [2]Amazon, USA

Abstract. Rainbow is a signature scheme that is based on multivariate polynomials. It is one of the Round-2 candidates of the NIST's Post-Quantum Cryptography Standardization project. Its computations rely heavily on $GF(2^8)$ arithmetic and the Rainbow submission optimizes the code by using AVX2 *shuffle* and *permute* instructions. In this paper, we show a new optimization that leverages: a) AVX512 architecture; b) the latest processor capabilities Galois Field New Instructions (GF-NI), available on Intel "Ice Lake" processor. We achieved a speedup of $2.43\times/3.13\times/0.64\times$ for key generation/signing/verifying, respectively. We also propose a variation of Rainbow, with equivalent security, using a different representation of $GF(2^8)$. With this variant, we achieve a speedup of $2.44\times/4.7\times/2.1\times$ for key generation/signing/verifying, respectively.

Keywords: Rainbow, Post Quantum Signature scheme, Constant-time implementation, GF-NI

1 Introduction

The potential threat to public key cryptography that large-scale quantum computers pose triggered the National Institute of Standards and Technology (NIST) to launch a standardization process for quantum-resistant crypto-algorithms [12]. This is currently a vibrant research topic. From the 69 Round-1 submission candidates only 17 Key Encapsulation Mechanisms (KEMs) and 9 signature schemes made it to Round-2 of this project. Rainbow [1] is one of these signature schemes and its security relies on the generic (NP-hard) Multivariate Quadratic (MQ) problem. It is a generalization of the Unbalanced Oil and Vinegar (UOV) scheme [11]. Rainbow enjoys a small signature size (64/156/204 bytes for the Ia/IIIc/Vc variants, respectively) in addition, the signing and verifying operations are relatively quick these features make it an appealing candidate. The submission includes several variants, from which we focus here on `IIIc-Classic` (the reasons are explained in Section 2.2).

The KeyGen/Sign/Verify computations of Rainbow rely on multiplications and inversions in $GF(2^8)$. A specific representation of $GF(2^8)$ as the $GF(2^{2^2})$ tower is named in the specification itself and we denote this representation by $\mathbb{F}_{Tower}$. The authors of Rainbow motivate this choice by the ease of a constant-time implementation of the code. However, we point out that any other field representation could also be used, with equivalent security.

In this paper, we explore the potential advantage that can be derived from a judicious use of new processor instructions in order to speedup Rainbow. Specifically, the GF-NI instructions [8] that are available on the latest x86-64 CPUs (microarchitecture codename "Ice Lake") [8]. The use of the GF-NI is demonstrated in [5] for some use-cases. Note that the GF-NI instructions operate over a specific representation of $GF(2^8)$, which we denote by $\mathbb{F}_{\texttt{AES}}$. To leverage these instructions, we first need to calculate the conversion between $\mathbb{F}_{Tower}$ and $\mathbb{F}_{\texttt{AES}}$. Furthermore, if one agrees to define Rainbow over $\mathbb{F}_{\texttt{AES}}$, conversion is no longer needed, and the implementation becomes faster.

The paper is organized as follows. Section 2 describes the new GF-NI instructions and the Rainbow signature scheme. We discuss the details of $\mathbb{F}_{Tower}$ and the conversion from/to $\mathbb{F}_{\texttt{AES}}$ in Section 3. Section 4 discusses different implementation choices for Rainbow. Section 5 describes the experimental setup and Section 6 provides the performance results that we obtain. We conclude with Section 7.

2 Preliminaries

In this paper, we mark hexadecimal notation with a `0x` prefix, and place the LSB on the right-most position. For example, the byte `0x11C` is the binary string `000100011100`. Let X be a string of bits. We use $X[j:i]$, $j \geq i$ to denote the sub-string of X that includes all the bits in the positions between i and j (included). We define $X[i:i] = X[i]$. For example, if $X = 000100011011$ we have $X[4:2] = 110$ and $X[7:7] = X[7] = 0$. Let $\mathbb{F}_{\texttt{AES}}$ be the polynomial representation of $GF(2^8)$ with polynomial reduction $P_{\texttt{AES}} = x^8 + x^4 + x^3 + x + 1$.

2.1 Vectorized GF-NI

GF-NI includes the instructions `VGF2P8MULB`, `VGF2P8AFFINEQB`, and `VGF2P8AFFINEINVQB`. For short, we denote them by `MULB`, `AFFINEB`, and `AFFINEINVB`, respectively. Alg. 1 describes `MULB`. It performs vectorized multiplication in $\mathbb{F}_{\texttt{AES}}$, of $KL = 16/32/64$ 8-bit elements that reside in two 128/256/512-bit registers (the registers are called `xmm`, `ymm`, `zmm`, respectively).

We note that `MULB` can be used for different $GF(2^8)$ representations. This requires some conversions to/from these representations that can be performed with the `AFFINEB`(and `AFFINEINVB`) instruction described in Alg. 2. Here, an affine transformation is $C \cdot x + b$ (or $Cx^{-1} + b$), for some 8×8-bit matrix C that is "vectorized" (duplicated) $KL = 2/4/8$ times and for some 8-bit vectors x and b.

Algorithm 1 MULB instruction

```
    Inputs: SRC1, SRC2 (wide registers)
    Outputs: DST (a wide register)
 1: procedure VGF2P8MULB(SRC1, SRC2)
 2:     for j in 0 to (KL-1) do
 3:         DEST.byte[j] ← GF2P8MULBYTE(SRC1.byte[j], SRC2.byte[j])

 4: procedure GF2P8MULBYTE(s1b, s2b)                    ▷ s1b,s2b (8 bits)
 5:     T[15:0] = 0
 6:     for i in 0 to 7 do
 7:         if s2b[i] then
 8:             T[15:0] = T[15:0] $\oplus (s1b \ll i)$
 9:     for i in 14 downto 8 do
10:         if T[i] then
11:             T[15:0] = T[15:0] $\oplus$ (0x11b $\ll (i-8)$)
12:     return T[7:0]
```

Algorithm 2 AFFINEB and AFFINEINVB instructions

```
    Inputs: S1, S2 (wide registers) imm8 (8 bits)
    Outputs: D (a wide register)
 1: procedure VGF2P8AFFINE[INV]QB(S1, S2)
 2:     for $j$ in 0 to $KL-1$ do
 3:         for $b$ in 0 to 7 do
 4:             $k = 64j$, $q = k + 8b$
 5:             D[$q+7:q$] = [Inv]AffB(S2[$k+63:k$], S1[$q+7:q$], imm8)
 6:     return D[$64KL-1:0$]

 7: procedure [Inv]AffB(s2, s1, imm8)
 8:     for i = 0 to 7 do
 9:         T[i] = parity(s2[8(7-i)+7 : 8(7-i)] & [inv](s1)) $\oplus$ imm8[i]
10:     return T[7:0]
```

2.2 Rainbow

Rainbow is a multivariate-polynomial signature scheme defined over a finite field $\mathbb{F}$. It uses a system of m equations with n variables. Let us fix the number of layers u and to set $v_1, \ldots, v_{u+1} \in \mathbb{Z}$ such that $0 < v_1 < \ldots < v_{u+1} = n$. In addition set $V_i = \{1, \ldots, v_i\}$ and $O_i = \{v_i + 1, \ldots, v_{i+1}\}$, $i = 1, \ldots, u$. Here, $m = n - v_1$, $|V_i| = v_i$ and set $o_i = |O_i|$. The Rainbow operations are as follows.

KeyGen. The private key consists of two invertible affine maps $\mathcal{S} : \mathbb{F}^m \implies \mathbb{F}^m$ and $\mathcal{T} : \mathbb{F}^n \implies \mathbb{F}^n$, and a quadratic central map $\mathcal{F} : \mathbb{F}^n \implies \mathbb{F}^m$, consisting of m multivariate polynomials $f(v_1 + 1), \ldots, f(n)$. The public key is the composed map $\mathcal{P} = \mathcal{S} \cdot \mathcal{F} \cdot \mathcal{T} : \mathbb{F}^n \implies \mathbb{F}^n$ and therefore consists of m quadratic polynomials in the ring $\mathbb{F}[x_1, \dots, x_n]$.

Sign. To sign a message m, compute its hash digest $h = H(m)$ with a hash function[1] $H : \{0, 1\} \Longrightarrow \mathbb{F}^m$. Compute $x = \mathcal{S}^{-1}(h) \in \mathbb{F}^m$ and its pre-image $y \in \mathbb{F}^n$ under the central map $\mathcal{F}$. Then, compute the signature $z = \mathcal{T}^{-1}(y) \in \mathbb{F}^n$.

Verify. To verify a signature $z \in \mathbb{F}^n$ on a message m, calculate $h = H(m)$ and $h' = \mathcal{P}(z) \in F^m$. Accept z if and only if $h' = h$.

Parameters choice. The Rainbow submission proposes three parameter sets in the form $(\mathbb{F}, v_1, o_1, o_2)$ as follows:

– Ia: $(GF(2^4), 32, 32, 32)$ with $m = 64$ equations and $n = 96$ variables. This is designed to meet NIST's security category Level-1/2.
– IIIc: $(GF(2^8), 68, 36, 36)$ with $m = 72$ equations and $n = 140$ variables. This is designed to meet NIST's security category Level-3/4.
– Vc: $(GF(2^8), 92, 48, 48)$ with $m = 96$ equations and $n = 188$ variables. This is designed to meet NIST's security category Level-5/6.

This paper focuses on the IIIc option, and the use of GF-NI to optimize its implementation.

Round-2 Rainbow variants. The Round-2 submission [9] adds two variants ("cyclic" and "compressed") to the Round-1 submission (called "standard"). As stated in [9], the KeyGen and Verify algorithms of cyclic-Rainbow are slower than the KeyGen and Verify of the standard-Rainbow. The compressed-Rainbow is similar to the cyclic-Rainbow and the only difference is that it views the private key as a 512-bit seed. For this reason, it is enough to focus on standard-Rainbow.

3 Finite field representations for Rainbow

From the security viewpoint, the finite field representation used in [9] is immaterial. The specific choice of a tower field ($\mathbb{F}_{Tower}$) targets a constant-time implementation for the field multiplications. This representation views an $GF(2^8)$ element in $GF(2^8)$ as a degree-1 polynomial over $GF(2^4)$ as follows

– $GF(2^2) = GF(2)[e_1] = (e_1^2 + e_1 + 1)$
– $GF(2^4) = GF(2^2)[e_2] = (e_2^2 + e_2 + e_1)$
– $GF(2^8) = GF(2^4)[e_3] = (e_3^2 + e_3 + e_2e_1)$

Here, $GF(2^8)$ multiplication translates to $GF(2^4)$ operations, and these translate to $GF(2^2)$ operations. These can be easily executed in constant-time.

In particular, working in $\mathbb{F}_{Tower}$ is convenient to program on a small device that can only perform $GF(2^2)$ multiplications (in constant-time). However, for

[1] The concrete instantiation of Rainbow IIIc_Classic uses the SHA-384 algorithm as its hash function H.

typical modern server CPUs, different field representations are more appealing. Specifically, the use of $\mathbb{F}_{\texttt{AES}}$ allows for leveraging the `MULB` instruction efficiently.

We outline several Rainbow flavors, based on different field representations.

- Working with $\mathbb{F}_{Tower}$: This requires conversion of inputs/outputs to/from $\mathbb{F}_{\texttt{AES}}$.
- Working with $\mathbb{F}_{\texttt{AES}}$: This does not require any conversion.
- Hybrid 1: The signing party stores the secret key in $\mathbb{F}_{\texttt{AES}}$ and converts the signatures to $\mathbb{F}_{Tower}$. The verifying party stores the public key in $\mathbb{F}_{Tower}$.
- Hybrid 2: The signing party stores the secret key in $\mathbb{F}_{Tower}$ and converts the signatures to $\mathbb{F}_{\texttt{AES}}$. The verifying party stores the public key in $\mathbb{F}_{\texttt{AES}}$.

The optimal choice depends on the compute power of the signing and verifying parties.

Conversion across field representations. All the representations of $GF(2^8)[x]$ are isomorphic. Therefore, it is possible to pass from one representation to another by means of multiplying by an 8×8-bit matrix. The `AFFINEB` instruction is ideal for this purpose, and all that remains is to compute the conversion matrix and its inverse [6].

For our purposes we show how to compute the conversion matrix A from $\mathbb{F}_{Tower}$ to $\mathbb{F}_{\texttt{AES}}$. We first choose a primitive element $\delta \in \mathbb{F}_{Tower}$ (e.g., $\delta = 0xbc$) such that δ is a root of $P_{\texttt{AES}}$ (arithmetic in $\mathbb{F}_{Tower}$). Then, we compute the 8×8 binary matrix

$$A = [\delta^7, \delta^6, \delta^5, \delta^4, \delta^3, \delta^2, \delta^1, \delta^0]$$

with arithmetic in $\mathbb{F}_{Tower}$, where δ^0 is the multiplicative unit (i.e., `0x01`). These are the matrices

$$A = \begin{pmatrix} 1&0&1&1&1&0&1&0 \\ 1&0&1&1&0&1&0&0 \\ 0&0&1&1&1&0&1&0 \\ 1&0&0&1&1&1&1&0 \\ 1&0&0&0&0&1&1&0 \\ 1&0&1&0&0&1&1&0 \\ 1&1&1&1&0&0&0&0 \\ 1&1&1&1&0&0&0&1 \end{pmatrix}$$

Fig. 1. The conversion matrix from $\mathbb{F}_{Tower}$ to the $\mathbb{F}_{\texttt{AES}}$.

For using in `AFFINEB` the matrix A is represented by `0xf1f0a6869e3ab4ba` and the matrix A^{-1} by `0x03349c68700cdea0`.

4 Our implementation

The official Round-2 implementation of Rainbow is found in [10]. It includes several variants one of which, is called "alternative", uses AVX2 (technically C

$$A^{-1} = \begin{pmatrix} 1&0&1&0&0&0&0&0 \\ 1&1&0&1&1&1&1&0 \\ 0&0&0&0&1&1&0&0 \\ 0&1&1&1&0&0&0&0 \\ 0&1&1&0&1&0&0&0 \\ 1&0&0&1&1&1&0&0 \\ 0&0&1&1&0&1&0&0 \\ 0&0&0&0&0&0&1&1 \end{pmatrix}$$

Fig. 2. The conversion matrix from $\mathbb{F}_{\texttt{AES}}$ to $\mathbb{F}_{Tower}$.

Table 1. The performance of different implementations of Rainbow KeyGen/Sign/Verify. The numbers represent cycles count (in thousands), i. e., smaller is better. The code is profiled in the two measurement methodologies explained in Section 5.

	KeyGen (10^6 cycles)		Sign (10^3 cycles)		Verify (10^3 cycles)	
Impl.	Method: Orig [10]	This work	Orig [10]	This work	Orig [10]	This work
Baseline	102	102	699	657	146	106
Baseline w/ CTR DRBG [3]	88.5 (1.16×)	88.5 (1.15×)	732 (0.95×)	675 (0.97×)	152 (0.96×)	106 (1.00×)
Impl1	42.1 (2.43×)	41.7 (2.44×)	264 (2.64×)	210 (3.13×)	226 (0.65×)	166 (0.64×)
Impl2	42.1 (2.43×)	41.7 (2.45×)	172 (4.05×)	142 (4.62×)	103 (1.41×)	56 (1.88×)
Impl3	42 (2.44×)	41.7 (2.45×)	168 (4.17×)	**141**(**4.64**×)	106 (1.38×)	59 (1.81×)
Impl4	42 (2.43×)	41.8 (2.44×)	168 (4.17×)	143 (4.60×)	100 (1.47×)	**50**(**2.13**×)

intrinsic) and is the fastest provided option. This "alternative" code performs fast GF multiplication of $a, b \in GF(2^8)$ by storing or calculating some multiplication tables using the same technique as described in [7,13]. Subsequently, the tables are placed in 256-bit `ymm` registers and the multiplication is computed by 2 shuffle instructions (using `VPSHUFB`), 2 AND instructions and one XOR instruction. By comparison, our implementation (available at [4]) simplifies the code because a multiplication involves only one `MULB` instruction with no tables at all. This also allows us to suggest further optimizations that are based on pipelining the code. Surprisingly, we found that although modern compilers can unroll loops (automatically through a flag), hand written pipelining can still achieve faster results.

IIIc_Classic dedicated code. The Rainbow implementation [10] supports multiple variants of Rainbow mentioned above with portable, SSE-based and AVX2 implementations. Our implementation is dedicated to the IIIc_Classic variant only. This facilitates dedicated optimizations for $o_1 = o_2 = 36$.

Inversion. To perform inversion during Sign, we use `AFFINEINVB` as follows. We set $C = I$ and $b = 0$ so that `AFFINEINVB` computes $I \cdot x^{-1} + b = x^{-1}$. The hex representation of I is `0x0102040810204080`.

Using AVX512. The computations of Rainbow IIIc_classic operate on 72-byte rows, while AVX512 architecture has 512-bit `zmm` registers (64-bytes). Therefore, we use the `AVX512` masking architecture that allows reading/writing only a part of a 512-bit `zmm` register. This saves some copies to/from temporary buffers and simplifies our code.

5 The experimental setup

The platform. For the experiments, we used a Dell XPS 13 7390 2-in-1 laptop. It has a 10th generation Intel®Core^{TM} processor (microarchitecture codename "Ice Lake"[ICL]). The specifics are Intel®Core^{TM} i7-1065G7 CPU 1.30GHz. This platform has 16 GB RAM, 48K L1d cache, 32K L1i cache, 512K L2 cache, and 8MiB L3 cache. For the experiments, we turned off the Intel® Turbo Boost Technology (in order to work with a fixed frequency and measure performance in cycles).

The code. We wrote the code mainly in C with some x86-64 assembly routines. The implementations use the GF-NI as well as other AVX512 instructions. We compiled the code with clang (version 9) in 64-bit mode, using the "-O3" optimization flag and ran it on a Linux OS (Ubuntu 18.04.2 LTS).

Remark 1 *We note that GCC-8/9 also support the GF-NI instructions. However, during our study we identified a bug in GCC that causes incorrect results when using GF-NI. The bug is still present at the time of writing this paper. We reported it in [2] and the proper fix is underway.*

Measurements methodology. The performance reported hereafter is measured in processor cycles (per single core), where lower count is better. We obtain the results using two measurement methodologies.

- The methodology of [10]: Taking the average of 10 runs for the key generation, and the average of 500 runs for the `Sign` and `Verify` operations.
- Our methodology: Every measured function was isolated, run 25 times (warm-up), followed by 100 iterations that were clocked (using the `RDTSC` instruction) and averaged. To minimize the effect of background tasks running on the system, every experiment was repeated 10 times, and the minimum result was recorded.

The difference is in the minimization of background noise on the platform.

Code packages. Our baseline is the official "Alternative" code package that is submitted to the PQC project [10]. This implementation is written with AVX2 instructions. We compare it to our implementations of Rainbow:

- Impl1 - using GF-NI with elements in $\mathbb{F}_{Tower}$.

- Impl2 - using GF-NI with elements in $\mathbb{F}_{\texttt{AES}}$.
- Impl3 - using GF-NI with elements in $\mathbb{F}_{\texttt{AES}}$ compiled with `-funroll-loops` clang flag.
- Impl4 - using GF-NI with elements in $\mathbb{F}_{\texttt{AES}}$ compiled with `-funroll-loops` clang flag and manual pipelining optimization for Verify.

6 Results

Table 1 shows the performance results of our study. The first row shows the baseline, which is compared to our implementations in the subsequent rows. The heaviest operations in the key generation implementation are: a) $GF(2^8)$ multiplications; b) random number generation (noted already in [9]). To help isolating the performance contribution of GF-NI and AVX512 instructions we also replaced the DRBG of [10] with our faster CTR DRBG implementation [3]. It is 1.16× faster.

For the Rainbow flavors that use $\mathbb{F}_{\texttt{AES}}$, we obtain a speedup factor of 4.64× for signing and 2.13× for verifying. Similar speedup is achieved for Rainbow flavors that use $\mathbb{F}_{Tower}$ for signing. However, verifying is slowed down by a factor of (0.65×). This is due to the cost of converting the public key across from $\mathbb{F}_{Tower}$ to $\mathbb{F}_{\texttt{AES}}$. This overhead can be eliminated by simply storing a copy of the public key in $\mathbb{F}_{\texttt{AES}}$ (converting it only once).

The difference between Impl2 and Impl3 is very small. This indicates that adding the `-funroll-loops` compilation flag has a negligible effect (in this case). Note that manual pipelining achieves observable speedups with our measurement methodology (best versus average).

7 Conclusion

This paper shows how the new GF-NI instructions can be used for Rainbow `IIIc_classic`. We achieve speedups of 2.44×, 4.7×, and 2.1× for KeyGen/ Sign/Verify, respectively, when the chosen field is $\mathbb{F}_{\texttt{AES}}$. This makes Rainbow a much more competitive candidate for the PQC standardization. Our results are measured on a laptop platform (the only platform with GF-NI that is currently available), and we expect to see even a stronger effect in future CPUs for server parts. We therefore recommend that the authors of Rainbow [9] consider a flavor of rainbow that operates in $\mathbb{F}_{\texttt{AES}}$ as part of the modifications for Round-3.

The new optimized code of this paper is publicly available in [4].

Acknowledgements

This research was supported by: The Israel Science Foundation (grant No. 1018/16); The Ministry of Science and Technology, Israel, and the Department of Science and Technology, Government of India; The BIU Center for Research in Applied Cryptography and Cyber Security, and the Center for Cyber Law and Policy at the University of Haifa, both in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

References

1. Ding, J., Schmidt, D.: Rainbow, a New Multivariable Polynomial Signature Scheme. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) Applied Cryptography and Network Security. vol. 3531, pp. 164–175. Springer Berlin Heidelberg, Berlin, Heidelberg (2005), https://doi.org/10.1007/11496137_1
2. Drucker, N.: GCC-8 considers the _mm_gf2p8affine_epi64_epi8 intrinsic to be symmetric (December 2019), https://www.mail-archive.com/gcc-bugs@gcc.gnu.org/msg632510.html
3. Drucker, N., Gueron, S.: Fast CTR DRBG for x86 platforms (March 2019), https://github.com/aws-samples/ctr-drbg-with-vector-aes-ni
4. Drucker, N., Gueron, S.: GFNI based Rainbow (IIIc_Classic) (January 2020), https://github.com/aws-samples/rainbow-with-gfni
5. Drucker, N., Gueron, S., Krasnov, V.: The Comeback of Reed Solomon Codes. In: 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH). pp. 125–129 (jun 2018). https://doi.org/10.1109/ARITH.2018.8464690
6. Gueron, S.: International NI: using AES-NI for fast and constant time Chinese SM4 and other ciphers. IACR ePrint (2020)
7. Gueron, S., Kounavis, M.: Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm. Information Processing Letters **110**(14), 549 – 553 (2010). https://doi.org/https://doi.org/10.1016/j.ipl.2010.04.011, http://www.sciencedirect.com/science/article/pii/S002001901000092X
8. Intel: Intel ®64 and IA-32 architectures software developer's manual. System Programming Guide (2019)
9. Jintai, D., Ming-Shing, C., Albrecht, P., Dieter, S., Bo-Yin, Y.: Rainbow (March 2019), https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions
10. Jintai, D., Ming-Shing, C., Albrecht, P., Dieter, S., Bo-Yin, Y.: Rainbow (IIIc_Classic) Alternative code package (March 2019), https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/Rainbow-Round2.zip
11. Kipnis, A., Patarin, J., Goubin, L.: Unbalanced Oil and Vinegar Signature Schemes. In: Stern, J. (ed.) Advances in Cryptology — EUROCRYPT '99. pp. 206–222. Springer Berlin Heidelberg, Berlin, Heidelberg (1999), https://doi.org/10.1007/3-540-48910-X_15
12. NIST: Post-Quantum Cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography (2019), last accessed 20 Aug 2019
13. Plank, J.S., Greenan, K.M., Miller, E.L.: Screaming fast galois field arithmetic using intel SIMD instructions. In: 11th USENIX Conference on File and Storage Technologies (FAST 13). pp. 298–306. USENIX Association, San Jose, CA (2013), https://www.usenix.org/conference/fast13/technical-sessions/presentation/plank_james_simd