Johannes de Fine Licht, Christopher A. Pattison, Alexandros Nikolaos Ziogas, David Simmons-Duffin, Torsten Hoefler

We Stuck an Arbitrary Precision Multiplier on an FPGA and it Ran Fast

JOHANNES DE FINE LICHT, CHRISTOPHER A. PATTISON, ALEXANDROS NIKOLAOS ZIOGAS, DAVID SIMMONS-DUFFIN, TORSTEN HOEFLER

~~We Stuck an Arbitrary Precision Multiplier on an FPGA and it Ran Fast~~

Fast Arbitrary Precision Floating Point on FPGA

JOHANNES DE FINE LICHT, CHRISTOPHER A. PATTISON, ALEXANDROS NIKOLAOS ZIOGAS, DAVID SIMMONS-DUFFIN, TORSTEN HOEFLER

~~We Stuck an Arbitrary Precision Multiplier on an FPGA and it Ran Fast~~

Fast Arbitrary Precision Floating Point on FPGA

…how a single FPGA can outperform a 10-node Xeon Cluster in <u>raw throughput</u> ☺

Floating point numbers

1 10000101 11001100110011111010010

$$-1.15202774 \cdot 10^2$$

Floating point numbers

Mantissa/significand

1 10000101 11001100110011111010010

$-1.15202774 \cdot 10^2$

Floating point numbers

Exponent — Mantissa/significand

1 10000101 11001100110011111010010

$$-1.15202774 \cdot 10^{2}$$

Floating point numbers

Sign	Exponent	Mantissa/significand
1	10000101	11001100110011111010010

$$-1.15202774 \cdot 10^{2}$$

Arbitrary precision floating point

Sign	Exponent	Mantissa/significand
1	10000101	11001100110011111010010

Arbitrary precision floating point

Sign	Exponent	Mantissa/significand
1	10000101	1100110011001111101001010010101011101010

Arbitrary precision floating point

(Very, very, very high precision floating point)

Sign	Exponent	Mantissa/significand
1	10000101	1100110011001111101001010010101011101010

Arbitrary precision floating point

(Very, very, very high precision floating point)

Sign	Exponent	Mantissa/significand
1	10000101	1100110011001111101001010010101011101010

In software, we must **partition** the mantissa into **chunks** supported by the ISA:

Arbitrary precision floating point

(Very, very, very high precision floating point)

Sign	Exponent	Mantissa/significand
1	10000101	1100110011001111101001010010101011101010

In software, we must **partition** the mantissa into **chunks** supported by the ISA:

64-bit 1100110011001111101001010010101011001100110011111010010100101010

64-bit 0001010111111001100110011111010010100101010110011001100111110100

64-bit 0111111001100010101111110011001100111110100101001010101100110011

64-bit 0000011011111100110001010111111001100110011111010010100101010110

Arbitrary precision floating point

(Very, very, very high precision floating point)

Sign | Exponent | Mantissa/significand

1 10000101 1100110011001111101001010010101011101010

In software, we must **partition** the mantissa into **chunks** supported by the ISA:

64-bit 1100110011001111101001010010101011001100110011111010010100101010
64-bit 0001010111111001100110011111010010100101010110011001100111110100
64-bit 0111111001100010101111111001100110011111010010100101010110011
64-bit 0000011011111100110001010111111001100110011111010010100101010110

On FPGA it's a bit less clear

When do we need this?

$$1.64071873283215111 3 \cdot 10^2$$

When do we need this?

$$1.6407187328321511 3 \cdot 10^{2}$$

$$- \; 1.0224103738805849 77 \cdot 10^{-9}$$

When do we need this?

$$1.6407187328321511 13 \cdot 10^2$$

$$- \ 1.0224103738805849 77 \cdot 10^{-9}$$

(shifted to align) $$- \ 0.0000000000102241 04 \cdot 10^2$$

When do we need this?

All this information is lost

$$1.64071873283215111 3 \cdot 10^{2}$$

$$- 1.0224103738805849 77 \cdot 10^{-9}$$

(shifted to align) $$- 0.00000000001022410 4 \cdot 10^{2}$$

Why does this matter?

Previous experience shows that high-precision arithmetic is important for accurately solving bootstrap optimization problems. It is not fully understood why. The naïve reason is that derivatives $\partial_z^m \partial_{\overline{z}}^n g_{\Delta,\ell}(z,\overline{z})$ of conformal blocks vary by many orders of magnitude relative to each other as Δ varies. It is not possible to scale away this large variation, and answers may depend on near cancellation of large numbers. In practice, the matrix manipulations in our interior point algorithm "leak" precision, so that the search direction (dx, dX, dy, dY) is less precise than the initial point (x, X, y, Y). By increasing the precision of the underlying arithmetic, the search direction can be made reliable again.

[1] David Simmons-Duffin. "A semidefinite program solver for the conformal bootstrap." *Journal of High Energy Physics* 2015(6).

Why does this matter?

Previous experience shows that high-precision arithmetic is important for accurately solving bootstrap optimization problems. It is not fully understood why. The naïve reason is that derivatives $\partial_z^m \partial_{\bar{z}}^n g_{\Delta,\ell}(z,\bar{z})$ of conformal blocks vary by many orders of magnitude relative to each other as Δ varies. It is not possible to scale away this large variation, and answers may depend on near cancellation of large numbers. In practice, the matrix manipulations in our interior point algorithm "leak" precision, so that the search direction (dx, dX, dy, dY) is less precise than the initial point (x, X, y, Y). By increasing the precision of the underlying arithmetic, the search direction can be made reliable again.

[1] David Simmons-Duffin. "A semidefinite program solver for the conformal bootstrap." *Journal of High Energy Physics* 2015(6).

GMP and MPFR

GMP

«Arithmetic without limitations»

The GNU Multiple Precision Arithmetic Library

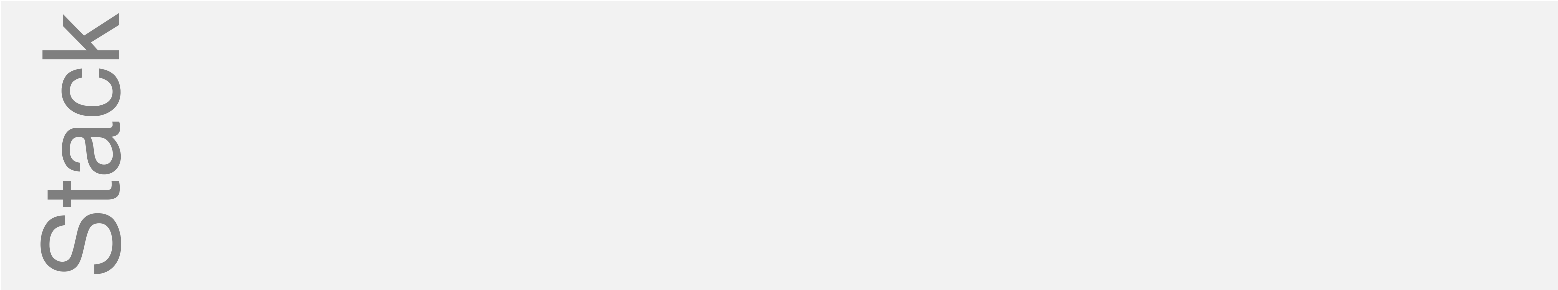
Last modified: 2021-11-19

MPFR

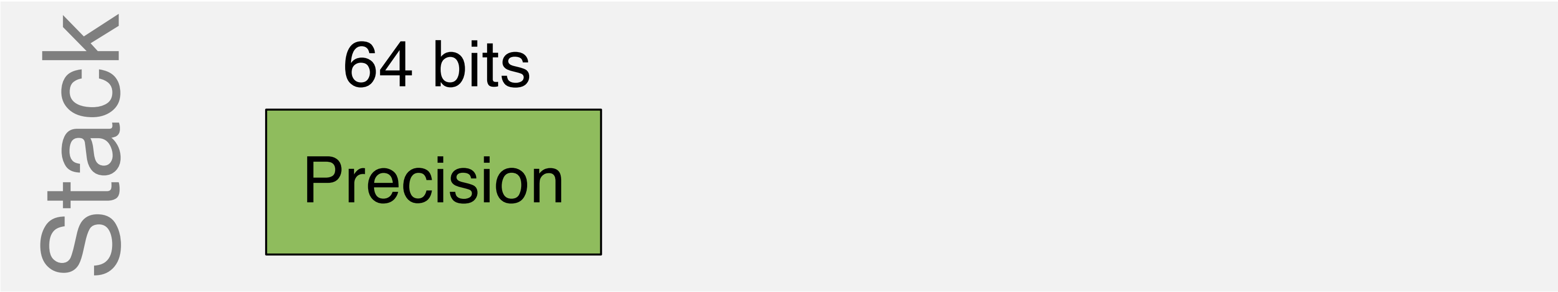
The *GNU MPFR* Library

(arbitrary precision arithmetic == multiple precision arithmetic == bignum arithmetic)

MPFR representation

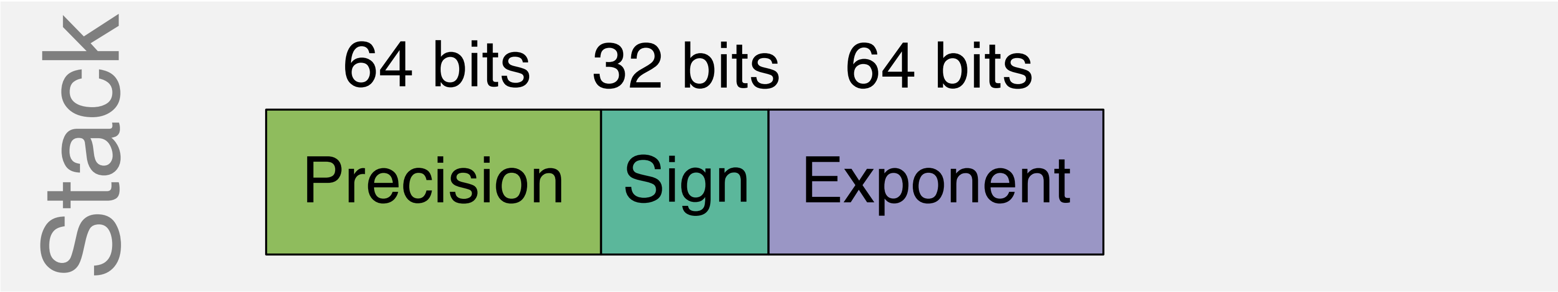


MPFR representation

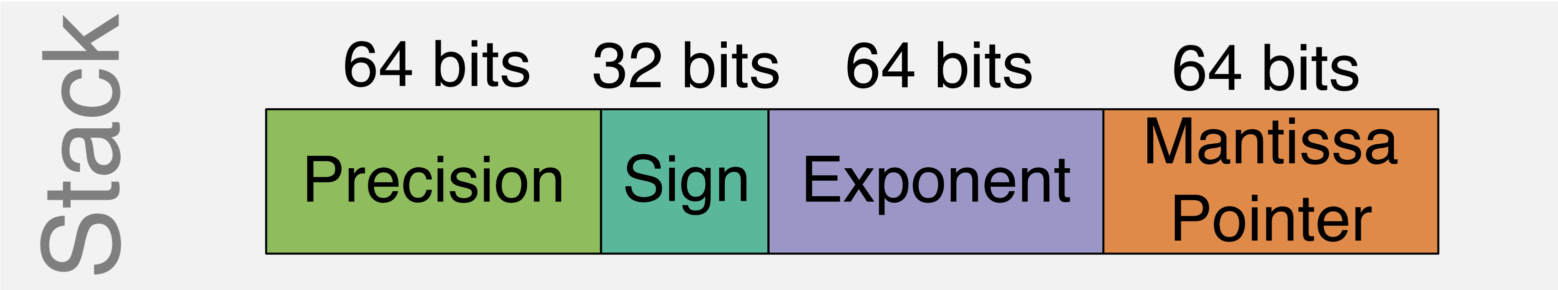


MPFR representation

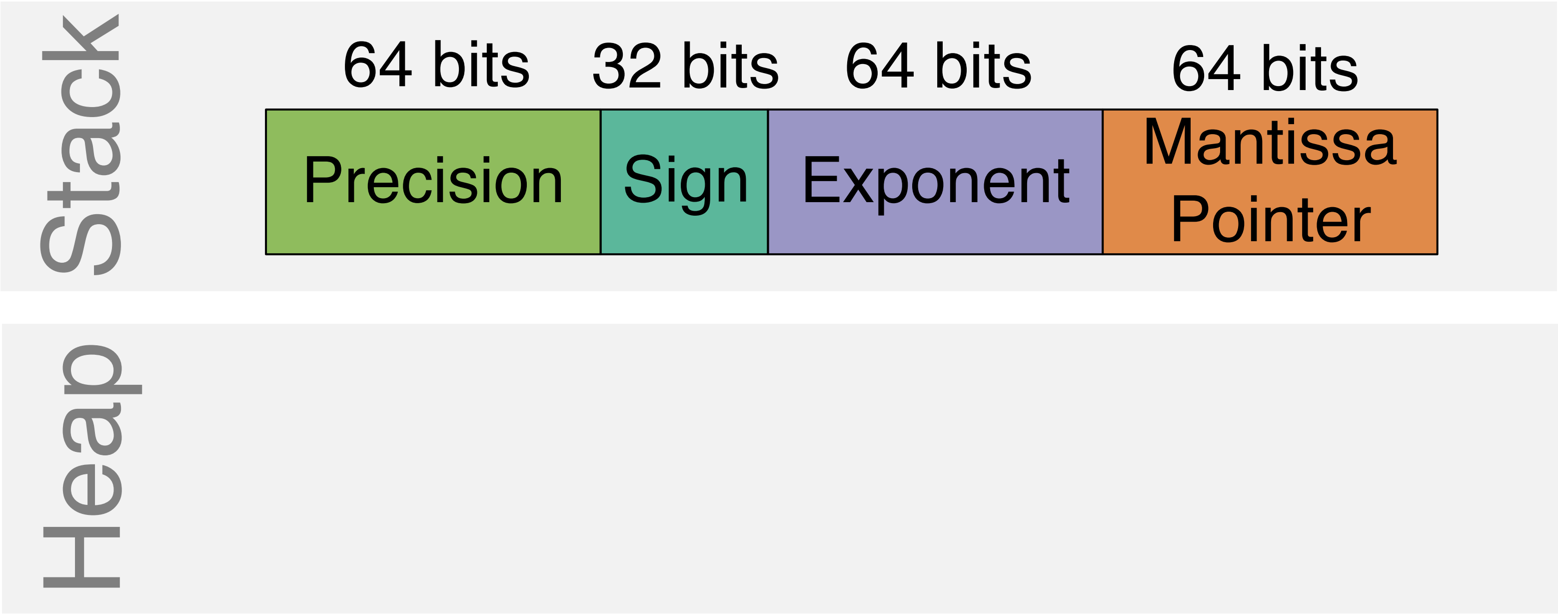
MPFR representation



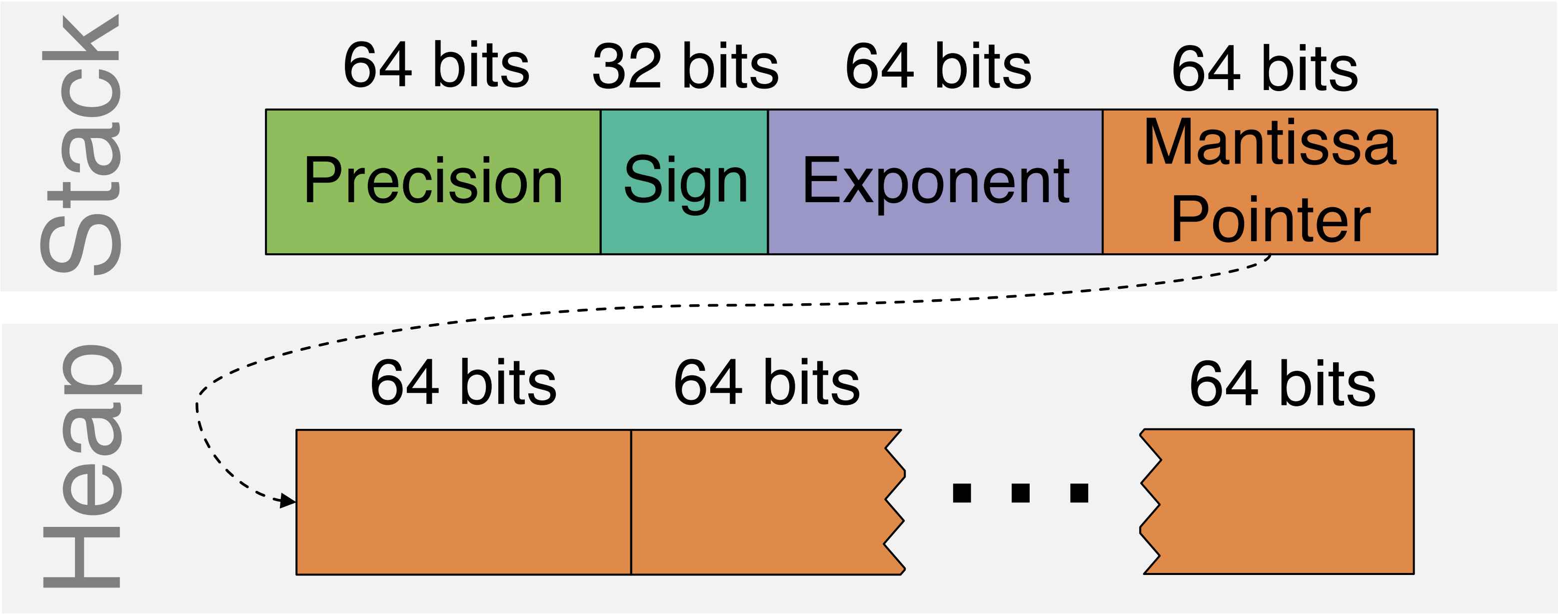
MPFR representation



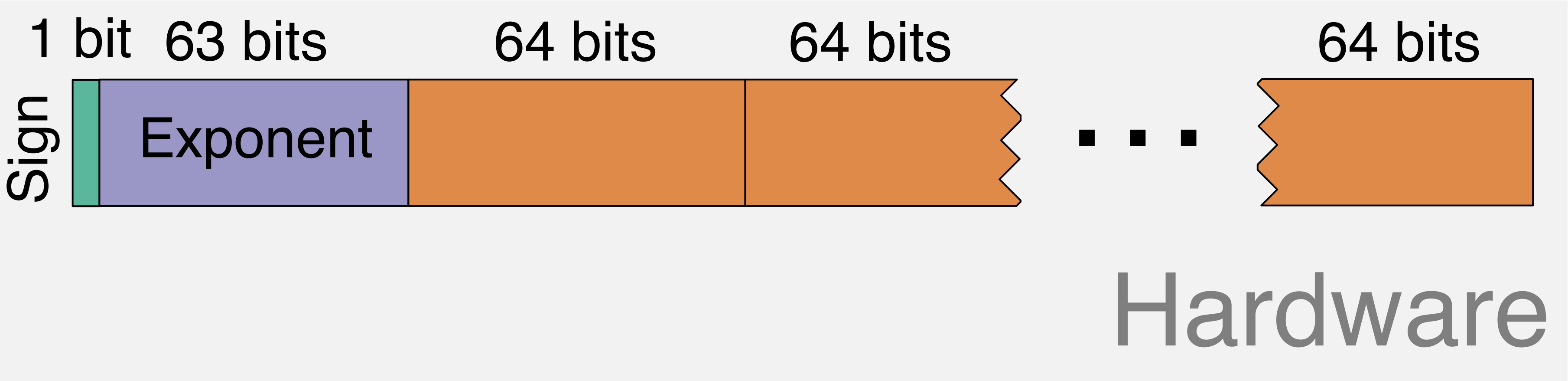
MPFR representation



MPFR representation



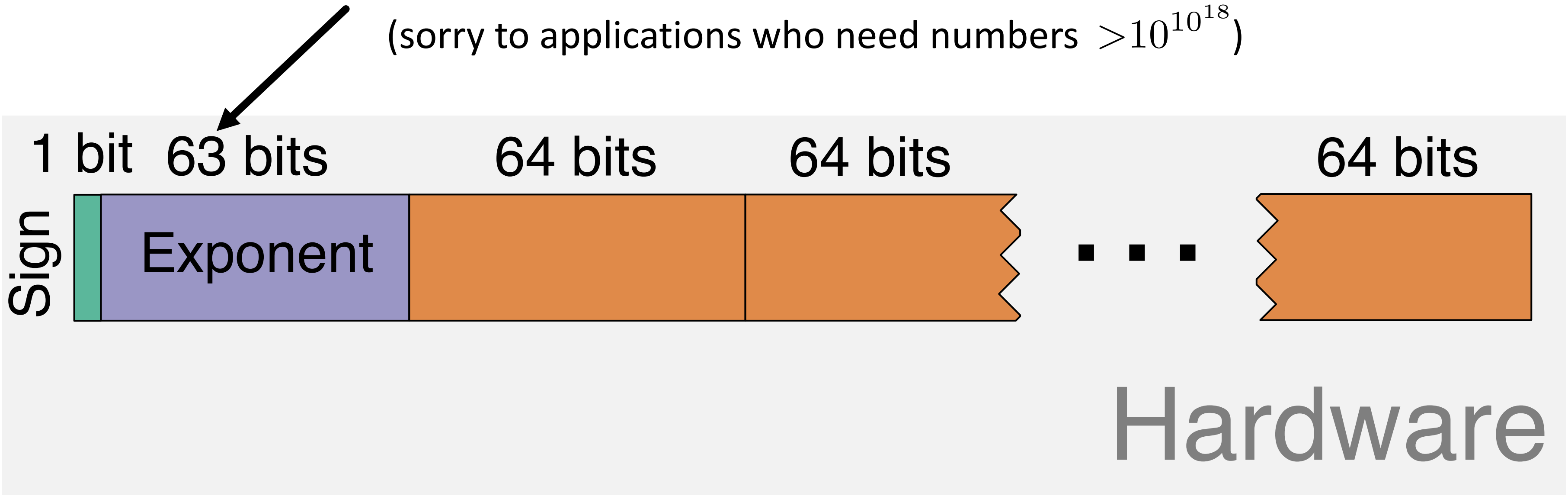
Hardware representation



Hardware representation

We borrow one bit for the sign

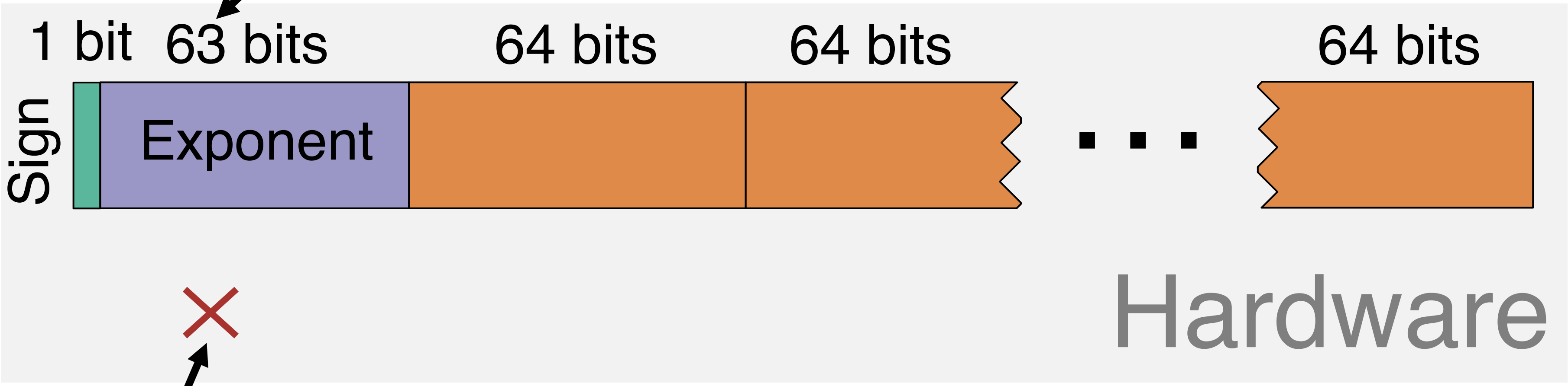
(sorry to applications who need numbers $>10^{10^{18}}$)



Hardware representation

We borrow one bit for the sign

(sorry to applications who need numbers $>10^{10^{18}}$)

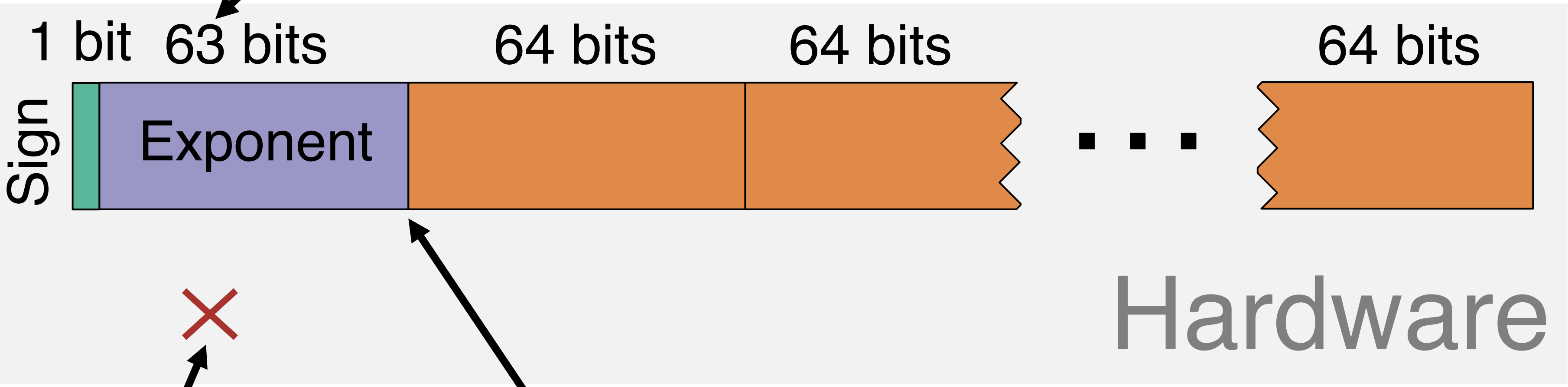


No precision field: Fixed at compile-time

Hardware representation

We borrow one bit for the sign

(sorry to applications who need numbers $>10^{10^{18}}$)



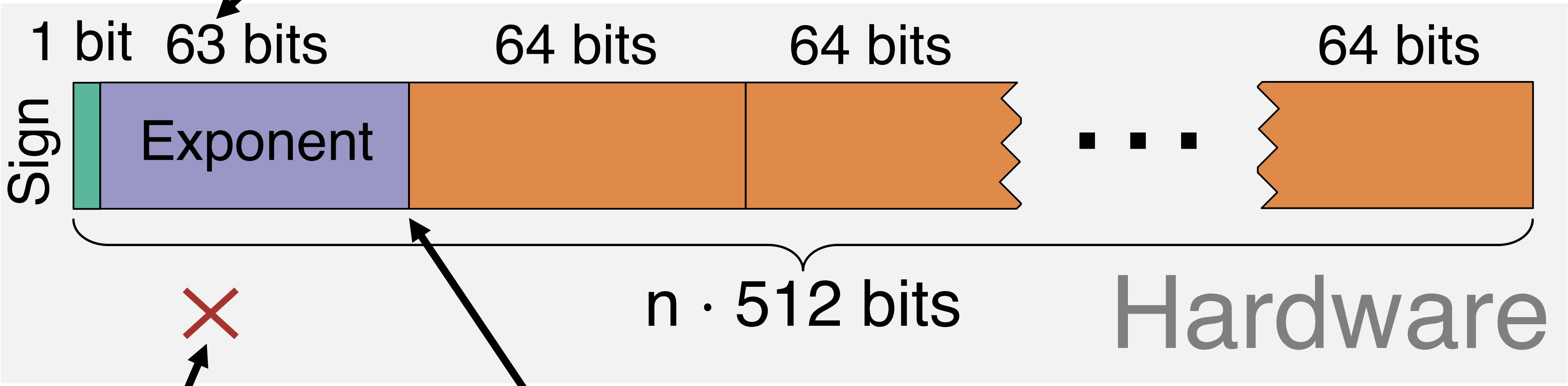
Mantissa packed tightly with number

No precision field: Fixed at compile-time

Hardware representation

We borrow one bit for the sign

(sorry to applications who need numbers $>10^{10^{18}}$)



Mantissa packed tightly with number

No precision field: Fixed at compile-time

Addition and multiplication

Addition and multiplication

```
11 10e4 + 10 01e2
```

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

```
  11 10e4        11 10e4
+ 10 01e2  ⇨   + 00 10e4
```

Addition and multiplication

```
11 10e4 + 10 01e2
```

1. Shift by difference in exponent

```
  11 10e4          11 10e4
+ 10 01e2   ⇒    + 00 10e4
```

2. Add mantissas as integers

```
     11 10
+    00 10
  01 00 00
```

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

```
  11 10e4          11 10e4
+ 10 01e2   ⇨    + 00 10e4
```

2. Add mantissas as integers

```
     11 10
+    00 10
  01 00 00
```

3. On overflow, shift and increment exponent

01 00 00e4 ⇨ 10 00e5

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

```
  11 10e4        11 10e4
+ 10 01e2   ⟹  + 00 10e4
```

2. Add mantissas as integers

```
   11 10
+  00 10
01 00 00
```

3. On overflow, shift and increment exponent

01 00 00e4 ⟹ 10 00e5

Linear in the number of bits.

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

```
  11 10e4          11 10e4
+ 10 01e2   ⟹   + 00 10e4
```

Linear in the number of bits.

2. Add mantissas as integers

```
    11 10
+   00 10
---------
 01 00 00
```

3. On overflow, shift and increment exponent

01 00 00e4 ⟹ 10 00e5

11 10e4 x 10 01e2

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

```
   11 10e4        11 10e4
 + 10 01e2   ⇒  + 00 10e4
```

2. Add mantissas as integers

```
   11 10
 + 00 10
 -------
01 00 00
```

3. On overflow, shift and increment exponent

01 00 00e4 ⇒ 10 00e5

Linear in the number of bits.

11 10e4 x 10 01e2

1. Multiply mantissas as integers

```
         11 10
       x 10 01
   -----------
         11 10
       0 00 00
      00 00 00
   01 11 00 00
   -----------
   01 11 11 10
```

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

```
  11 10e4        11 10e4
+ 10 01e2  ⇒   + 00 10e4
```

2. Add mantissas as integers

```
   11 10
 + 00 10
01 00 00
```

3. On overflow, shift and increment exponent

01 00 00e4 ⇒ 10 00e5

Linear in the number of bits.

11 10e4 x 10 01e2

1. Multiply mantissas as integers

```
       11 10
     x 10 01
    --------
       11 10
     0 00 00
    00 00 00
 01 11 00 00
 -----------
 01 11 11 10
```

2. Drop lower bits

01 11 11 10 ⇒ 01 11

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

```
  11 10e4   ⇒    11 10e4
+ 10 01e2      + 00 10e4
```

2. Add mantissas as integers

```
  11 10
+ 00 10
--------
01 00 00
```

3. On overflow, shift and increment exponent

01 00 00e4 ⇒ 10 00e5

Linear in the number of bits.

11 10e4 x 10 01e2

1. Multiply mantissas as integers

```
         11 10
       x 10 01
  ------------
         11 10
       0 00 00
      00 00 00
   01 11 00 00
  ------------
   01 11 11 10
```

2. Drop lower bits

01 11 11 10 ⇒ 01 11

3. Add exponents and XOR sign

01 11e6

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

```
  11 10e4        11 10e4
+ 10 01e2  ⇒   + 00 10e4
```

Linear in the number of bits.

2. Add mantissas as integers

```
  11 10
+ 00 10
01 00 00
```

3. On overflow, shift and increment exponent

01 00 00e4 ⇒ 10 00e5

11 10e4 x 10 01e2

1. Multiply mantissas as integers

```
         11 10
       x 10 01
         11 10
      0 00 00
     00 00 00
  01 11 00 00
  01 11 11 10
```

2. Drop lower bits

01 11 11 10 ⇒ 01 11

3. Add exponents and XOR sign

01 11e6

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

```
  11 10e4        11 10e4
+ 10 01e2   ⇒  + 00 10e4
```

Linear in the number of bits.

2. Add mantissas as integers

```
  11 10
+ 00 10
01 00 00
```

3. On overflow, shift and increment exponent

01 00 00e4 ⇒ 10 00e5

11 10e4 x 10 01e2

1. Multiply mantissas as integers

```
         11 10
       x 10 01
         11 10
      00 00
   00 00
01 11 00 00
01 11 11 10
```

Super-linear in the number of bits.

2. Drop lower bits

01 11 11 10 ⇒ 01 11

3. Add exponents and XOR sign

01 11e6

Addition and multiplication

11 10e4 + 10 01e2

1. Shift by difference in exponent

2. Add mantissas as integers

3. On overflow, shift and increment exponent

01 00 00e4 → 10 00e5

Linear in the number of bits.

11 10e4 x 10 01e2

1. Multiply mantissas as integers

2. Drop lower bits

Super-linear in the number of bits.

In a fully pipelined design:
Super-linear **resource utilization**.

Arbitrary precision multiplication

$$a \cdot b$$

Arbitrary precision multiplication

$$a_0|a_1 \cdot b_0|b_1$$

Arbitrary precision multiplication

$$\mathrm{a}_0|\mathrm{a}_1 \cdot \mathrm{b}_0|\mathrm{b}_1$$

$$a \cdot b = 2^n a_1 b_1 + 2^{\frac{n}{2}}(a_1 b_0 + a_0 b_1) + a_0 b_0$$

Arbitrary precision multiplication

$$\mathrm{a}_0|\mathrm{a}_1 \cdot \mathrm{b}_0|\mathrm{b}_1$$

$$a \cdot b = 2^n a_1 b_1 + 2^{\frac{n}{2}}(a_1 b_0 + a_0 b_1) + a_0 b_0$$

right-shift by n

right-shift by n/2

Karatsuba arbitrary precision multiplication

$$\mathrm{a}_0|\mathrm{a}_1 \cdot \mathrm{b}_0|\mathrm{b}_1$$

$$a \cdot b = 2^n a_1 b_1 + 2^{\frac{n}{2}}(a_1 b_0 + a_0 b_1) + a_0 b_0$$

Karatsuba arbitrary precision multiplication

$$\mathrm{a}_0|\mathrm{a}_1 \cdot \mathrm{b}_0|\mathrm{b}_1$$

$$a \cdot b = 2^n \boxed{a_1 b_1} + 2^{\frac{n}{2}}(a_1 b_0 + a_0 b_1) + \boxed{a_0 b_0}$$

Karatsuba arbitrary precision multiplication

$$\mathrm{a}_0|\mathrm{a}_1 \cdot \mathrm{b}_0|\mathrm{b}_1$$

$$a \cdot b = 2^n \boxed{a_1 b_1} + 2^{\frac{n}{2}} \boxed{(a_1 b_0 + a_0 b_1)} + \boxed{a_0 b_0}$$

$$(a_1 + a_0)(b_1 + b_0) = a_1 b_1 + a_1 b_0 + a_0 b_1 + a_0 b_0$$

Karatsuba arbitrary precision multiplication

$$\mathrm{a}_0|\mathrm{a}_1 \cdot \mathrm{b}_0|\mathrm{b}_1$$

$$a \cdot b = 2^n \boxed{a_1 b_1} + 2^{\frac{n}{2}} \boxed{(a_1 b_0 + a_0 b_1)} + \boxed{a_0 b_0}$$

$$(a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0 = a_1 b_0 + a_0 b_1$$

Karatsuba arbitrary precision multiplication

$$\mathrm{a}_0|\mathrm{a}_1 \cdot \mathrm{b}_0|\mathrm{b}_1$$

$$a \cdot b = 2^n a_1 b_1 + 2^{\frac{n}{2}} (a_1 b_0 + a_0 b_1) + a_0 b_0$$

$$(a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0 = a_1 b_0 + a_0 b_1$$

Karatsuba arbitrary precision multiplication

$$\mathrm{a_0|a_1} \cdot \mathrm{b_0|b_1}$$

$$a \cdot b = 2^n \boxed{a_1 b_1} + 2^{\frac{n}{2}} \boxed{(a_1 b_0 + a_0 b_1)} + \boxed{a_0 b_0}$$

$$(a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0 = \boxed{a_1 b_0 + a_0 b_1}$$

✅ ✅

Karatsuba arbitrary precision multiplication

$$\mathrm{a}_0|\mathrm{a}_1 \cdot \mathrm{b}_0|\mathrm{b}_1$$

$$a \cdot b = 2^n a_1 b_1 + 2^{\frac{n}{2}} (a_1 b_0 + a_0 b_1) + a_0 b_0$$

$$(a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0 = a_1 b_0 + a_0 b_1$$

✅ ✅

Karatsuba arbitrary precision multiplication

We went **from 4 to 3** multiplications!

$$a_0|a_1 \cdot b_0|b_1$$

$$a \cdot b = 2^n \boxed{a_1 b_1} + 2^{\frac{n}{2}} \boxed{(a_1 b_0 + a_0 b_1)} + \boxed{a_0 b_0}$$

$$\boxed{(a_1 + a_0)(b_1 + b_0)} - a_1 b_1 - a_0 b_0 = \boxed{a_1 b_0 + a_0 b_1}$$

✅ ✅

Karatsuba arbitrary precision multiplication

We went **from 4 to 3** multiplications!

$$a_0|a_1 \cdot b_0|b_1$$

Instead of $O(n^2)$ we now have $O(n^{\log_2 3}) \approx O(n^{1.58})$!

$$(a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0 = a_1 b_0 + a_0 b_1$$

✅ ✅

Karatsuba in HLS

```
template <int bits>
auto Karatsuba(ap_uint<bits> const &a, ap_uint<bits> const &b) ->
    typename std::enable_if<(bits > MULT_BASE_BITS), ap_uint<2*bits >>::type {

  using Full = ap_uint<bits>;
  using Half = ap_uint<bits / 2>;

  Half a0 = a(bits/2-1, 0); Half a1 = a(bits-1, bits/2);
  Half b0 = b(bits/2-1, 0); Half b1 = b(bits-1, bits/2);

  Full c0 = Karatsuba<bits / 2>(a0, b0); // Recurse
  Full c2 = Karatsuba<bits / 2>(a1, b1); // Recurse
  // ...compute |a1-a0| and |b1-b0|...
  Full c1 = Karatsuba<bits / 2>(a1_a0, b1_b0); // Recurse

  // ...combine all contributions and return...
}

template <int bits>
auto Karatsuba(ap_uint<bits> const &a, ap_uint<bits> const &b) ->
    typename std::enable_if<(bits <= MULT_BASE_BITS), ap_uint<2*bits>>::type {

  return a * b; // Bottom out using naive mult
}
```

Karatsuba in HLS

```
template <int bits>
auto Karatsuba(ap_uint<bits> const &a, ap_uint<bits> const &b) ->
    typename std::enable_if<(bits > MULT_BASE_BITS), ap_uint<2*bits >>::type {

  using Full = ap_uint<bits>;
  using Half = ap_uint<bits / 2>;

  Half a0 = a(bits/2-1, 0); Half a1 = a(bits-1, bits/2);
  Half b0 = b(bits/2-1, 0); Half b1 = b(bits-1, bits/2);

  Full c0 = Karatsuba<bits / 2>(a0, b0); // Recurse
  Full c2 = Karatsuba<bits / 2>(a1, b1); // Recurse
  // ...compute |a1-a0| and |b1-b0|...
  Full c1 = Karatsuba<bits / 2>(a1_a0, b1_b0); // Recurse

  // ...combine all contributions and return...
}

template <int bits>
auto Karatsuba(ap_uint<bits> const &a, ap_uint<bits> const &b) ->
    typename std::enable_if<(bits <= MULT_BASE_BITS), ap_uint<2*bits>>::type {

  return a * b; // Bottom out using naive mult
}
```

Karatsuba in HLS

```
template <int bits>
auto Karatsuba(ap_uint<bits> const &a, ap_uint<bits> const &b) ->
    typename std::enable_if<(bits > MULT_BASE_BITS), ap_uint<2*bits >>::type {

  using Full = ap_uint<bits>;
  using Half = ap_uint<bits / 2>;

  Half a0 = a(bits/2-1, 0); Half a1 = a(bits-1, bits/2);
  Half b0 = b(bits/2-1, 0); Half b1 = b(bits-1, bits/2);

  Full c0 = Karatsuba<bits / 2>(a0, b0); // Recurse
  Full c2 = Karatsuba<bits / 2>(a1, b1); // Recurse
  // ...compute |a1-a0| and |b1-b0|...
  Full c1 = Karatsuba<bits / 2>(a1_a0, b1_b0); // Recurse

  // ...combine all contributions and return...
}

template <int bits>
auto Karatsuba(ap_uint<bits> const &a, ap_uint<bits> const &b) ->
    typename std::enable_if<(bits <= MULT_BASE_BITS), ap_uint<2*bits>>::type {

  return a * b; // Bottom out using naive mult
}
```

Karatsuba in HLS

```
template <int bits>
auto Karatsuba(ap_uint<bits> const &a, ap_uint<bits> const &b) ->
    typename std::enable_if<(bits > MULT_BASE_BITS), ap_uint<2*bits >>::type {

  using Full = ap_uint<bits>;
  using Half = ap_uint<bits / 2>;

  Half a0 = a(bits/2-1, 0); Half a1 = a(bits-1, bits/2);
  Half b0 = b(bits/2-1, 0); Half b1 = b(bits-1, bits/2);

  Full c0 = Karatsuba<bits / 2>(a0, b0); // Recurse
  Full c2 = Karatsuba<bits / 2>(a1, b1); // Recurse
  // ...compute |a1-a0| and |b1-b0|...
  Full c1 = Karatsuba<bits / 2>(a1_a0, b1_b0); // Recurse

  // ...combine all contributions and return...
}

template <int bits>
auto Karatsuba(ap_uint<bits> const &a, ap_uint<bits> const &b) ->
    typename std::enable_if<(bits <= MULT_BASE_BITS), ap_uint<2*bits>>::type {

  return a * b; // Bottom out using naive mult
}
```

Fully pipelined

Multiplier performance

512-bit (448-bit)

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	2100 MHz	-	-	490 MOp/s	1.0×	36×
FPGA 1 CU	456 MHz	16%	4%	451 MOp/s	0.9×	33.1×

Xilinx Alveo U250 vs. CPU node with 2× Intel Xeon E5-2695 v4 18-core CPUs in a dual-socket configuration (36 cores per node)

Multiplier performance

512-bit (448-bit)

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	2100 MHz	-	-	490 MOp/s	1.0×	36×
FPGA 1 CU	456 MHz	16%	4%	451 MOp/s	0.9×	33.1×
FPGA 4 CUs	376 MHz	37%	14%	1502 MOp/s	3.1×	110.3×
FPGA 8 CUs	300 MHz	48%	28%	2401 MOp/s	4.9×	176.3×
FPGA 12 CUs	300 MHz	62%	42%	3595 MOp/s	7.3×	264.0×
FPGA 16 CUs	300 MHz	75%	56%	4784 MOp/s	9.8×	351.3×

Xilinx Alveo U250 vs. CPU node with 2× Intel Xeon E5-2695 v4 18-core CPUs in a dual-socket configuration (36 cores per node)

Multiplier performance

512-bit (448-bit)

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	2100 MHz	-	-	490 MOp/s	1.0×	36×
FPGA 1 CU	456 MHz	16%	4%	451 MOp/s	0.9×	33.1×
FPGA 4 CUs	376 MHz	37%	14%	1502 MOp/s	3.1×	110.3×
FPGA 8 CUs	300 MHz	48%	28%	2401 MOp/s	4.9×	176.3×
FPGA 12 CUs	300 MHz	62%	42%	3595 MOp/s	7.3×	264.0×
FPGA 16 CUs	300 MHz	75%	56%	4784 MOp/s	9.8×	351.3×

Xilinx Alveo U250 vs. CPU node with 2× Intel Xeon E5-2695 v4 18-core CPUs in a dual-socket configuration (36 cores per node)

Multiplier performance

512-bit (448-bit)

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	2100 MHz	-	-	490 MOp/s	1.0×	36×
FPGA 1 CU	456 MHz	16%	4%	451 MOp/s	0.9×	33.1×
FPGA 4 CUs	376 MHz	37%	14%	1502 MOp/s	3.1×	110.3×
FPGA 8 CUs	300 MHz	48%	28%	2401 MOp/s	4.9×	176.3×
FPGA 12 CUs	300 MHz	62%	42%	3595 MOp/s	7.3×	264.0×
FPGA 16 CUs	300 MHz	75%	56%	4784 MOp/s	9.8×	351.3×

1024-bit (960-bit)

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	-	-	-	227 MOp/s	1×	36×
FPGA 1 CU	361 MHz	27%	8%	361 MOp/s	1.6×	57.3×
FPGA 4 CUs	293 MHz	58%	42%	1202 MOp/s	5.3×	190.9×

Xilinx Alveo U250 vs. CPU node with 2× Intel Xeon E5-2695 v4 18-core CPUs in a dual-socket configuration (36 cores per node)

Multiplier performance

512-bit (448-bit)

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	2100 MHz	-	-	490 MOp/s	1.0×	36×
FPGA 1 CU	456 MHz	16%	4%	451 MOp/s	0.9×	33.1×
FPGA 4 CUs	376 MHz	37%	14%	1502 MOp/s	3.1×	110.3×
FPGA 8 CUs	300 MHz	48%	28%	2401 MOp/s	4.9×	176.3×
FPGA 12 CUs	300 MHz	62%	42%	3595 MOp/s	7.3×	264.0×
FPGA 16 CUs	300 MHz	75%	56%	4784 MOp/s	9.8×	351.3×

1024-bit (960-bit)

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	-	-	-	227 MOp/s	1×	36×
FPGA 1 CU	361 MHz	27%	8%	361 MOp/s	1.6×	57.3×
FPGA 4 CUs	293 MHz	58%	42%	1202 MOp/s	5.3×	190.9×

Xilinx Alveo U250 vs. CPU node with 2× Intel Xeon E5-2695 v4 18-core CPUs in a dual-socket configuration (36 cores per node)

Multiplier performance

…48-bit)

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	2100 MHz	-	-	490 MOp/s	1.0×	36×
FPGA 1 CU	456 MHz	16%	4%	451 MOp/s	0.9×	33.1×

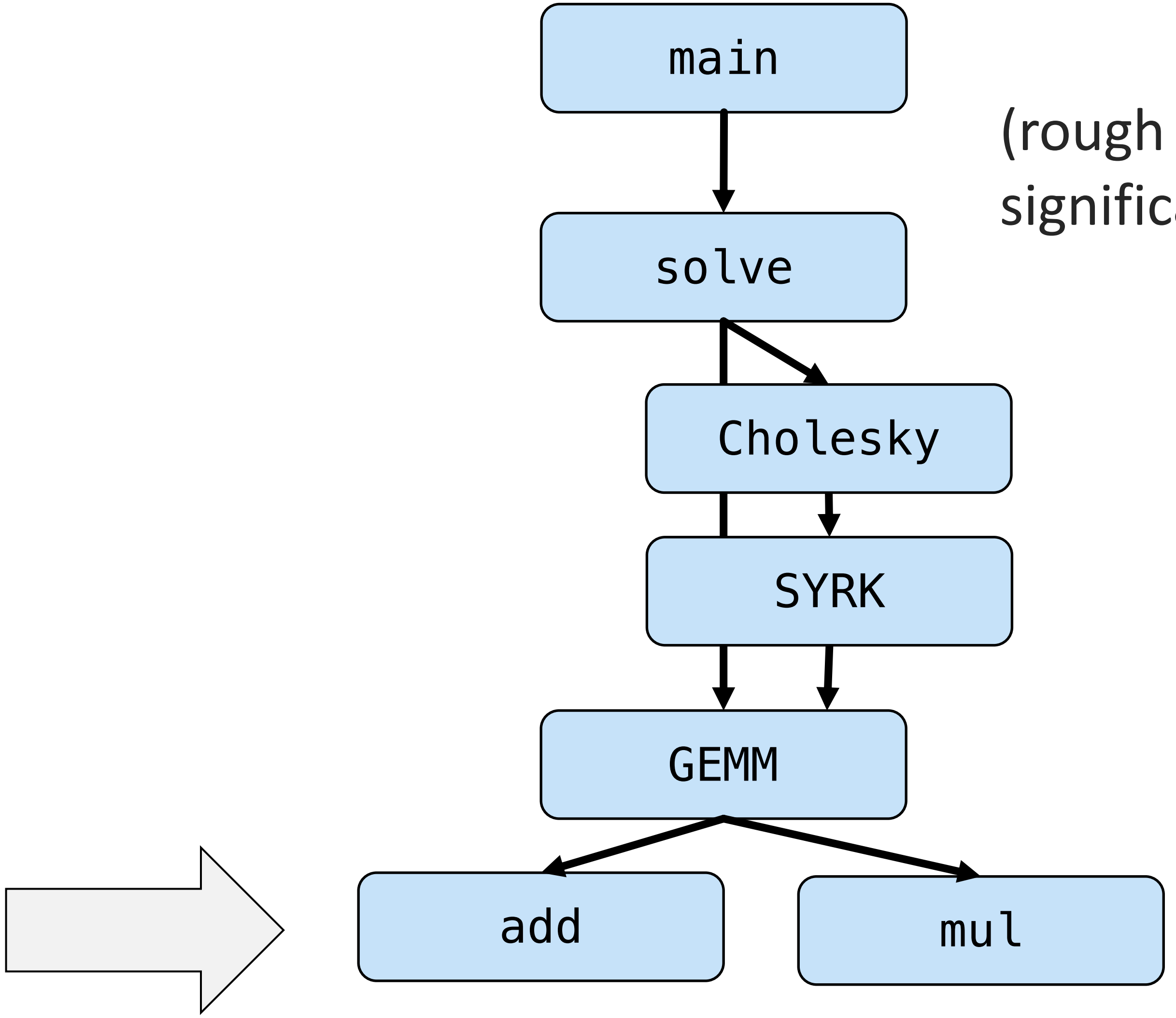
Unfortunately, we are now utterly memory bound…

1024-bit (960-bit)

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	-	-	-	227 MOp/s	1×	36×
FPGA 1 CU	361 MHz	27%	8%	361 MOp/s	1.6×	57.3×
FPGA 4 CUs	293 MHz	58%	42%	1202 MOp/s	5.3×	190.9×

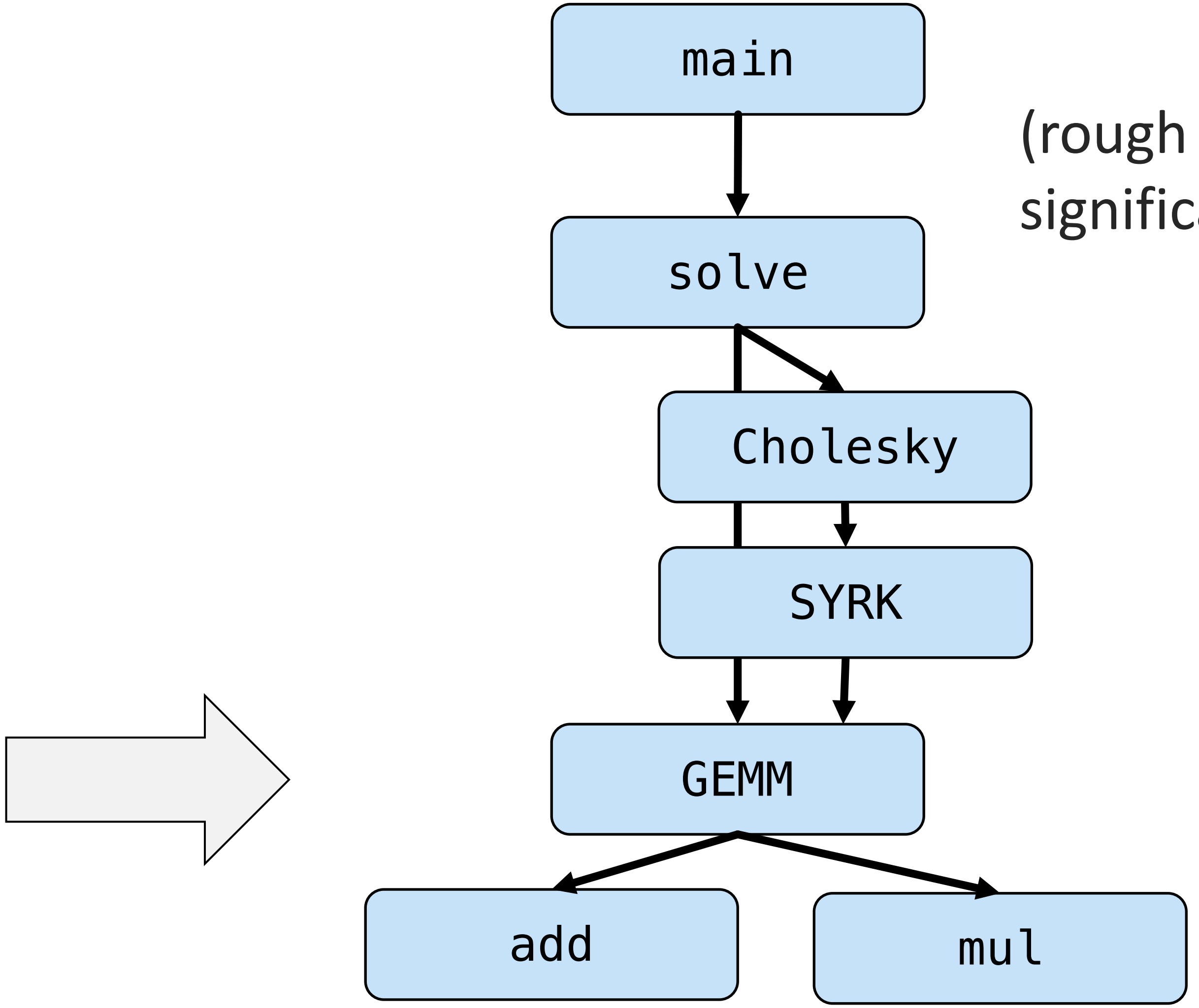
Xilinx Alveo U250 vs. CPU node with 2× Intel Xeon E5-2695 v4 18-core CPUs in a dual-socket configuration (36 cores per node)

Pop the callstack



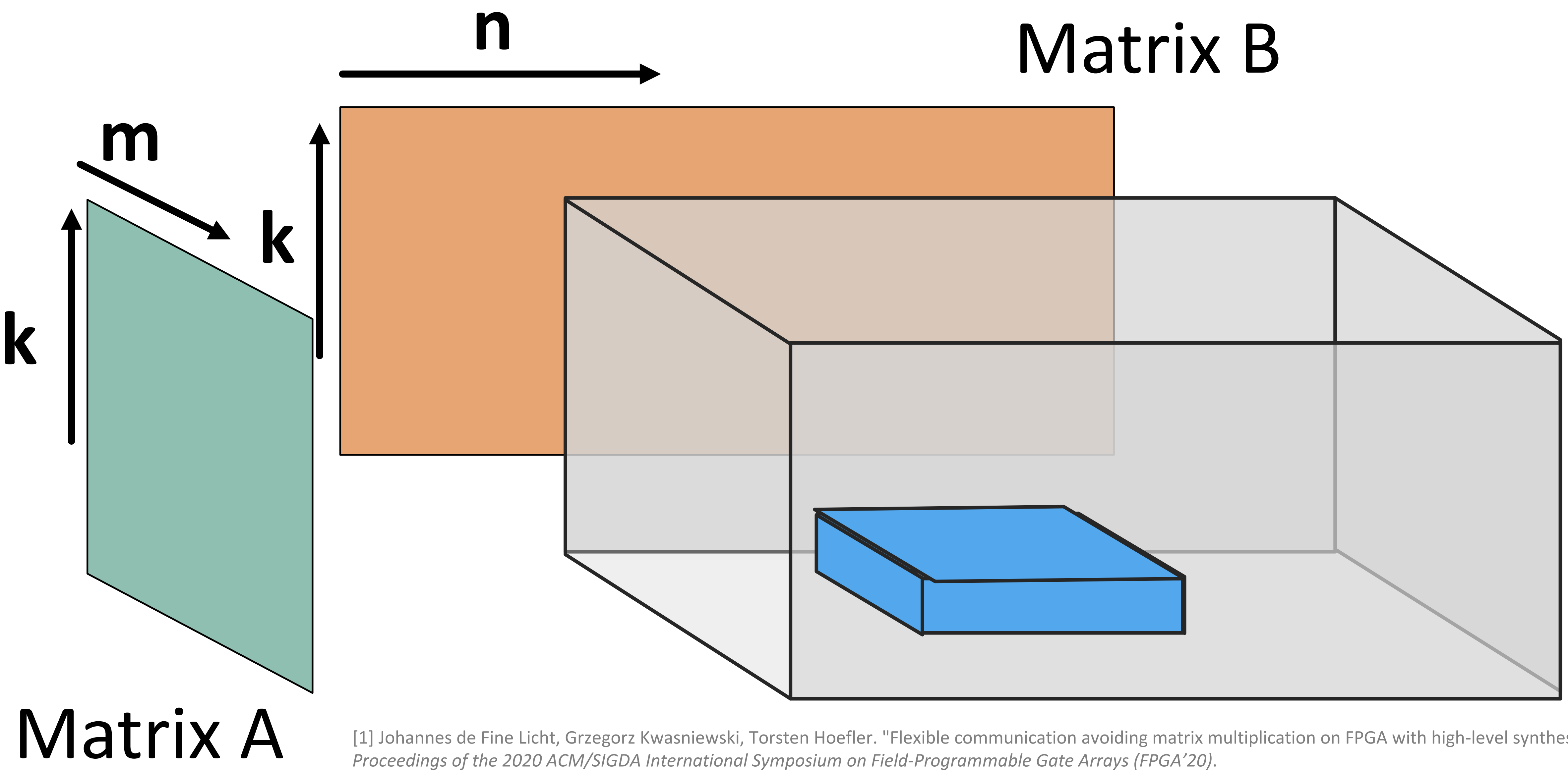
(rough approximation of most significant SDPB callgraph)

Pop the callstack



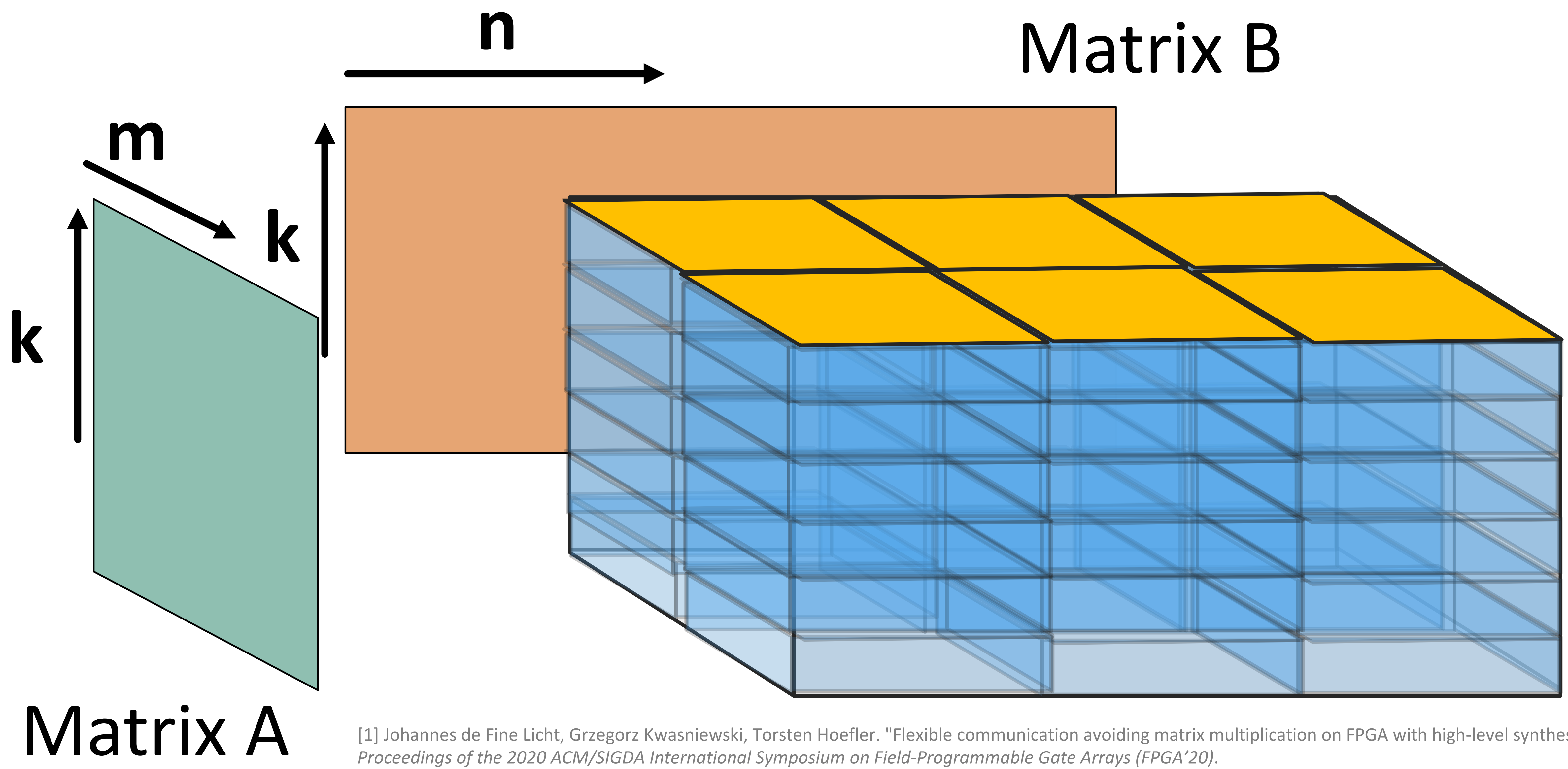
(rough approximation of most significant SDPB callgraph)

We know matrix multiplication!



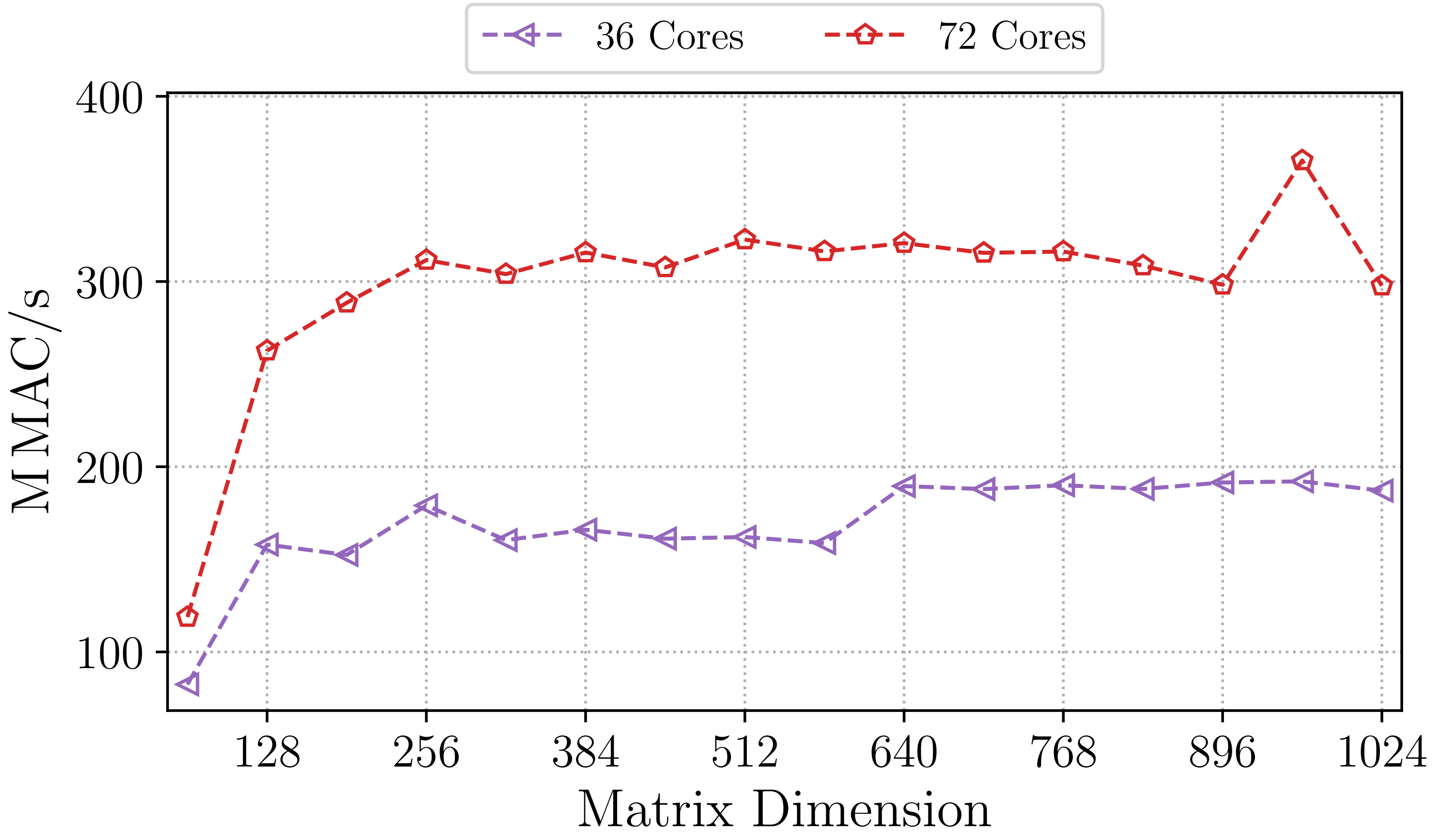
[1] Johannes de Fine Licht, Grzegorz Kwasniewski, Torsten Hoefler. "Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis." *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20).*

We know matrix multiplication!



[1] Johannes de Fine Licht, Grzegorz Kwasniewski, Torsten Hoefler. "Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis." *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20).*

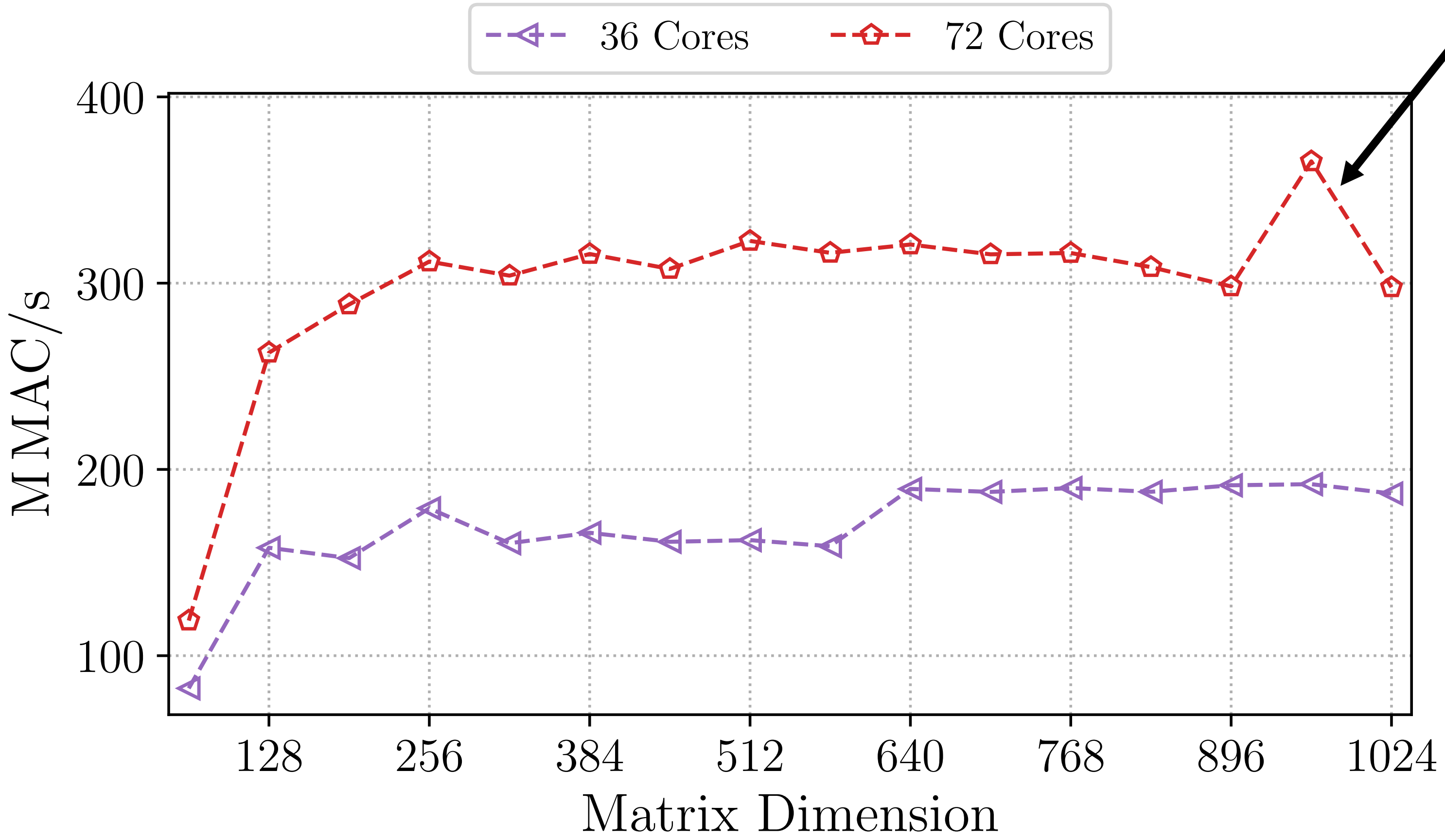
Matrix multiplication performance (512/448-bit)



Xilinx Alveo U250 vs. CPU nodes with 2× Intel Xeon E5-2695 v4
18-core CPUs in a dual-socket configuration (36 cores per node)

Matrix multiplication performance (512/448-bit)

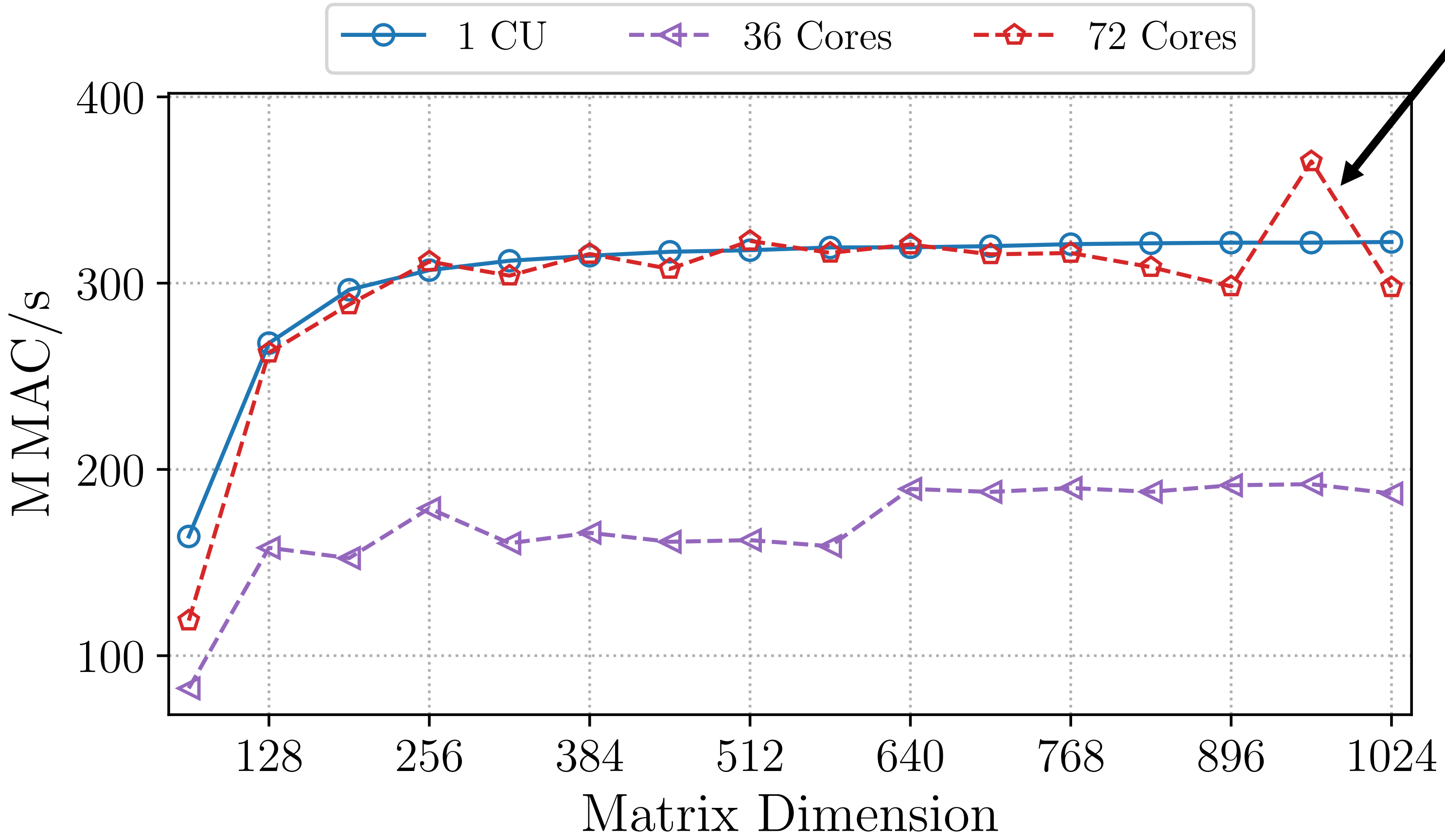
Running **Elemental** with **MPI**.



Xilinx Alveo U250 vs. CPU nodes with 2× Intel Xeon E5-2695 v4
18-core CPUs in a dual-socket configuration (36 cores per node)

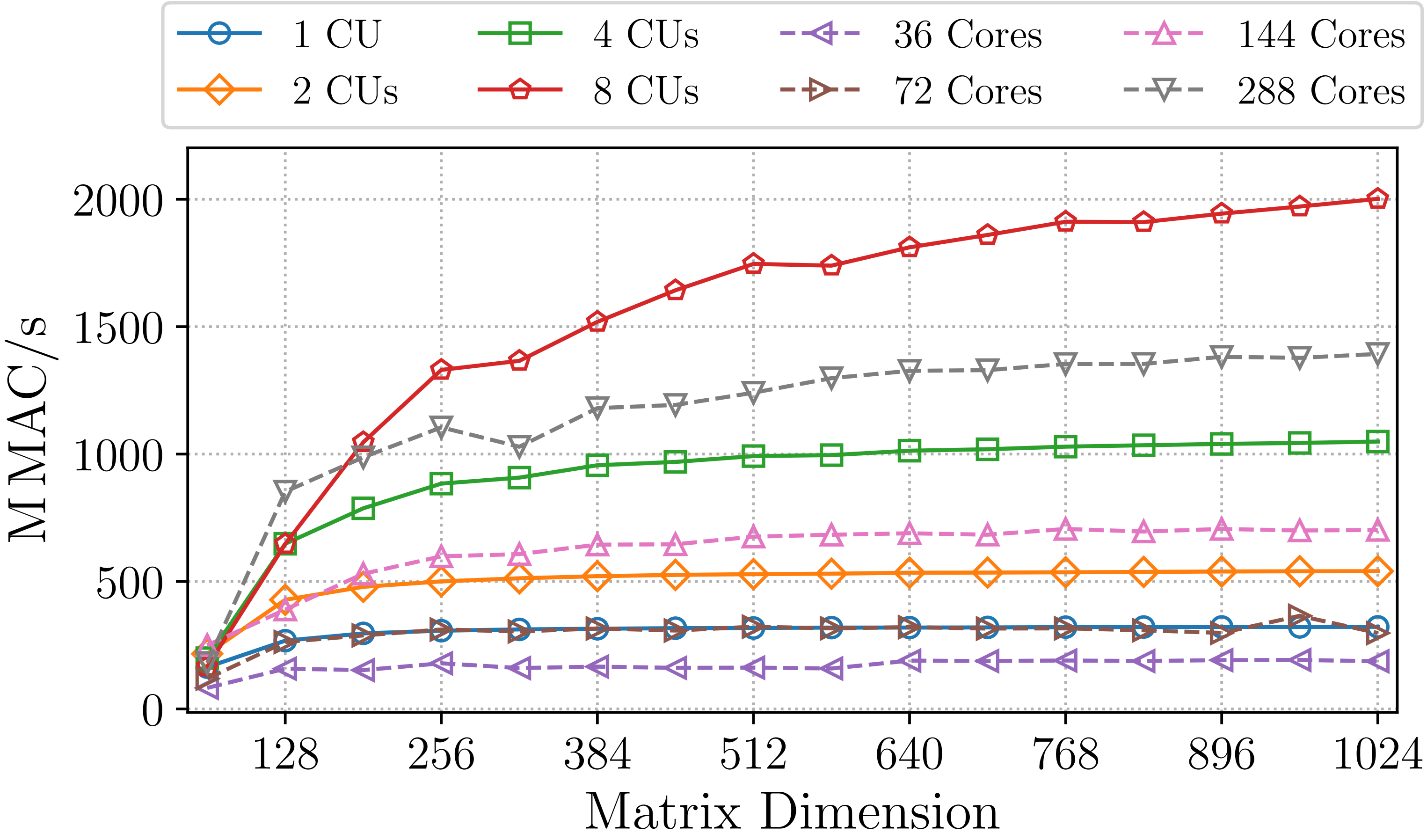
Matrix multiplication performance (512/448-bit)

Running **Elemental** with **MPI**.



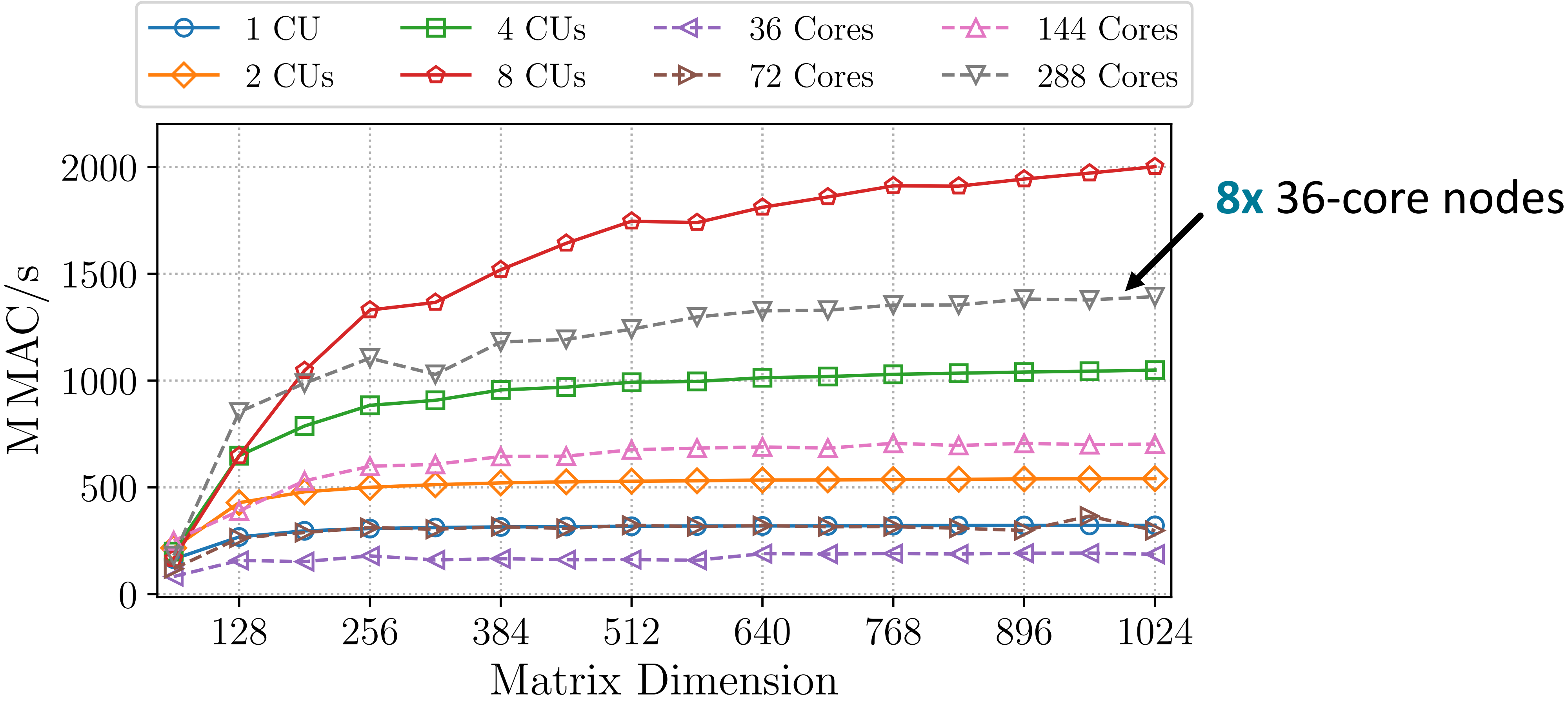
Xilinx Alveo U250 vs. CPU nodes with 2× Intel Xeon E5-2695 v4
18-core CPUs in a dual-socket configuration (36 cores per node)

Matrix multiplication performance (512/448-bit)



Xilinx Alveo U250 vs. CPU nodes with 2× Intel Xeon E5-2695 v4
18-core CPUs in a dual-socket configuration (36 cores per node)

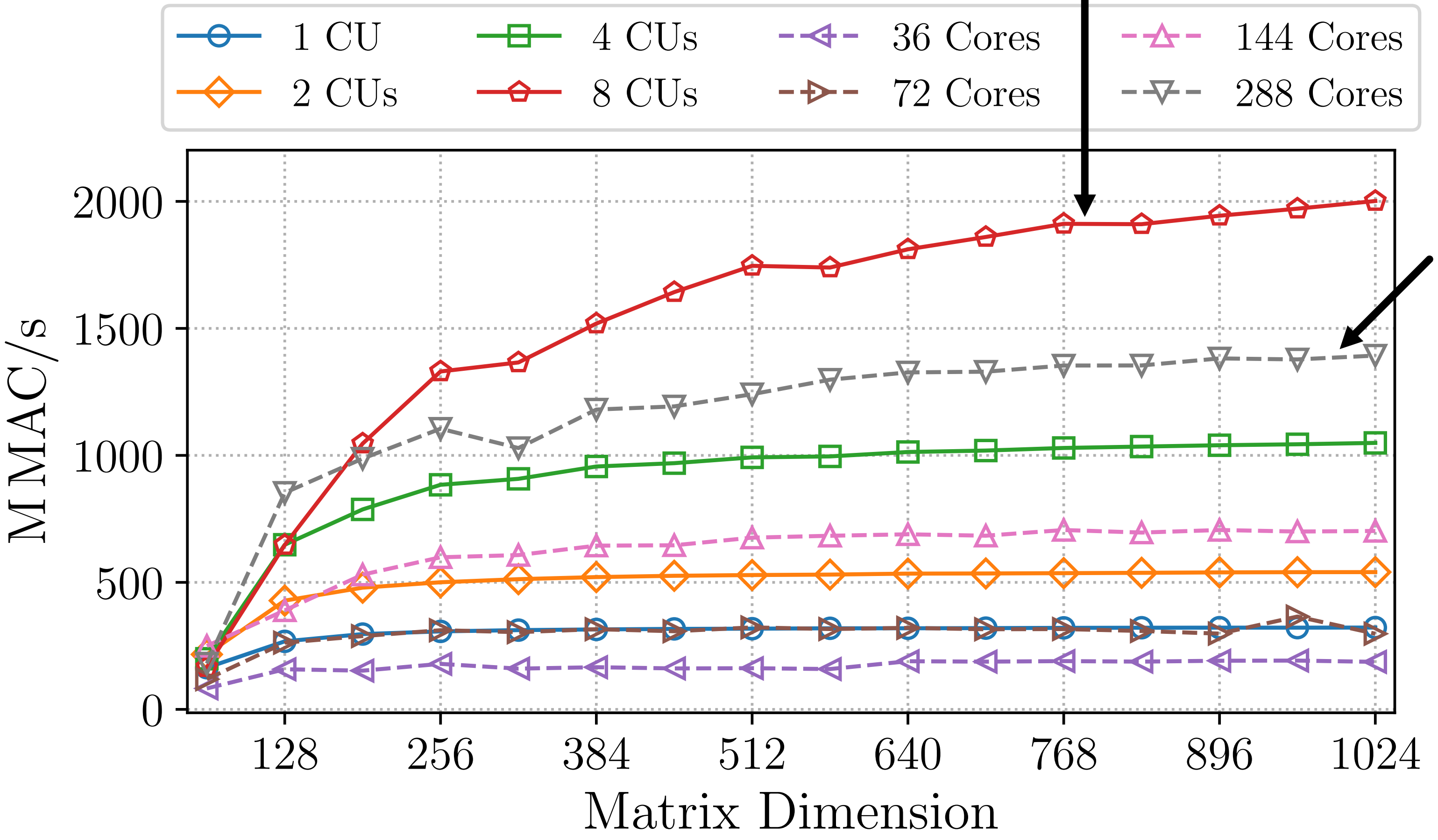
Matrix multiplication performance (512/448-bit)



Xilinx Alveo U250 vs. CPU nodes with 2× Intel Xeon E5-2695 v4
18-core CPUs in a dual-socket configuration (36 cores per node)

Matrix multiplication performance (512/448-bit)

Still a **single FPGA** running off four banks.

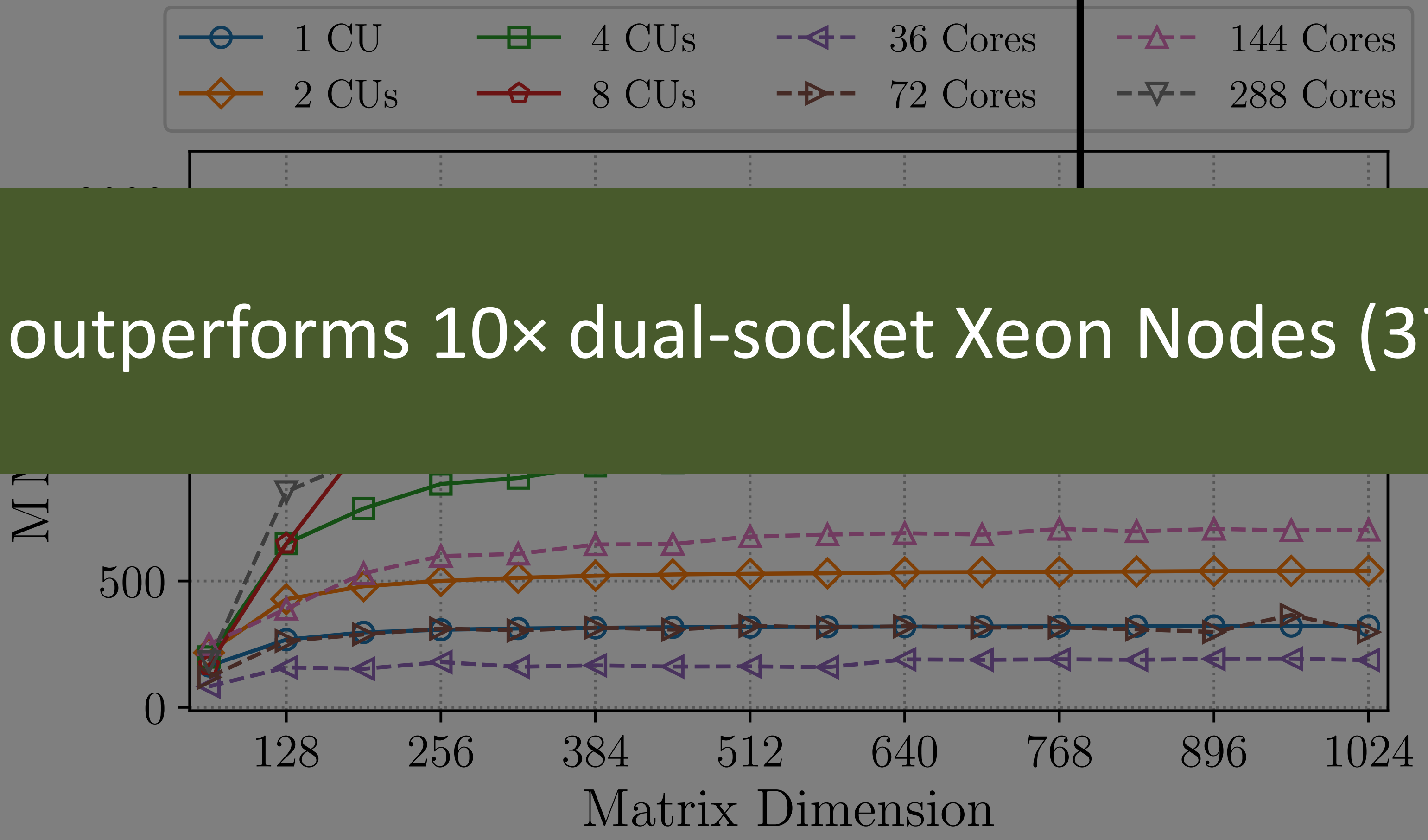


8x 36-core nodes

Xilinx Alveo U250 vs. CPU nodes with 2× Intel Xeon E5-2695 v4
18-core CPUs in a dual-socket configuration (36 cores per node)

Matrix multiplication performance (512/448-bit)

Still a **single FPGA** running off four banks.



One FPGA outperforms 10× dual-socket Xeon Nodes (375× cores)

Xilinx Alveo U250 vs. CPU nodes with 2× Intel Xeon E5-2695 v4
18-core CPUs in a dual-socket configuration (36 cores per node)

Plug-and-play

Plug-and-play

Step 1: Configure, build, and install	cmake .. -DAPFP_PLATFORM=xilinx_u250_gen3x16_xdma_3_1_202020_1 -DAPFP_COMPUTE_UNITS=8 make hw make install

Plug-and-play

Step 1:
Configure, build, and install

```
cmake .. -DAPFP_PLATFORM=xilinx_u250_gen3x16_xdma_3_1_202020_1 -DAPFP_COMPUTE_UNITS=8
make hw
make install
```

Step 2:
Link from CMake

```
find_package(MPFR REQUIRED)
find_package(APFP REQUIRED)

include_directories(SYSTEM ${APFP_INCLUDES} ${MPFR_INCLUDES})
add_executable(foo src/foo.cpp)
target_link_libraries(foo ${APFP_LIBRARIES} ${MPFR_LIBRARIES})
```

Plug-and-play

Step 1:
Configure, build, and install

```
cmake .. -DAPFP_PLATFORM=xilinx_u250_gen3x16_xdma_3_1_202020_1 -DAPFP_COMPUTE_UNITS=8
make hw
make install
```

Step 2:
Link from CMake

```
find_package(MPFR REQUIRED)
find_package(APFP REQUIRED)

include_directories(SYSTEM ${APFP_INCLUDES} ${MPFR_INCLUDES})
add_executable(foo src/foo.cpp)
target_link_libraries(foo ${APFP_LIBRARIES} ${MPFR_LIBRARIES})
```

Step 3:
Call BLAS API

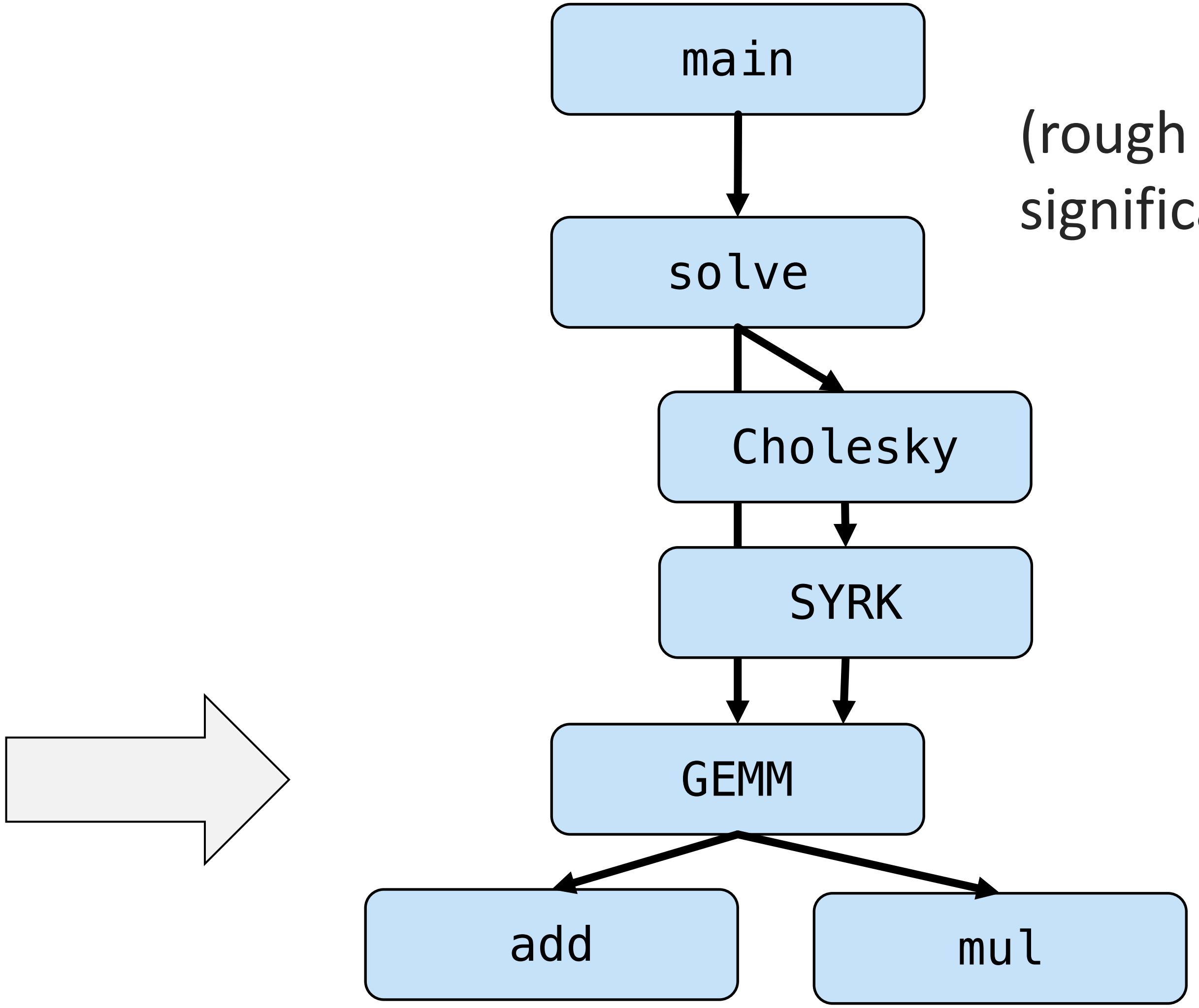
```
apfp::Gemm(apfp::BlasTrans::normal, apfp::BlasTrans::normal,
           m, n, k, IndexA, local_a.Matrix().LDim(),
           IndexB, local_b.Matrix().LDim(),
           IndexC, local_c.Matrix().LDim());
```

We still have work to do at higher bit widths: HLS struggles with the giant, **monolithic pipeline**, and we get issues with **contention**.

We still have work to do at higher bit widths: HLS struggles with the giant, **monolithic pipeline**, and we get issues with **contention**.

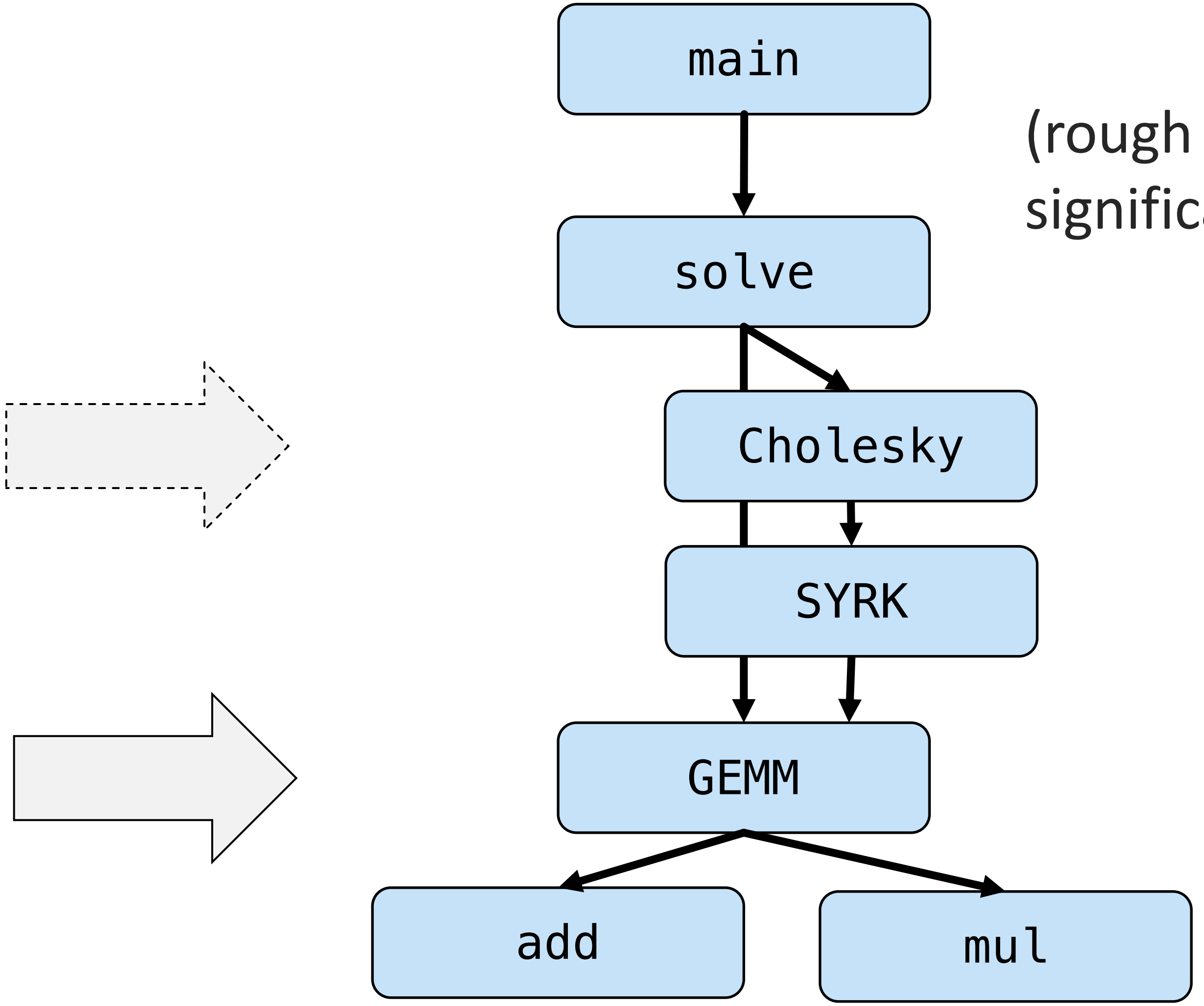
…there's potentially **2**× on the table for existing results!

We need one more pop



(rough approximation of most significant SDPB callgraph)

We need one more pop



(rough approximation of most significant SDPB callgraph)

Thank you!

Reach me at: definelicht@inf.ethz.ch

Try our code: github.com/spcl/apfp

When to bottom out