Fast basecases for arbitrary-size multiplication

Albin Ahlbäck[1]   Fredrik Johansson[2]

[1]LIX, CNRS, École polytechnique

[2]Inria Bordeaux

5 May 2025

Albin Ahlbäck has been supported by an ERC-2023-ADG grant for the ODELIX project.

Fredrik Johansson has been supported by the French ANR-20-CE-48-0014 NuSCAP.

Fast multiple-precision arithmetic – why?

Multiple-precision arithmetic is arithmetic with precision exceeding the natural register width.

We care about fast multiple-precision arithmetic. *Why?*

Examples include:

- Correctly rounded floating point arithmetic without the use of lookup tables (e.g. MPFR [1])
- Verifying the Riemann hypothesis up to very big numbers [2]
- Certified homotopy continuation (ex. [3, 4])

A common multiple-precision library to use is GMP, which is also very fast.

Fast multiple-precision arithmetic – why?

Multiple-precision arithmetic is arithmetic with precision exceeding the natural register width.

We care about fast multiple-precision arithmetic. *Why?*

Examples include:

- Correctly rounded floating point arithmetic without the use of lookup tables (e.g. MPFR [1])
- Verifying the Riemann hypothesis up to very big numbers [2]
- Certified homotopy continuation (ex. [3, 4])

A common multiple-precision library to use is GMP, which is also very fast.

Fast multiple-precision arithmetic – why?

Multiple-precision arithmetic is arithmetic with precision exceeding the natural register width.

We care about fast multiple-precision arithmetic. *Why?*

Examples include:

- Correctly rounded floating point arithmetic without the use of lookup tables (e.g. MPFR [1])
- Verifying the Riemann hypothesis up to very big numbers [2]
- Certified homotopy continuation (ex. [3, 4])

A common multiple-precision library to use is GMP, which is also very fast.

Fast multiple-precision arithmetic – why?

Multiple-precision arithmetic is arithmetic with precision exceeding the natural register width.

We care about fast multiple-precision arithmetic. *Why?*

Examples include:

- Correctly rounded floating point arithmetic without the use of lookup tables (e.g. MPFR [1])
- Verifying the Riemann hypothesis up to very big numbers [2]
- Certified homotopy continuation (ex. [3, 4])

A common multiple-precision library to use is GMP, which is also very fast.

Fast multiple-precision arithmetic – why?

Multiple-precision arithmetic is arithmetic with precision exceeding the natural register width.

We care about fast multiple-precision arithmetic. *Why?*

Examples include:

- Correctly rounded floating point arithmetic without the use of lookup tables (e.g. MPFR [1])
- Verifying the Riemann hypothesis up to very big numbers [2]
- Certified homotopy continuation (ex. [3, 4])

A common multiple-precision library to use is GMP, which is also very fast.

Basics of Multiple-Precision Arithmetic

Fundamentals are these schoolbook $\mathcal{O}(n)$ operations:

- Left and right shift: $r \leftarrow \lfloor a \cdot 2^e \rfloor$
- Addition and subtraction: $r \leftarrow a \pm b$
- $m \times 1$-multiplication: $r \leftarrow a \cdot b_0$ (`mul_1`)
- Addition of $m \times 1$-multiplication: $r \leftarrow r + a \cdot b_0$ (`addmul_1`)

Schoolbook $m \times n$-multiplication is then

$r \leftarrow a \cdot b_0$ // `mul_1`
for $i \leftarrow 1$ *to* $n-1$ **do**
  $r \leftarrow r + (a \cdot b_i) \cdot \beta^i$ // `addmul_1`
end

With these we can generate division, greatest common divisor, Toom-Cook multiplication, fast polynomial multiplication, ...

Basics of Multiple-Precision Arithmetic

Fundamentals are these schoolbook $\mathcal{O}(n)$ operations:

- Left and right shift: $r \leftarrow \lfloor a \cdot 2^e \rfloor$
- Addition and subtraction: $r \leftarrow a \pm b$
- $m \times 1$-multiplication: $r \leftarrow a \cdot b_0$ (`mul_1`)
- Addition of $m \times 1$-multiplication: $r \leftarrow r + a \cdot b_0$ (`addmul_1`)

Schoolbook $m \times n$-multiplication is then

$r \leftarrow a \cdot b_0$ // `mul_1`
for $i \leftarrow 1$ *to* $n-1$ **do**
  $r \leftarrow r + (a \cdot b_i) \cdot \beta^i$ // `addmul_1`
end

With these we can generate division, greatest common divisor, Toom-Cook multiplication, fast polynomial multiplication, ...

Basics of Multiple-Precision Arithmetic

Fundamentals are these schoolbook $\mathcal{O}(n)$ operations:

- Left and right shift: $r \leftarrow \lfloor a \cdot 2^e \rfloor$
- Addition and subtraction: $r \leftarrow a \pm b$
- $m \times 1$-multiplication: $r \leftarrow a \cdot b_0$ (`mul_1`)
- Addition of $m \times 1$-multiplication: $r \leftarrow r + a \cdot b_0$ (`addmul_1`)

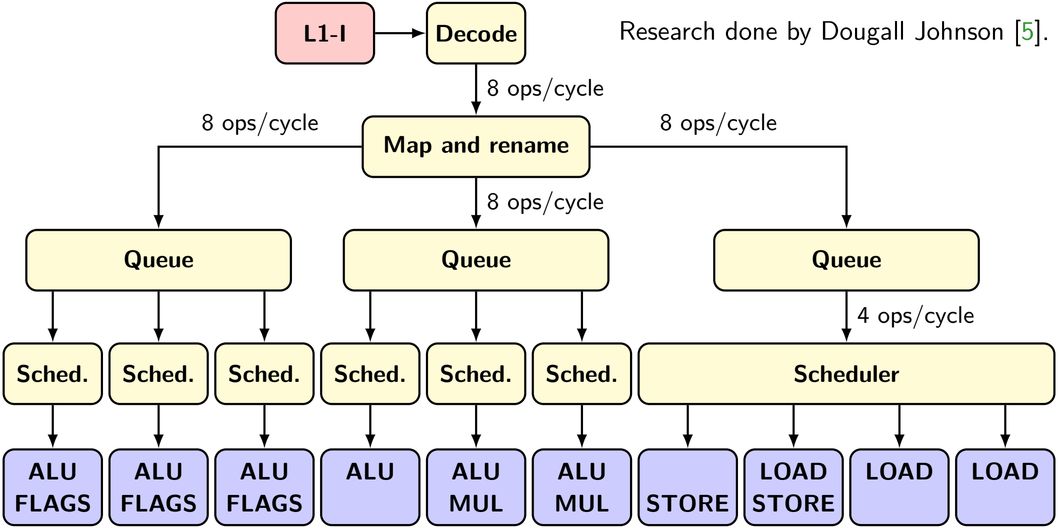
Schoolbook $m \times n$-multiplication is then

$r \leftarrow a \cdot b_0$ // `mul_1`
for $i \leftarrow 1$ *to* $n-1$ **do**
  $r \leftarrow r + (a \cdot b_i) \cdot \beta^i$ // `addmul_1`
end

With these we can generate division, greatest common divisor, Toom-Cook multiplication, fast polynomial multiplication, ...

Apple M1 Pipeline (Simplified)

Research done by Dougall Johnson [5].



CPU Pipeline

Simple version:

1. Read some instructions from memory
2. Schedule the instructions to the correct unit
3. Units executes instructions

This scheme allows for:

- Concurrent execution of multiple instructions
- Out-of-order execution

But one has to be aware of dependency chains.

Example: (Dependency chain)
Consider the algorithm

$$x_1 \leftarrow x_0 + a_0\,,$$
$$x_2 \leftarrow x_1 + a_1\,,$$
$$x_3 \leftarrow x_2 + a_2\,.$$

The result x_3 depends on x_2 which depends on x_1. This is called a *dependency chain*.

CPU Pipeline

Simple version:

1. Read some instructions from memory
2. Schedule the instructions to the correct unit
3. Units executes instructions

This scheme allows for:

- Concurrent execution of multiple instructions
- Out-of-order execution

But one has to be aware of dependency chains.

Example: (Dependency chain)
Consider the algorithm

$$x_1 \leftarrow x_0 + a_0,$$
$$x_2 \leftarrow x_1 + a_1,$$
$$x_3 \leftarrow x_2 + a_2.$$

The result x_3 depends on x_2 which depends on x_1. This is called a *dependency chain*.

CPU Pipeline

Simple version:

1. Read some instructions from memory
2. Schedule the instructions to the correct unit
3. Units executes instructions

This scheme allows for:

- Concurrent execution of multiple instructions
- Out-of-order execution

But one has to be aware of dependency chains.

Example: (Dependency chain)
Consider the algorithm

$$x_1 \leftarrow x_0 + a_0,$$
$$x_2 \leftarrow x_1 + a_1,$$
$$x_3 \leftarrow x_2 + a_2.$$

The result x_3 depends on x_2 which depends on x_1. This is called a *dependency chain*.

Lower bound of GMP's `addmul_1`

```
L(top): ldp     u0, u1, [up], #16        adds    x0, r0, x0
        ldp     u2, u3, [up], #16        adcs    u0, r1, u0
        ldp     r0, r1, [rp]             adcs    u1, r2, u1
        ldp     r2, r3, [rp,#16]         adcs    u2, r3, u2
        mul     x0, u0, v0               adc     u3, u3, zero
        umulh   u0, u0, v0               adds    x0, x0, CY
        mul     x1, u1, v0               adcs    u0, u0, x1
        umulh   u1, u1, v0               adcs    u1, u1, x2
        mul     x2, u2, v0               adcs    u2, u2, x3
        umulh   u2, u2, v0               adc     CY, u3, zero
        mul     x3, u3, v0               stp     x0, u0, [rp], #16
        umulh   u3, u3, v0               stp     u1, u2, [rp], #16
                                         sub     n, n, #1
                                         cbnz    n, L(top)
```

Unit type (amount)	**Cycles / 4 words**
LOAD/STORE (3/2)	
MUL (2)	
ALU+FLAGS (3)	

Lower bound of GMP's `addmul_1`

```
L(top): ldp     u0, u1, [up], #16
        ldp     u2, u3, [up], #16
        ldp     r0, r1, [rp]
        ldp     r2, r3, [rp,#16]
        mul     x0, u0, v0
        umulh   u0, u0, v0
        mul     x1, u1, v0
        umulh   u1, u1, v0
        mul     x2, u2, v0
        umulh   u2, u2, v0
        mul     x3, u3, v0
        umulh   u3, u3, v0
        adds    x0, r0, x0
        adcs    u0, r1, u0
        adcs    u1, r2, u1
        adcs    u2, r3, u2
        adc     u3, u3, zero
        adds    x0, x0, CY
        adcs    u0, u0, x1
        adcs    u1, u1, x2
        adcs    u2, u2, x3
        adc     CY, u3, zero
        stp     x0, u0, [rp], #16
        stp     u1, u2, [rp], #16
        sub     n, n, #1
        cbnz    n, L(top)
```

Unit type (amount)	**Cycles / 4 words**
LOAD/STORE (3/2)	2
MUL (2)	
ALU+FLAGS (3)	

Lower bound of GMP's `addmul_1`

```
L(top): ldp    u0, u1, [up], #16
        ldp    u2, u3, [up], #16
        ldp    r0, r1, [rp]
        ldp    r2, r3, [rp,#16]
        mul    x0, u0, v0
        umulh  u0, u0, v0
        mul    x1, u1, v0
        umulh  u1, u1, v0
        mul    x2, u2, v0
        umulh  u2, u2, v0
        mul    x3, u3, v0
        umulh  u3, u3, v0
        adds   x0, r0, x0
        adcs   u0, r1, u0
        adcs   u1, r2, u1
        adcs   u2, r3, u2
        adc    u3, u3, zero
        adds   x0, x0, CY
        adcs   u0, u0, x1
        adcs   u1, u1, x2
        adcs   u2, u2, x3
        adc    CY, u3, zero
        stp    x0, u0, [rp], #16
        stp    u1, u2, [rp], #16
        sub    n, n, #1
        cbnz   n, L(top)
```

Unit type (amount)	**Cycles / 4 words**
LOAD/STORE (3/2)	2
MUL (2)	4
ALU+FLAGS (3)	

Lower bound of GMP's `addmul_1`

```
L(top): ldp     u0, u1, [up], #16
        ldp     u2, u3, [up], #16
        ldp     r0, r1, [rp]
        ldp     r2, r3, [rp,#16]
        mul     x0, u0, v0
        umulh   u0, u0, v0
        mul     x1, u1, v0
        umulh   u1, u1, v0
        mul     x2, u2, v0
        umulh   u2, u2, v0
        mul     x3, u3, v0
        umulh   u3, u3, v0
```

```
adds    x0, r0, x0
adcs    u0, r1, u0
adcs    u1, r2, u1
adcs    u2, r3, u2
adc     u3, u3, zero
adds    x0, x0, CY
adcs    u0, u0, x1
adcs    u1, u1, x2
adcs    u2, u2, x3
adc     CY, u3, zero
stp     x0, u0, [rp], #16
stp     u1, u2, [rp], #16
sub     n, n, #1
cbnz    n, L(top)
```

Unit type (amount)	**Cycles / 4 words**
LOAD/STORE (3/2)	2
MUL (2)	4
ALU+FLAGS (3)	4

Lower bound of GMP's `addmul_1`

```
L(top): ldp     u0, u1, [up], #16
        ldp     u2, u3, [up], #16
        ldp     r0, r1, [rp]
        ldp     r2, r3, [rp,#16]
        mul     x0, u0, v0
        umulh   u0, u0, v0
        mul     x1, u1, v0
        umulh   u1, u1, v0
        mul     x2, u2, v0
        umulh   u2, u2, v0
        mul     x3, u3, v0
        umulh   u3, u3, v0
```

Unit type (amount)	Cycles / 4 words
LOAD/STORE (3/2)	2
MUL (2)	4
ALU+FLAGS (3)	~~4~~ 5

```
        adds    x0, r0, x0
        adcs    u0, r1, u0
        adcs    u1, r2, u1
        adcs    u2, r3, u2
        adc     u3, u3, zero
        adds    x0, x0, CY
        adcs    u0, u0, x1
        adcs    u1, u1, x2
        adcs    u2, u2, x3
        adc     CY, u3, zero
        stp     x0, u0, [rp], #16
        stp     u1, u2, [rp], #16
        sub     n, n, #1
        cbnz    n, L(top)
```

5 cycles per 4 words?

Benchmarks says yes!

What can be improved?

GMP's `addmul_1` will do $\frac{k+1}{k}$ cycles per word *asymptotically* on Apple M1, where k is the number of unrolls.

To improve this, we fully unroll one size parameter in the full multiplication:

- Reduces overhead,
- Avoids breaking carry chains, hopefully lets

$$\frac{k+1}{k} \text{ cycles per word} \longrightarrow 1 \text{ cycle per word}$$

  asymptotically.

On x86 CPUs (Intel and AMD), we completely unroll both size parameters.

What can be improved?

GMP's `addmul_1` will do $\frac{k+1}{k}$ cycles per word *asymptotically* on Apple M1, where k is the number of unrolls.

To improve this, we fully unroll one size parameter in the full multiplication:

- Reduces overhead,
- Avoids breaking carry chains, hopefully lets

  $$\frac{k+1}{k} \text{ cycles per word} \longrightarrow 1 \text{ cycle per word}$$

  asymptotically.

On x86 CPUs (Intel and AMD), we completely unroll both size parameters.

What can be improved?

GMP's `addmul_1` will do $\dfrac{k+1}{k}$ cycles per word *asymptotically* on Apple M1, where k is the number of unrolls.

To improve this, we fully unroll one size parameter in the full multiplication:

- Reduces overhead,
- Avoids breaking carry chains, hopefully lets

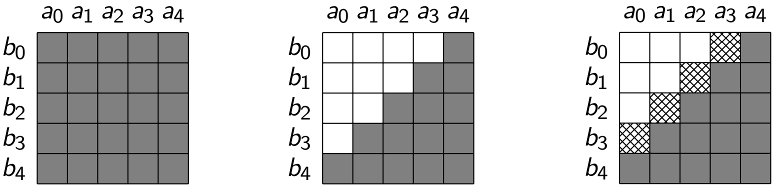
  $$\frac{k+1}{k} \text{ cycles per word} \longrightarrow 1 \text{ cycle per word}$$

  asymptotically.

On x86 CPUs (Intel and AMD), we completely unroll both size parameters.

High multiplication

High multiplication is multiplication where we scrap the lower part of the result.
Important use cases include floating point arithmetic and modular arithmetic.



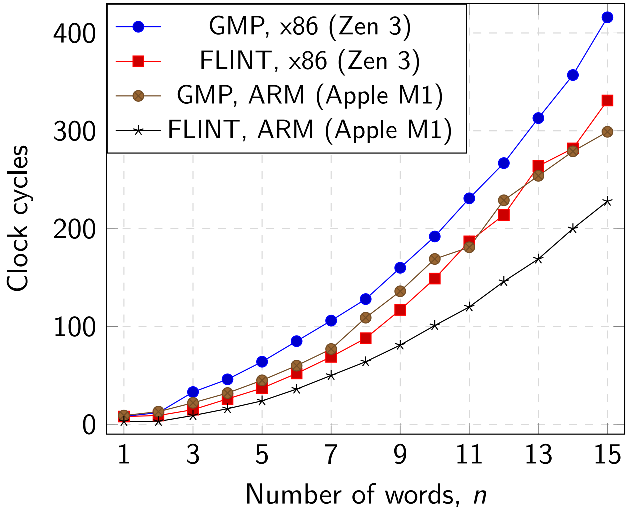
Full multiplication, $2n$ words | Sloppy approximate, $\sim n$ words | Precise approximate, $\sim n+1$ words

☐ – scrapped

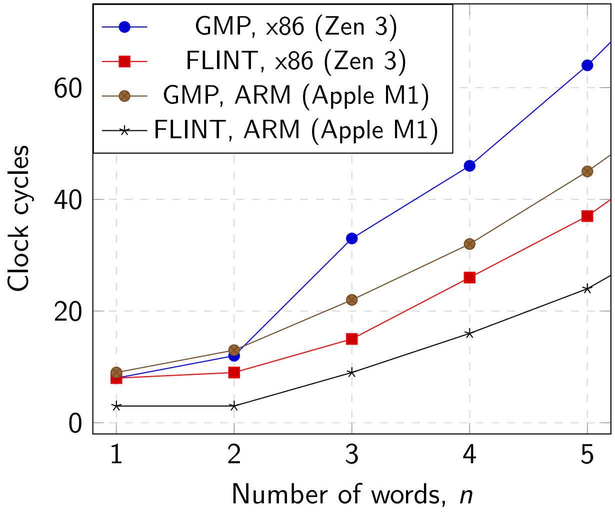
▩ – high multiplication between two words u and v: $\lfloor uv/\beta \rfloor$

■ – full multiplication

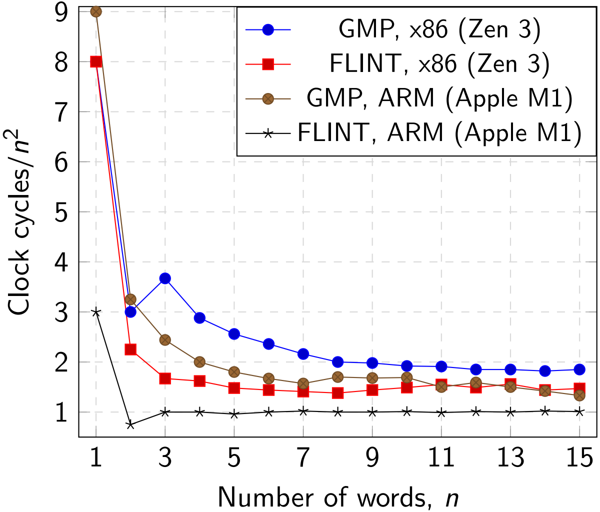
Results, full multiplication (throughput)



Results, full multiplication (throughput)



Results, full multiplication (throughput)

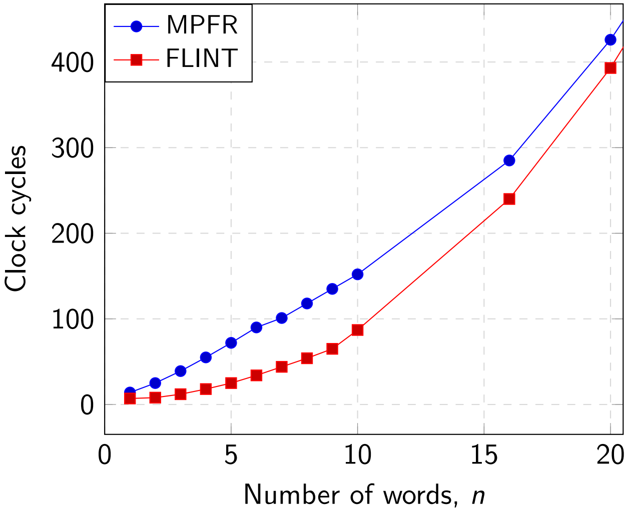


10^7 multiplications with lengths $m, n \in \{1, 2, \ldots, N\}$, uniformly random

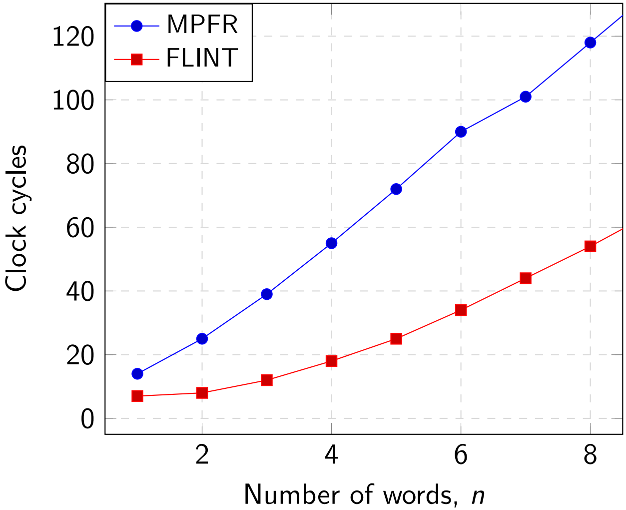
	GMP (`mpn_mul`)				Ours (`flint_mpn_mul`)			
N	Time	C	J	I	Time	C	J	I
Random, x86-64 (Zen 3)								
8	0.32 s	18.3%	22.3%	0%	0.18 s	20.9%	48.4%	0.00%
16	0.55 s	10.0%	18.2%	0%	0.43 s	14.3%	33.1%	3.05%
32	1.39 s	10.5%	12.7%	0%	1.32 s	10.7%	16.7%	0.41%
64	4.48 s	11.5%	11.6%	0%	4.29 s	12.9%	14.3%	0.12%
Random, ARM64 (M1)								
8	0.30 s	11.4%	0.00%	0.00%	0.23 s	11.2%	41.7%	0.01%
16	0.50 s	10.9%	0.00%	0.00%	0.43 s	10.5%	41.6%	0.00%
32	1.31 s	9.6%	0.00%	0.00%	1.13 s	10.0%	13.9%	0.00%
64	4.16 s	8.3%	0.20%	0.02%	3.82 s	9.8%	4.2%	0.06%

Table: Conditional branch misprediction rates “C”, indirect jump address misprediction rates “J” and instruction cache miss rates “I”.

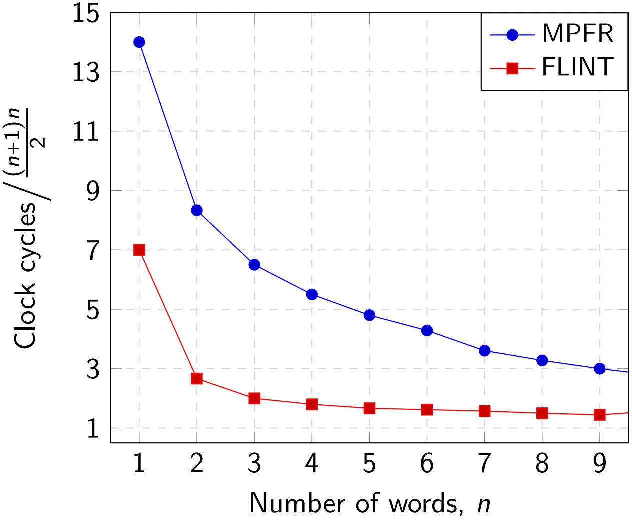
Results, high multiplication on Zen 3 (throughput)



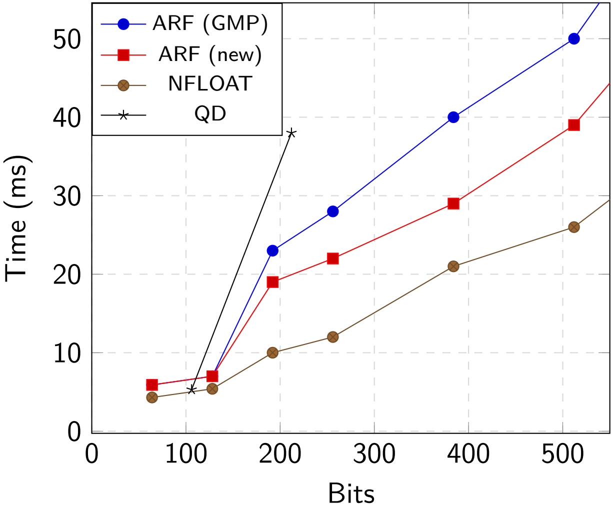
Results, high multiplication on Zen 3 (throughput)



Results, high multiplication on Zen 3 (throughput)



Multiply two 100×100 FP matrices using dot products (Zen 3)



Conclusions and thoughts

- Critical functions require hardware awareness – in our case, ISA
- Straight line programs can be important to reduce overhead when going from native data types to multiple precision arithmetic
- Poor compiler support for multiple precision arithmetic – handwritten assembly remain critical

Conclusions and thoughts

- Critical functions require hardware awareness – in our case, ISA
- Straight line programs can be important to reduce overhead when going from native data types to multiple precision arithmetic
- Poor compiler support for multiple precision arithmetic – handwritten assembly remain critical

Conclusions and thoughts

- Critical functions require hardware awareness – in our case, ISA
- Straight line programs can be important to reduce overhead when going from native data types to multiple precision arithmetic
- Poor compiler support for multiple precision arithmetic – handwritten assembly remain critical

Bibliography

[1] Laurent Fousse et al. "MPFR: A multiple-precision binary floating-point library with correct rounding". In: *ACM Trans. Math. Softw.* 33.2 (June 2007), 13–es. ISSN: 0098-3500. DOI: 10.1145/1236463.1236468.

[2] Dave Platt and Tim Trudgian. "The Riemann hypothesis is true up to $3 \cdot 10^{12}$". In: *Bulletin of the London Mathematical Society* 53.3 (Jan. 2021), pp. 792–797. ISSN: 1469-2120. DOI: 10.1112/blms.12460.

[3] Joris van der Hoeven. *Reliable homotopy continuation*. Research Report. LIX, Ecole polytechnique, Jan. 2015. URL: https://hal.science/hal-00589948.

[4] Alexandre Guillemot and Pierre Lairez. "Validated Numerics for Algebraic Path Tracking". In: *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation*. ISSAC '24. ACM, July 2024, pp. 36–45. DOI: 10.1145/3666000.3669673.

[5] Dougall Johnson. *Firestorm Overview*. 2023. URL: https://dougallj.github.io/applecpu/firestorm.html (visited on 02/28/2025).