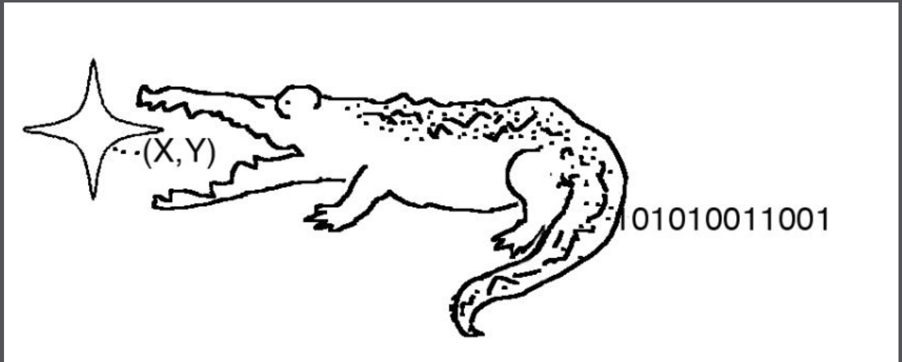
Optimizing Elligator 1 on Curve1174

Christopher Vogelsanger, Freya Murphy, Miro Haller

Introduction

Elligator

- Elligator: Elliptic-Curve points indistinguishable from uniform random strings. [1]
    - Helps prevent censorship of obvious curve points



[1] D. Bernstein et al. Elligator: Elliptic-curve points indistinguishable from uniform random strings. ACM Conference on Computer and Communications Security 2013. 2013.

Elligator mapping

$$\begin{aligned}
u &= (1-t)/(1+t),\\
v &= u^5 + (r^2-2)u^3 + u,\\
X &= \chi(v)u,\\
Y &= (\chi(v)v)^{(q+1)/4}\chi(v)\chi(u^2+1/c^2),\\
x &= (c-1)sX(1+X)/Y,\\
y &= (rX-(1+X)^2)/(rX+(1+X)^2)
\end{aligned}$$

Forward mapping (string to point)

$$\begin{aligned}
\eta &= \frac{y-1}{2(y+1)},\\
\bar{X} &= -(1+\eta r) + ((1+\eta r)^2 - 1)^{(q+1)/4},\\
z &= \chi((c-1)s\bar{X}(1+\bar{X})x(\bar{X}^2+1/c^2)),\\
\bar{u} &= z\bar{X},\\
\bar{t} &= (1-\bar{u})/(1+\bar{u})
\end{aligned}$$

Inverse mapping (point to string)

Straightforward C Implementation

Build, Test, and Benchmark Environment

- Unit testing framework: Check [1]
  - Organized in test suites and test cases
  - Nice test result report
  - GitLab CI/CD Pipeline Integration

[1] https://libcheck.github.io/check/

Build, Test, and Benchmark Environment

- Benchmarking Library
  - Takes prepare, benchmark, and cleanup function
  - Execute benchmark function in *S* sets each with *R* repetitions
    - Take median

- Benchmarks
  - Measure runtime of all functions
  - Count function calls
  - Count integer operations

$$
\begin{array}{l}
\text{for set in } \{1, 2, ..., \text{S}\} \\
\quad \texttt{prep()} \\
\quad t_0 = \texttt{tsc}() \\
\quad \text{for j in } \{1, 2, ..., \text{R}\} \\
\quad\quad \texttt{bench\_fn}(j) \\
\quad T = T \cap \{(\texttt{tsc}() - t_0)/R\} \\
\quad \texttt{cleanup}() \\
\text{return } \texttt{median}(T)
\end{array}
$$

Reference Implementations

- Could not find any available implementations
- Elligator website mentions a Sage implementation [1].
- We made our own Sage implementation
- BigInt arithmetic:
  - GMP (The GNU Multi Precision Library) [2]
  - Used to benchmark BigInt operations and Elligator mapping

[1] https://elligator.cr.yp.to/software.html
[2] https://gmplib.org

Straightforward BigInt Library

- BigInt allocates *alloc_size* 64-bit chunks of memory
- *size* chunks are currently used
- Big integer arithmetics
  - “The Art of Computer Programming” [1]
- Clean code & convenient interface
  - Allows aliasing names, nested calls
  - Explicit error messages

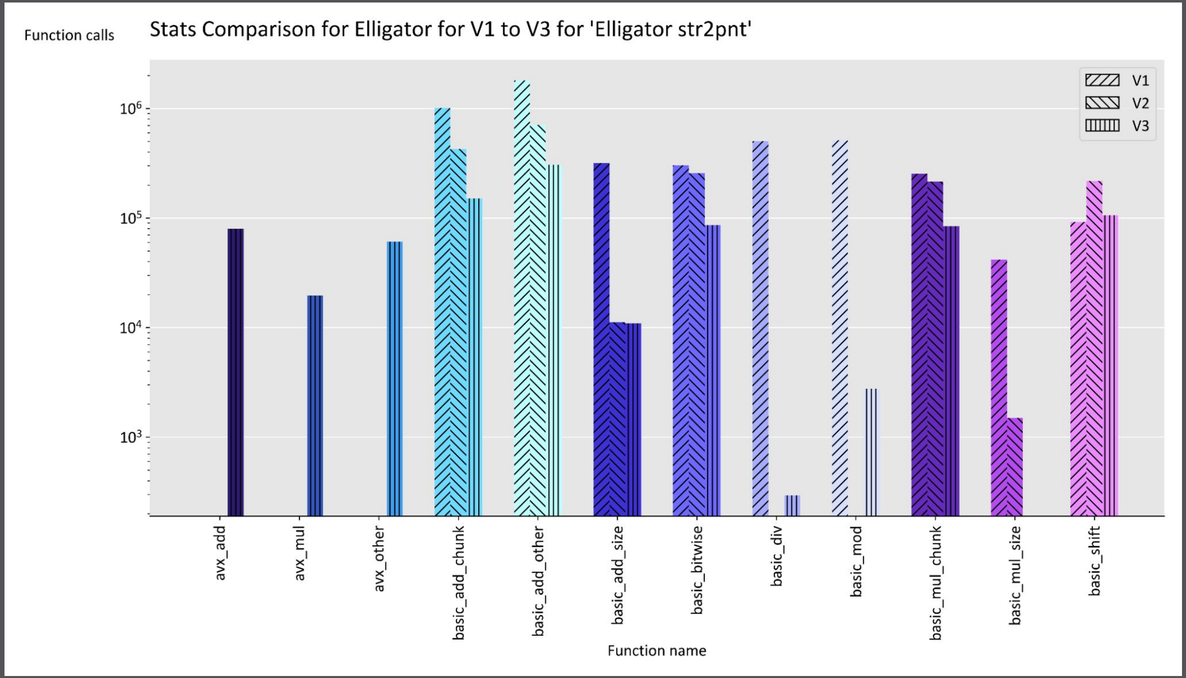
```c
typedef struct BigInts
{
    uint64_t sign : 1;
    uint64_t overflow : 1;
    uint64_t size : 62;
    uint64_t alloc_size : 62;
    dbl_chunk_size_t *chunks;
} BigInt;
```

[1] Knuth, Donald Ervin. The Art of Computer Programming. Volume 2, Seminumerical Algorithms. 3rd ed. Place of publication not identified: Addison Wesley, 1997. Print.

Cost Analysis

Integer Operations

- Keep track of following iops
  - Add/Sub
  - Mul
  - Div
  - Mod
  - Shift
  - Bitwise
- Cost function:
  - $C(x) = \Sigma \text{ iops}(x)$
- Add up all integer operations



Roofline Plot

- Main optimization target:
  - MacBook Pro Mid 2015
  - Intel Haswell i7-4980HQ 2.8 GHz
  - Apple clang version 12.0.0
- Ports with execution units for integers [1]

Port 0	**Port 1**	**Port 5**	**Port 6**
ALU Shift	ALU	ALU	ALU, Shift
Divide	Slow int		

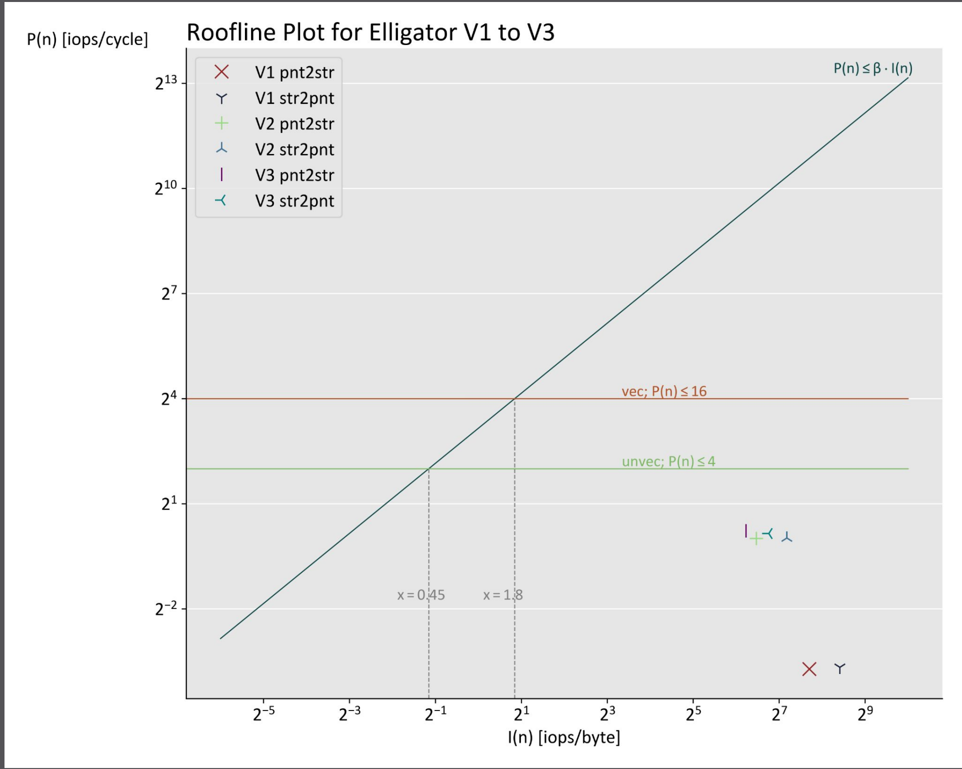
[1] Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B Intel Corporation, September 2019

Roofline Plot

- Peak performance
  - Without vectorization: 4 iops/cycle
  - With vectorization: 16 iops/cycle
    - Assuming 64-bit integers
- Memory bandwidth
  - Novabench: ≅25 GB/s
    - 8.9 B/cycle

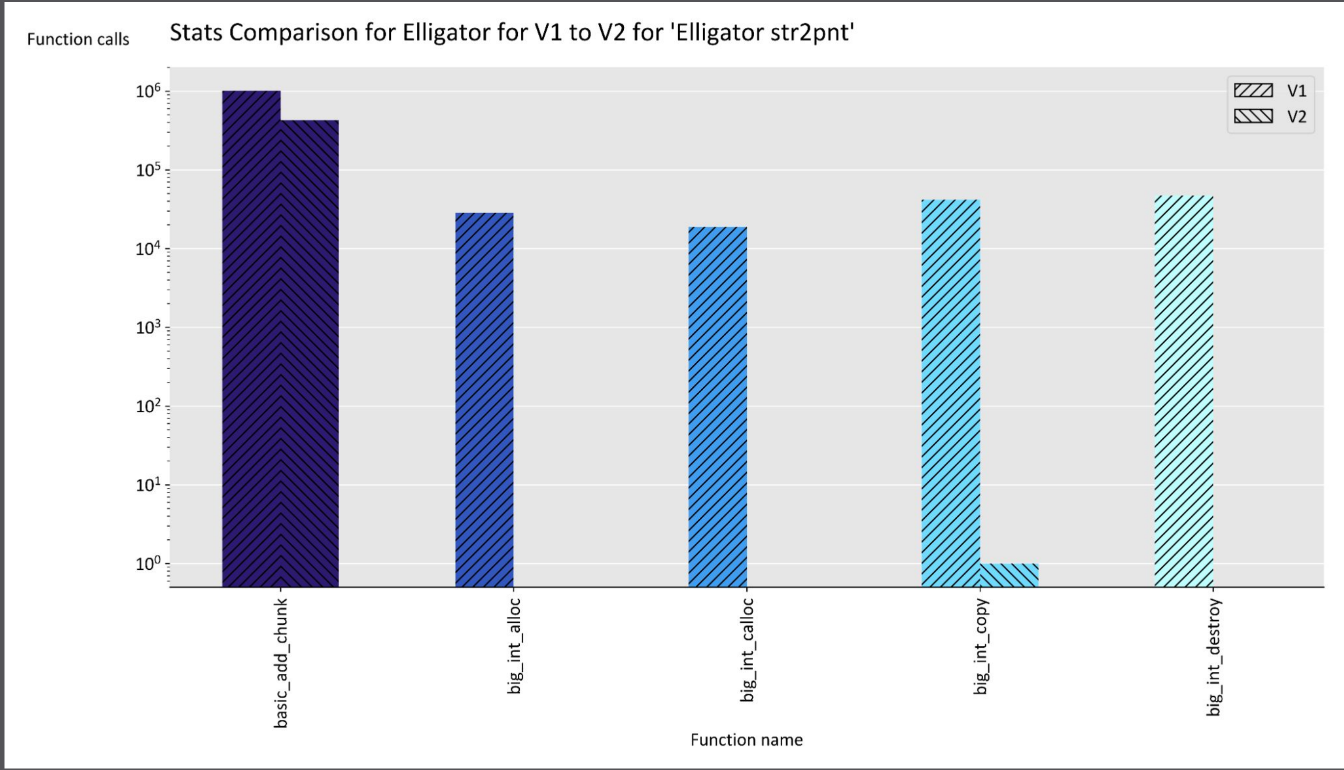
[1] Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B Intel Corporation, September 2019

Roofline Plot

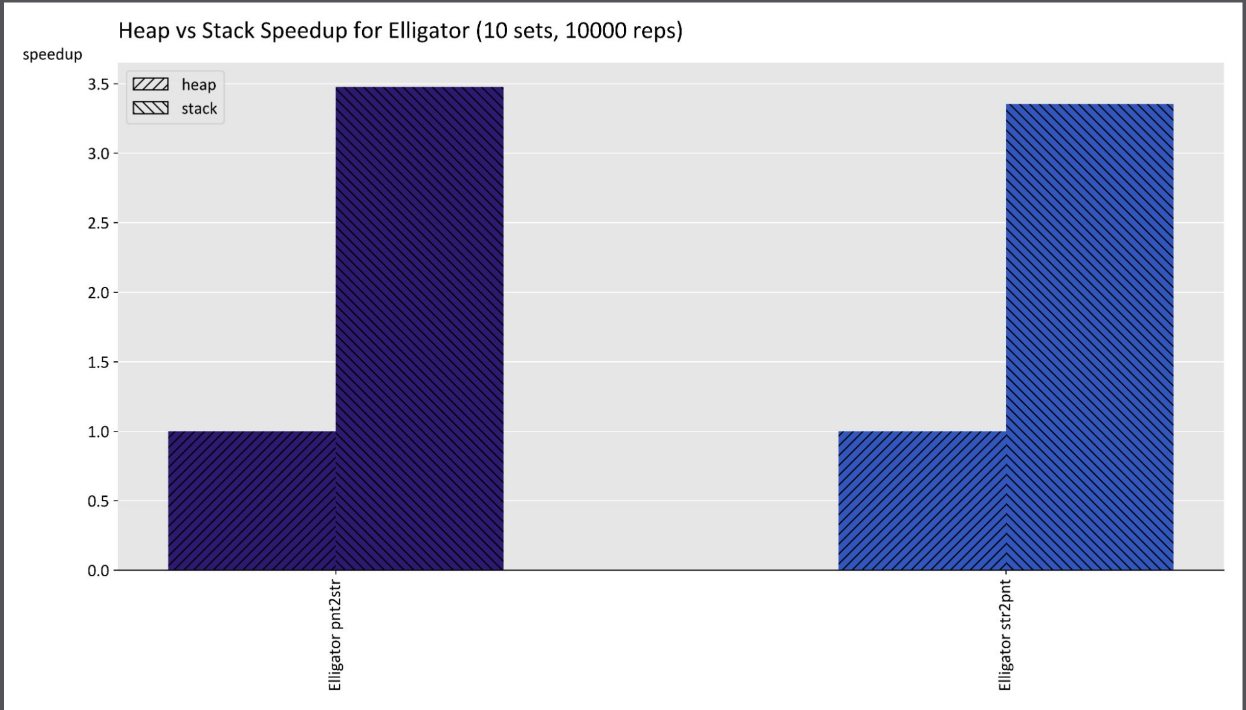


Non-Vector Optimizations

Memory Operations



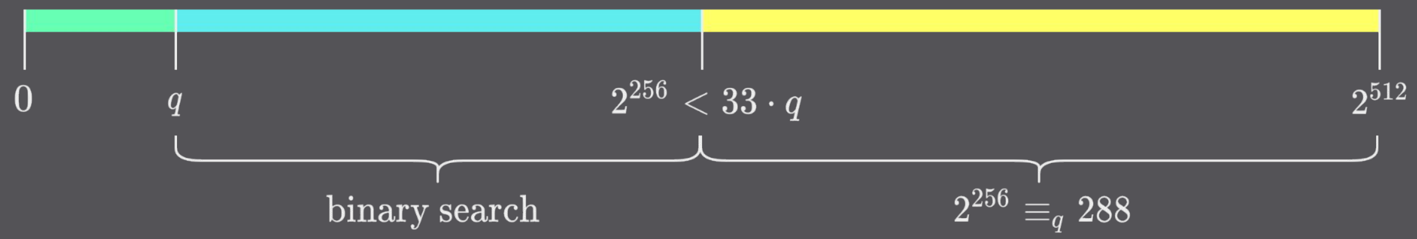
Stack vs Heap



Basic Optimizations

- Replace ‘mod power of 2’ with bitwise AND
- Replace power of 2 divisions by right-shift
- Assume no aliasing in BigInt parameters
- Create specific functions
  - Single chunk multiplication
  - Power with integer exponent
- Remove multiplications by χ
- Loop unrolling
- Pre-computation
- Optimization flags
- Compile all at once

Algorithmic Optimizations – *mod*



- Normally, *mod* requires division with rest
- Special prime of Curve1174
  - Recursion necessary
  - Only works for $X \leq 2^{256}$
- Binary search with precomputed values
  - Search $a \in [1, 32]$ s.t. $0 \leq X - aq < q$

$$q = 2^{251} - 9$$
$$\Rightarrow 2^{251} - 9 \equiv_q 0$$
$$\Rightarrow 2^{256} \equiv_q 288$$

$$X \in \{0,1\}^{512}$$
$$X = X_1 \cdot 2^{256} + X_0$$
$$= X_1 \cdot 288 + X_0$$

Algorithmic Optimizations – Square

- Special case of multiplication
  - Reduce memory access
    - Only one operand
  - Can save around half the chunk multiplications

			a_0a_3	a_0a_2	a_0a_1	a_0a_0
		a_1a_3	a_1a_2	a_1a_1	a_1a_0	
	a_2a_3	a_2a_2	a_2a_1	a_2a_0		
a_3a_3	a_3a_2	a_3a_1	a_3a_0			

Algorithmic Optimizations – special pow

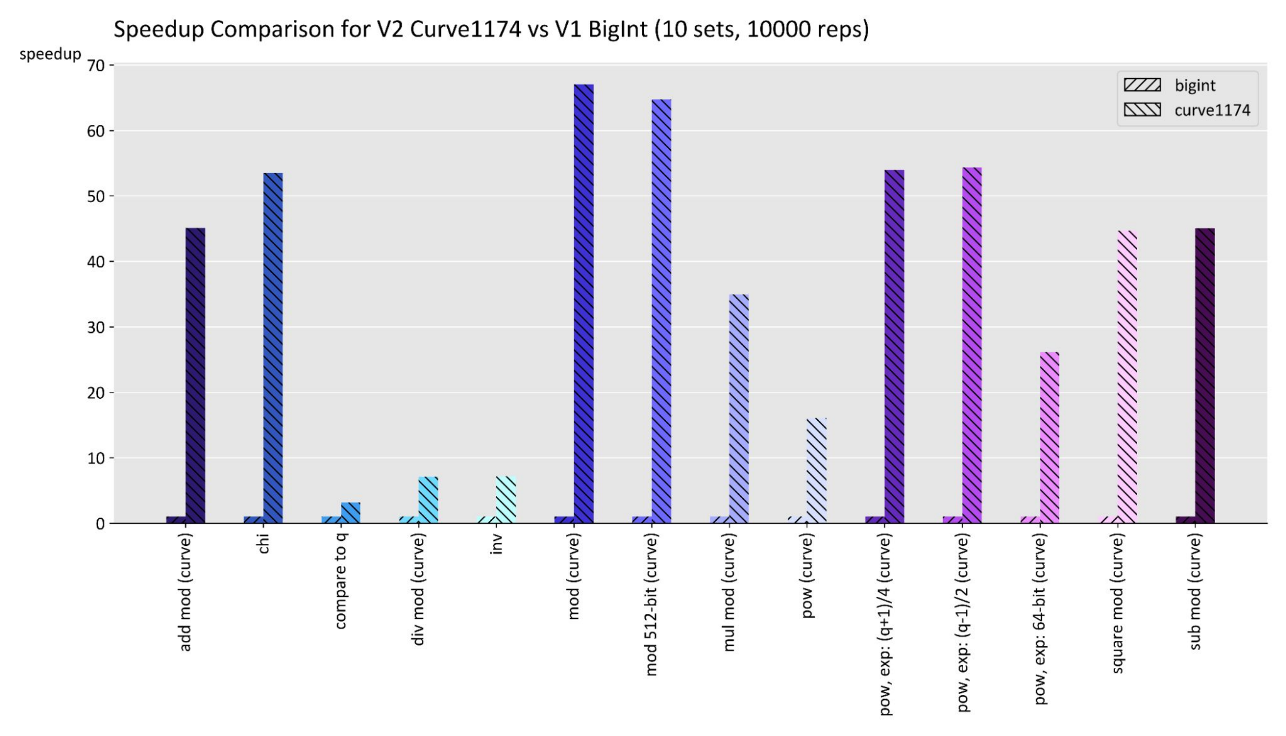
- Multiple special power operations
  - *Chi*: $\chi(a) = a^{(q-1)/2}$
  - Inverse mapping: $a^{(q+1)/4}$
  - Fermat inverse: $a^{-1} \equiv a^{q-2} \pmod q$
- Exponents have prefix of ‘ones’:
  - (q-1)/2 = 0b1111...11111011 (247 ones in prefix)
  - (q+1)/4 = 0b111111...111110 (248 ones in prefix)
  - q-2 = 0b11111...1110101 (247 ones in prefix)
- Ensure suffix separately
- Remove branching from square-and-multiply
- Enables AVX optimizations (later)

$\texttt{pow}(b, e)$:
  $r = 1$
  while e > 0
    if e & 1
      $r \equiv_q r \cdot b$
    $b \equiv_q b^2$
    $e = e/2$

$\Downarrow$

$\texttt{pow}(b, e)$:
  $r = 1$
  // ensure suffix
  while e > 0
    $r \equiv_q r \cdot b$
    $b \equiv_q b^2$
    $e = e/2$

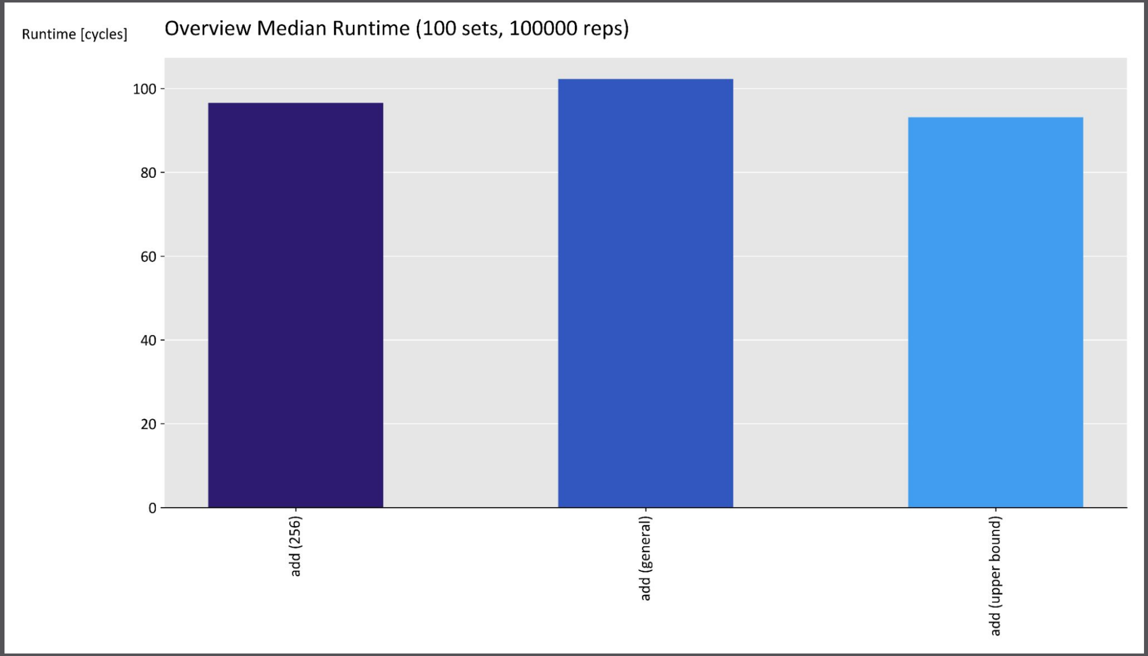
Algorithmic Optimizations – speedup



Vector Optimizations

AVX *add, sub, mul*

- Little benefit
- Carries
  - Manually over lanes
  - Needs #chunks ops
- Data movement
  - Costly in AVX



AVX *mul* 4 indep. inputs

- Linear dependency for square operations
- Result can use four variables r_1, r_2, r_3, r_4 for independent partial products
- Combine at end $r = r_1 \times r_2 \times r_3 \times r_4$
- Loop unrolling to avoid aliasing

```
// Ensure suffix

// Handle offset iterations

for (uint32_t i = 0; i < 30; ++i) {
    big_int_curve1174_square_mod(b_0_0, b_3_1);
    big_int_curve1174_square_mod(b_1_0, b_0_0);
    big_int_curve1174_square_mod(b_2_0, b_1_0);
    big_int_curve1174_square_mod(b_3_0, b_2_0);
    big_int_curve1174_mul_mod_4(
        r_0_0, r_1_0, r_2_0, r_3_0,
        r_0_1, r_1_1, r_2_1, r_3_1,
        b_0_0, b_1_0, b_2_0, b_3_0
    );

    big_int_curve1174_square_mod(b_0_1, b_3_0);
    big_int_curve1174_square_mod(b_1_1, b_0_1);
    big_int_curve1174_square_mod(b_2_1, b_1_1);
    big_int_curve1174_square_mod(b_3_1, b_2_1);
    big_int_curve1174_mul_mod_4(
        r_0_1, r_1_1, r_2_1, r_3_1,
        r_0_0, r_1_0, r_2_0, r_3_0,
        b_0_1, b_1_1, b_2_1, b_3_1
    );
}

// Combine variables
```

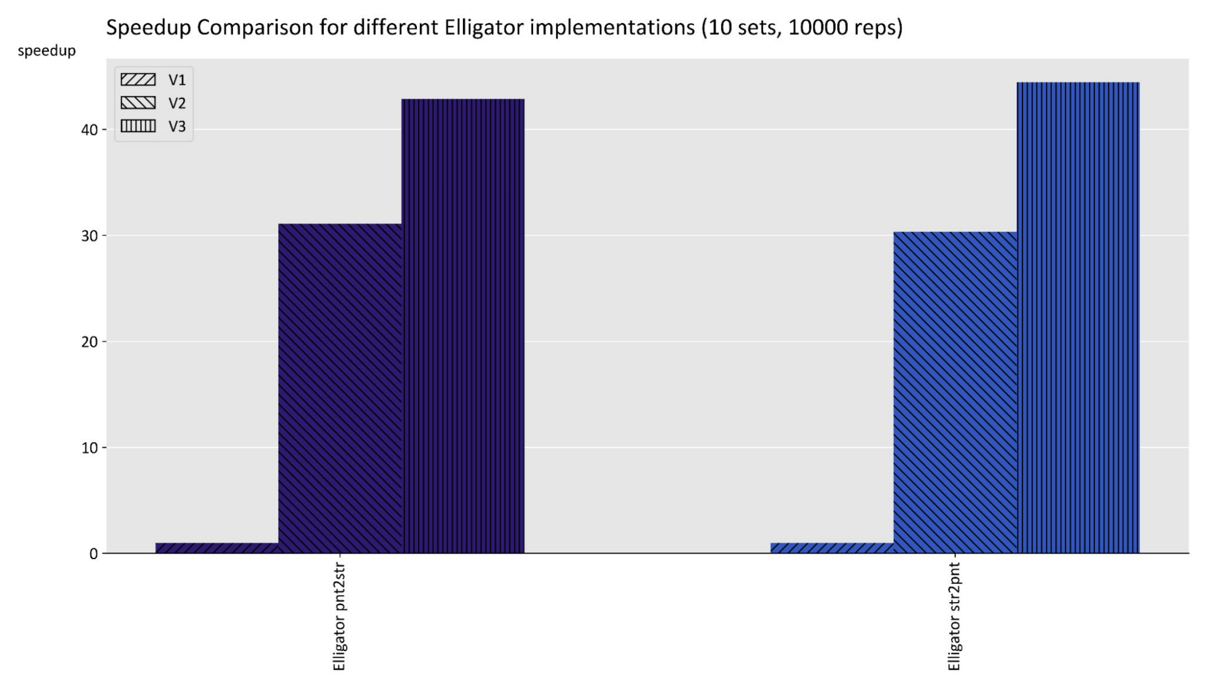
AVX *mul* 4 indep. inputs

- Pack data
  - The same chunk from 4 different BigInts are adjacent
- Do the normal mul algorithm with vector instructions
  - No need to move data horizontally
- Unpack the data

- Moderate speed
  - Packing/Unpacking is an overhead

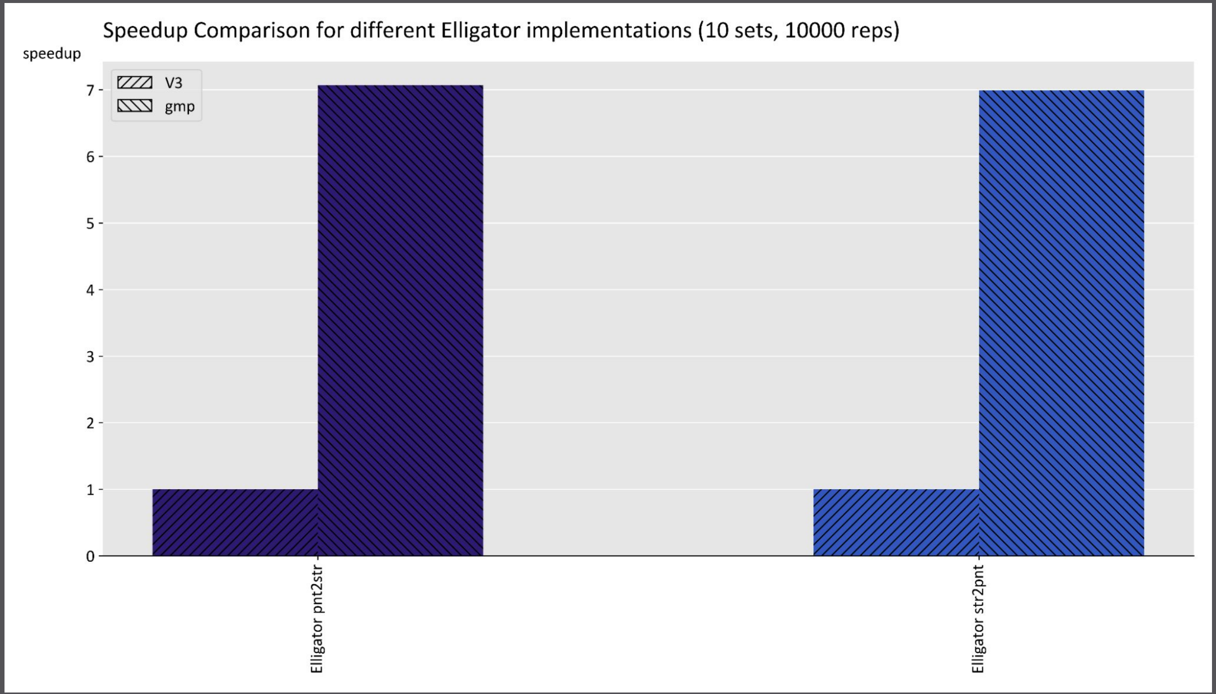
Conclusion

Overall speedup

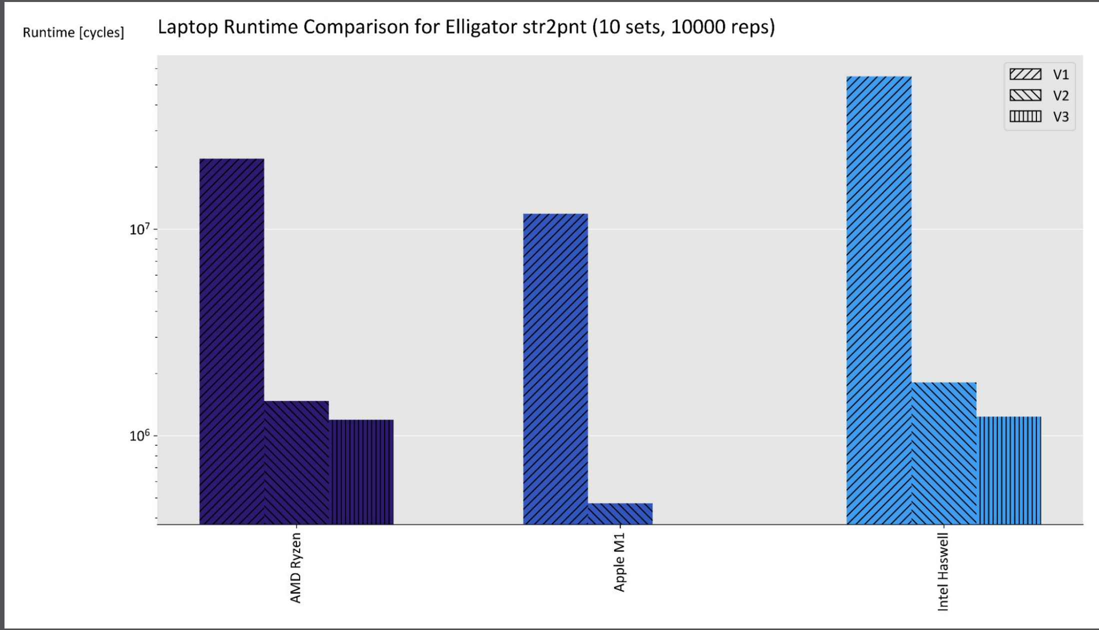


44x

Comparison to GMP



Laptop Comparison



- Devices:
  - MacBook Pro Mid 2015 with Intel Haswell i7-4980HQ 2.8 GHz, native clang
  - MacBook Pro 2020 with Apple M1, native clang
  - AMD Ryzen 9 3900X @4.1GHz, Win10 WSL Ubuntu and gcc