Nibbling MAYO

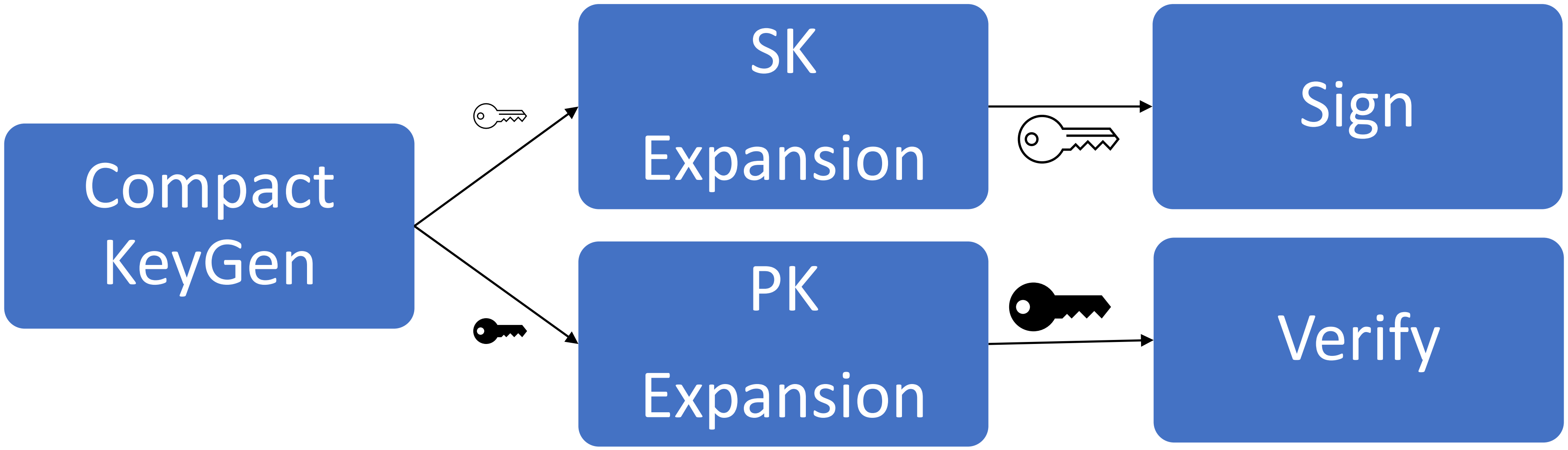
Ward Beullens,
Fabio Campos,
Sofía Celi,
Basil Hess, and
Matthias J. Kannwischer.

Summary of results

We propose new high-speed implementations of MAYO for x86 and Cortex-M4 platforms.

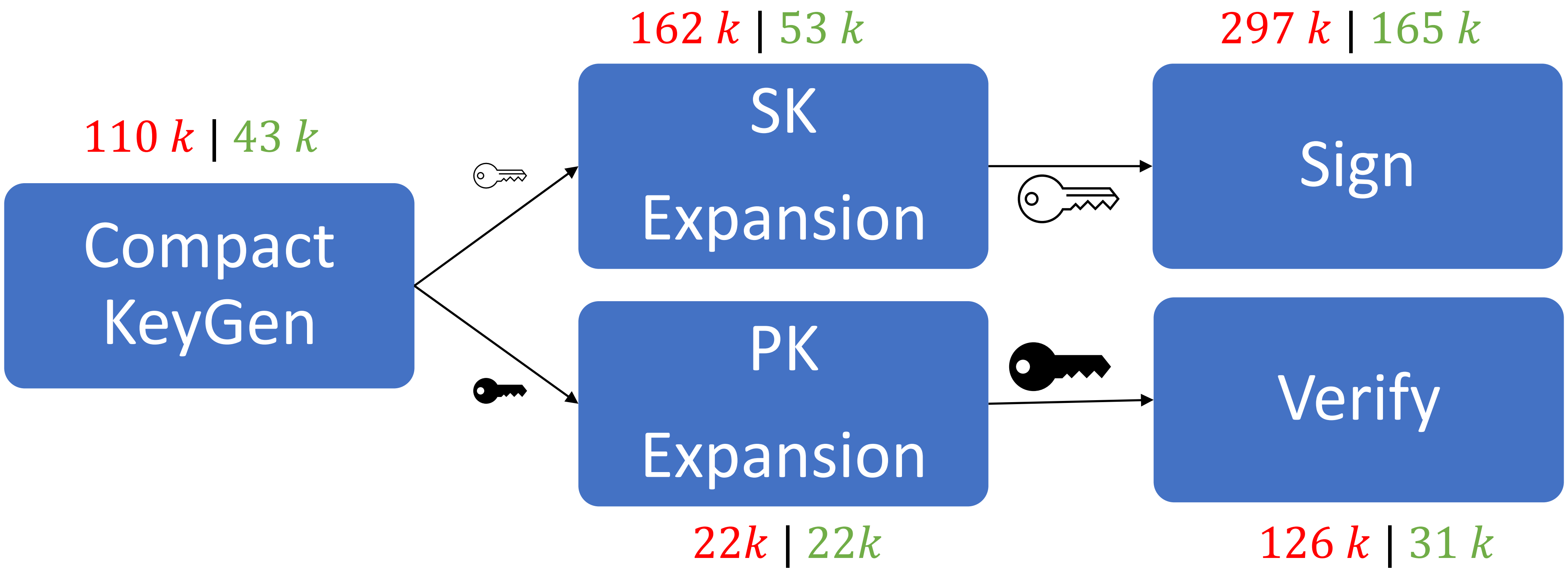
- Our implementations use a new representation of public key. Nibble-sliced instead of bit-sliced.
- Our implementations are based on the “Method of the four Russians” which we found to be more efficient that bitsliced arithmetic.

5-part API



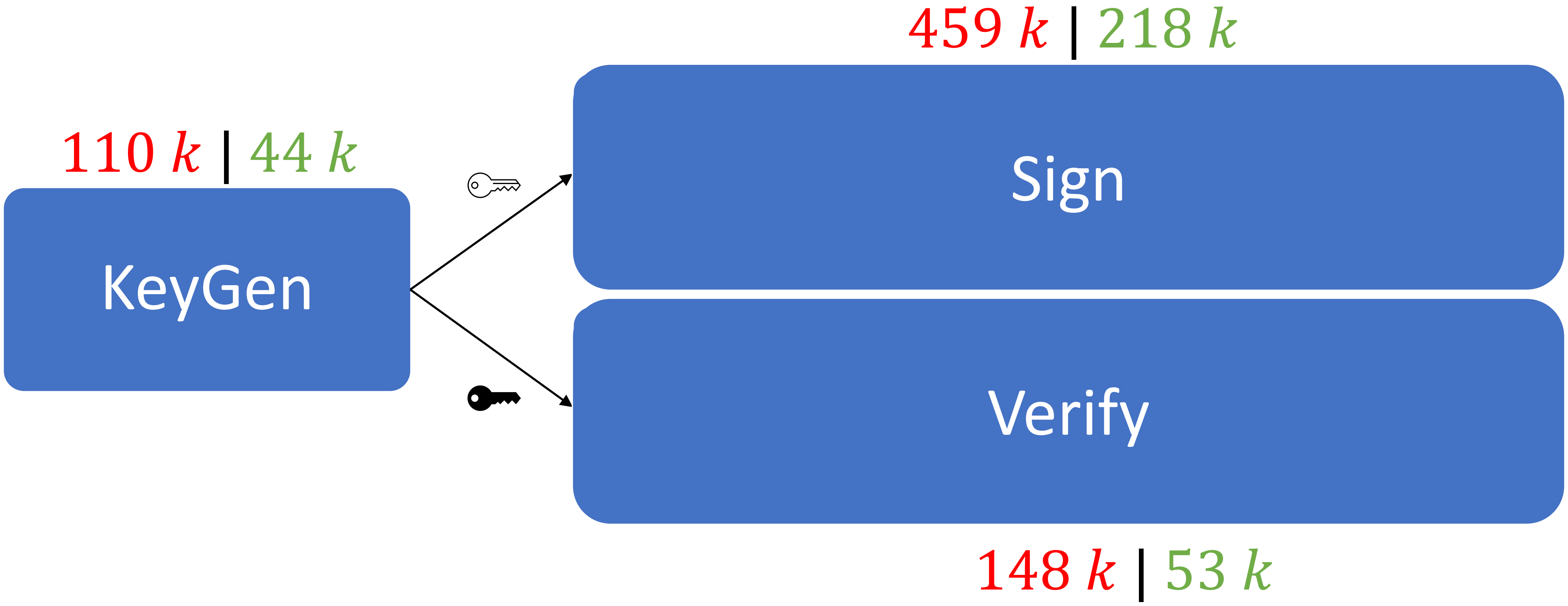
Ice Lake performance MAYO 1

AVX2 + AESNI Bit-sliced | Nibble-sliced implementation



Ice Lake performance MAYO 1

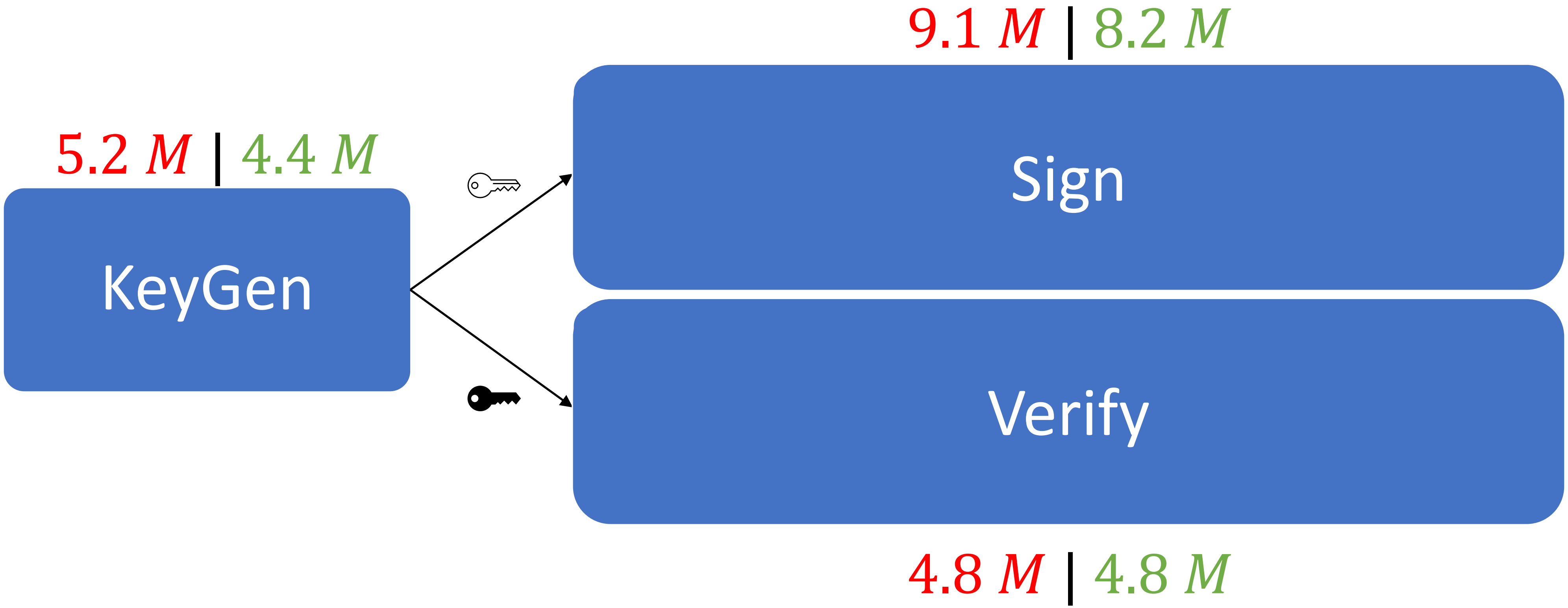
AVX2 + AESNI Bit-sliced | Nibble-sliced implementation



Dilithium2: KeyGen 81 k, Sign 219 k, Verify 79 k

Cortex-M4 performance MAYO 1

ST NUCLEO-L4R5ZI @ 20 MHz Bit-sliced | Nibble-sliced



Dilithium2: KeyGen 1.6 M, Sign 4.0 M, Verify 1.6 M

Overly simplified description of MAYO1

Key Gen:

- Multiply 64 matrices of size 58-by-58 by a 58-by-8 matrix

Sign:

- Multiply 64 matrices of size 58-by-58 by a 58-by-8 matrix
- Multiply 64 matrices of size 58-by-58 by a 58-by-9 matrix
- Solve a system of 64 linear equations in 72 variables

Verify:

- Multiply 64 matrices of size 66-by-66 by a 66-by-9 matrix

Bit-sliced v.s. Nibble-sliced representations

How to represent matrices over $GF(16)$? Representation is irrelevant for security, but important for interoperability and efficient implementation.

$a_0 + a_1x + a_2x^2 + a_3x^3$	$d_0 + d_1x + d_2x^2 + d_3x^3$	$g_0 + g_1x + g_2x^2 + g_3x^3$
$b_0 + b_1x + b_2x^2 + b_3x^3$	$e_0 + e_1x + e_2x^2 + e_3x^3$	$h_0 + h_1x + h_2x^2 + h_3x^3$
$c_0 + c_1x + c_2x^2 + c_3x^3$	$f_0 + f_1x + f_2x^2 + f_3x^3$	$i_0 + i_1x + i_2x^2 + i_3x^3$

(Column major) bit-sliced representation:

$$a_0b_0c_0 \;\; a_1b_1c_1 \;\; a_2b_2c_2 \;\; a_3b_3c_3 \;\; \ldots$$

Good for bit-sliced arithmetic on embedded platforms.

(Column major) nibble-sliced representation:

$$a_0a_1a_2a_3 \;\; b_0b_1b_2b_3 \;\; c_0c_1c_2c_3 \;\; \ldots$$

Good for AVX2 shuffle-based arithmetic on “big” CPUs

Initially, we chose the bit-sliced representation, because on “big” CPUs MAYO is fast enough anyway.

Bit-sliced v.s. Nibble-sliced representations

How to represent matrices over $GF(16)$? Representation is irrelevant for security, but important for interoperability and efficient implementation.

$a_0 + a_1x + a_2x^2 + a_3x^3$	$d_0 + d_1x + d_2x^2 + d_3x^3$	$g_0 + g_1x + g_2x^2 + g_3x^3$
$b_0 + b_1x + b_2x^2 + b_3x^3$	$e_0 + e_1x + e_2x^2 + e_3x^3$	$h_0 + h_1x + h_2x^2 + h_3x^3$
$c_0 + c_1x + c_2x^2 + c_3x^3$	$f_0 + f_1x + f_2x^2 + f_3x^3$	$i_0 + i_1x + i_2x^2 + i_3x^3$

(Column major) bit-sliced representation:

$$a_0b_0c_0 \;\; a_1b_1c_1 \;\; a_2b_2c_2 \;\; a_3b_3c_3 \;\; \ldots$$

Good for bit-sliced arithmetic on embedded platforms.

(Column major) nibble-sliced representation:

$$a_0a_1a_2a_3 \;\; b_0b_1b_2b_3 \;\; c_0c_1c_2c_3 \;\; \ldots$$

Good for AVX2 shuffle-based arithmetic on “big” CPUs

Initially, we chose the bit-sliced representation, because on “big” CPUs MAYO is fast enough anyway.

Contribution of this paper: Nibble-sliced representation is also good for Cortex M4, so we should switch.

Method of the 4 Russians

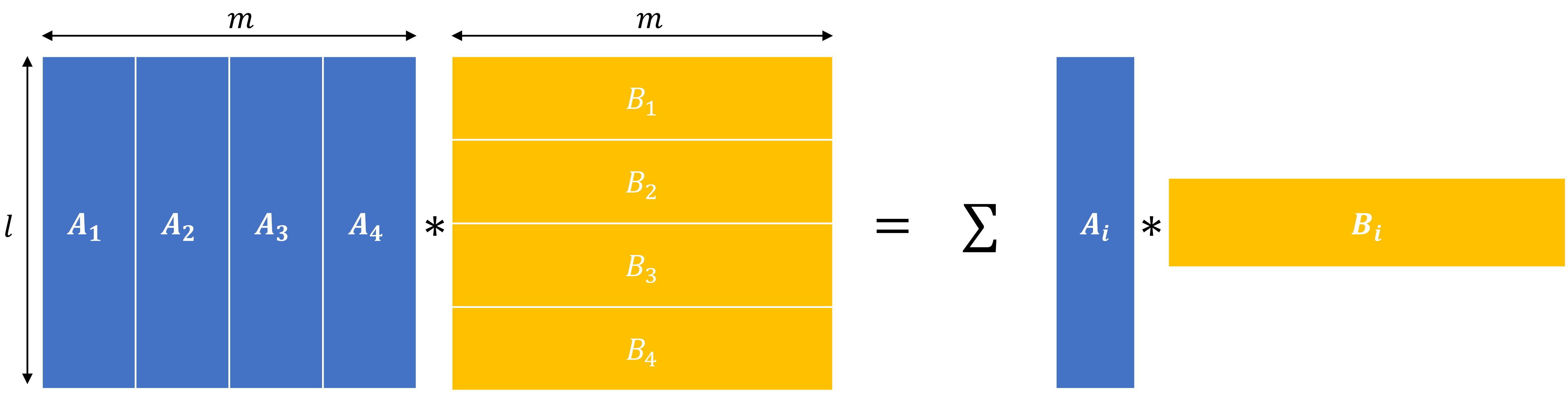
Arlazarov, Dinic, Kronrod, Faradzev (1974)

Method for computing matrix multiplication $A * B$, where A has dimensions l by m, and B has dimensions m by n
Using only $O(\frac{lmn}{\log_q l})$ additions and table lookups.

Step 1: Reduce to to case where A is very tall and narrow

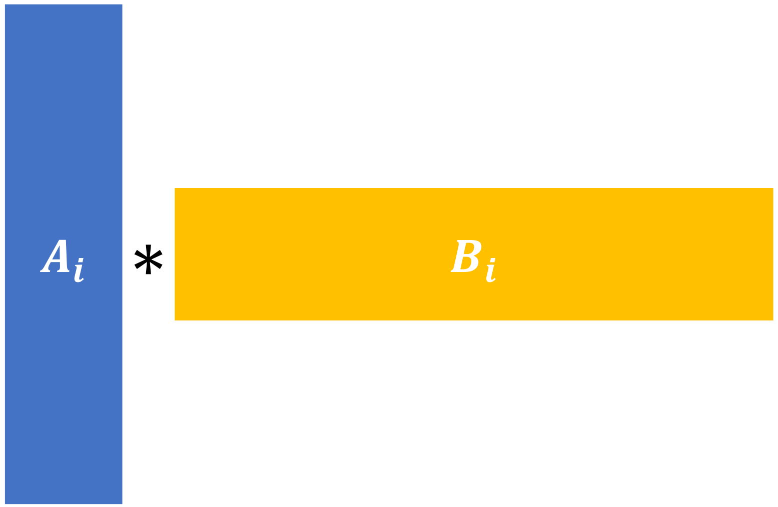
Step 2: Do multiplication by A using table lookups

Step 1: split A and B in narrow strips



A_i have width $t \approx \log_q l$, where l is the height of A, and q is the size of the finite field.

Step 2: Multiplication by table lookup



- Make a table that contains all the linear combinations of rows of B_i.
  (Table has size $q^t n = ln$, requires ln additions to construct)

- Compute $A_i * B_i$ by looking up each row in the table. (l lookups of rows of n elements)

- Cost is $O(ln)$ and needs to be repeated $\frac{m}{\log_q l}$ times, so total cost is $O(\frac{lmn}{\log_q l})$

Results on Cortex M4:

		$(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O}$ Sign		$\mathbf{P}_i^{(1)}\mathbf{V}^{\mathsf{T}}$ Sign		$\mathbf{P}_i^{(1)}\mathbf{O}$ KeyGen	
MAYO$_1$	bitsliced	2 165 337		1 323 797		1 177 752	
	M4R	1 244 009	(1.74 ×)	1 119 136	(1.18 ×)	714 332	(1.65 ×)
MAYO$_2$	bitsliced	5 199 607		629 400		2 830 681	
	M4R	2 906 460	(1.79 ×)	681 081	(0.92 ×)	1 683 616	(1.68 ×)
MAYO$_3$	bitsliced	9 535 835		5 635 495		5 126 000	
	M4R	6 576 258	(1.45 ×)	3 452 417	(1.63 ×)	3 525 668	(1.45 ×)

Conclusion: We can switch to Nibble-based representation and get a nice speedup on embedded platforms, as well as a huge speedup on AVX2 platforms.

Other contributions

- Improved AVX2 shuffle-based matrix multiplication
  New records: 56.5 multiply-and-accumulates / cycle (Skylake)
  78.8 multiply-and-accumulates / cycle (Ice Lake)

- Constant time Gaussian elimination for rectangular matrices.

- Read paper for more …