NETHERMIND
nethermind.io

eryx CRYPTO ZK
eryx.co

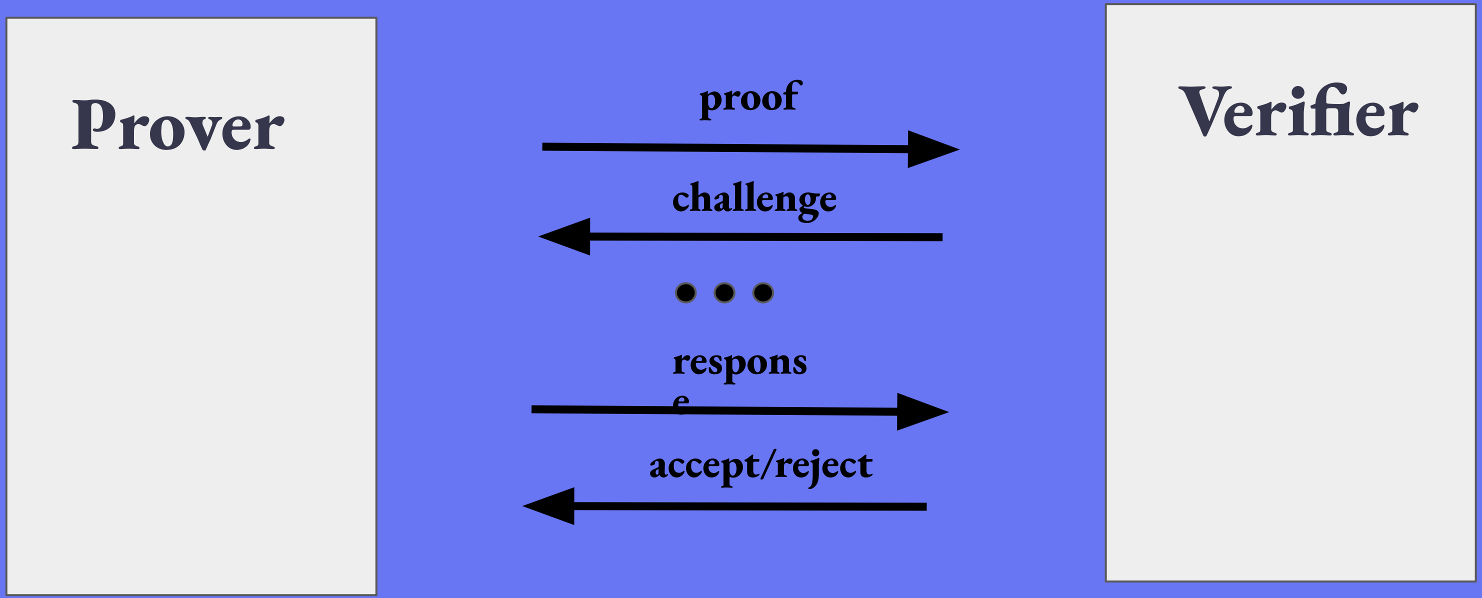
# Circle STARK GPU Acceleration

An Analysis of Performance and Implementation

01

# Circle STARK

# Proof Systems

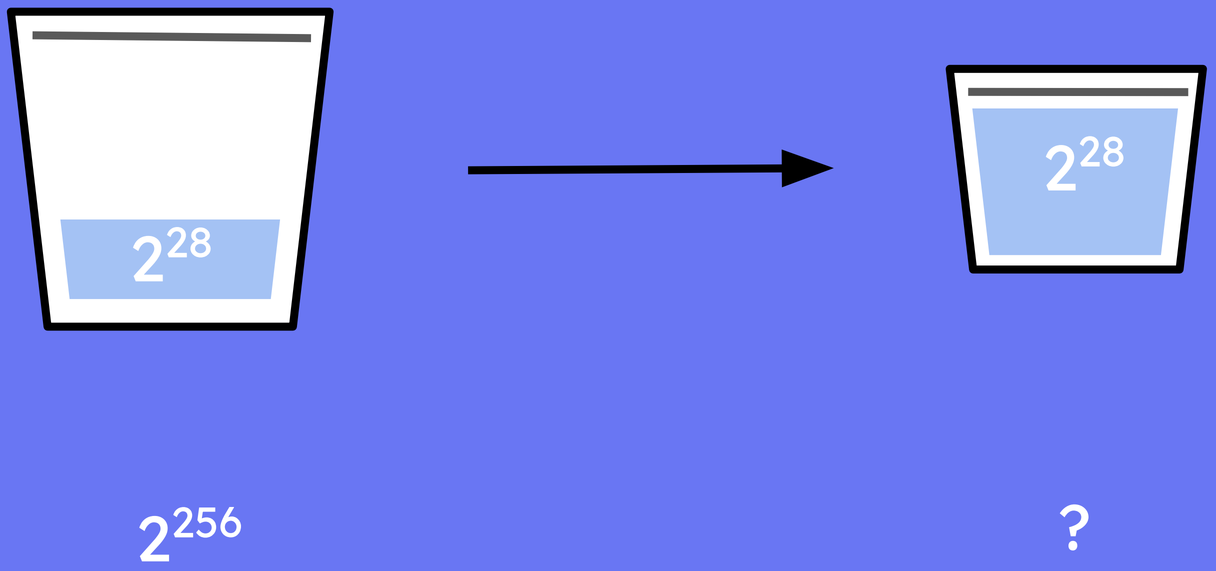


# ZK-STARK

- **Z**ero-**K**nowledge
- **S**calable
- **T**ransparent
- **Ar**gument
- of **K**nowledge

# A Matter of Efficiency

$2^{256} - 2^{28} = 2^{228}$ Bits of Unused Space!



# A Matter of Efficiency

Prime Fields

- STARK field prime : $p = 2^{251} + 17 * 2^{192} + 1$
- Goldilocks prime: $p = 2^{64} – 2^{32} + 1$
- Babybear prime: $p = 15 * 2^{27} + 1$
- Mersenne 31 Prime: $p = 2^{31} - 1$

# Why the Circle?

1. Map the field onto coordinates of the circle
2. Circle gives us the extra point for FFT

# A Matter of Efficiency

1. Mersenne 31 is very efficient on 32-bit architecture
2. The Circle enables FFTs
3. 1.4x performance increase over Babybear

# 02

# GPU Optimization

# GPU parallelization

GPUs can handle many simultaneous calculations.

1. Data-intensive computations
2. Algebraic operations
3. Image or signal processing

# GPU problems

The GPU and CPU use separate memory spaces

1. One cannot access the other's memory
2. Copying data from one to the other is expensive

# GPU problems

Memory accessing is not trivial

1. GPU threads are grouped in blocks
2. Blocks cannot access each other's *shared* memory either

# GPU benefits

When we launch threads to process our data

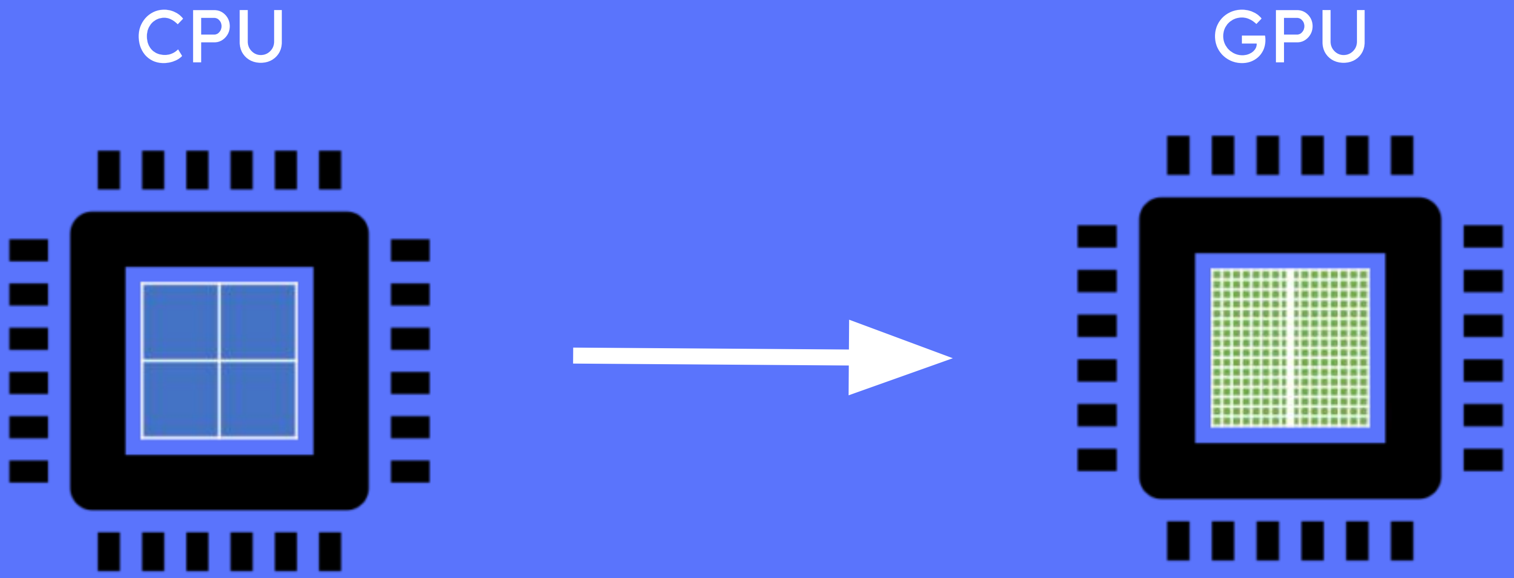
1. Threads will run concurrently
2. The GPU will handle as many threads in parallel as it can.

# GPU benefits

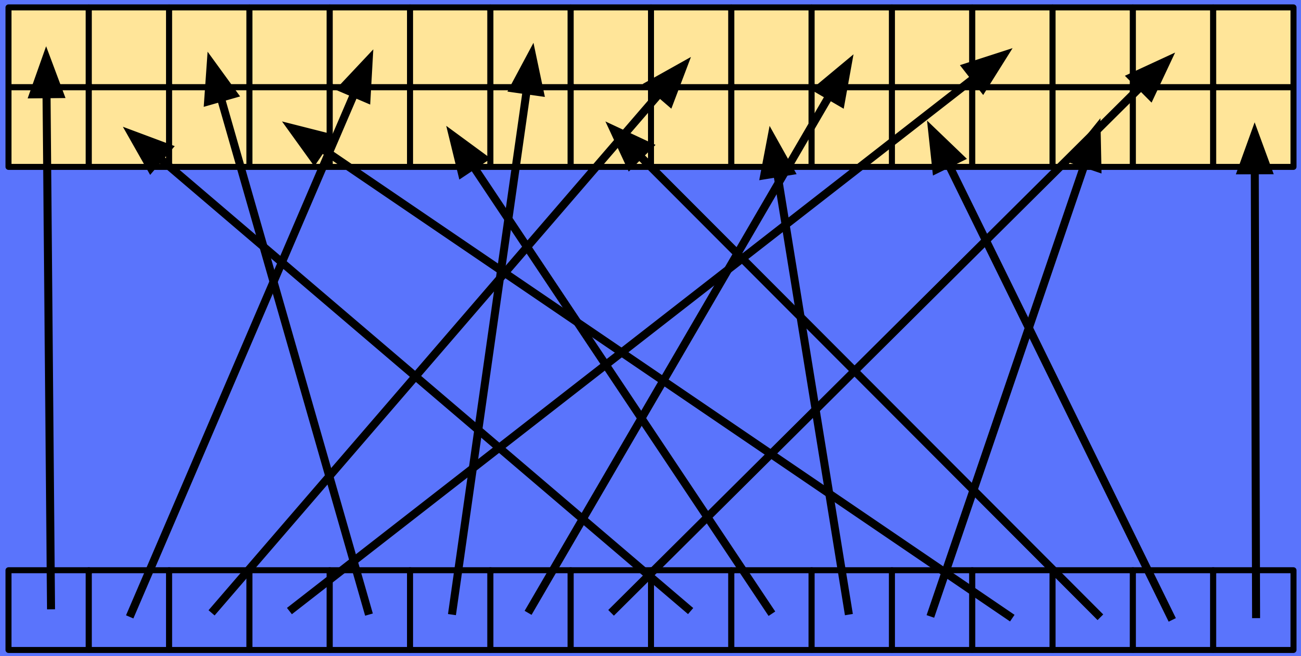
There are a lot of architecture-specific features.

1. Blocks and warps
2. Memory banks, global memory, shared memory
3. Consecutive memory access operations
4. Shared registers

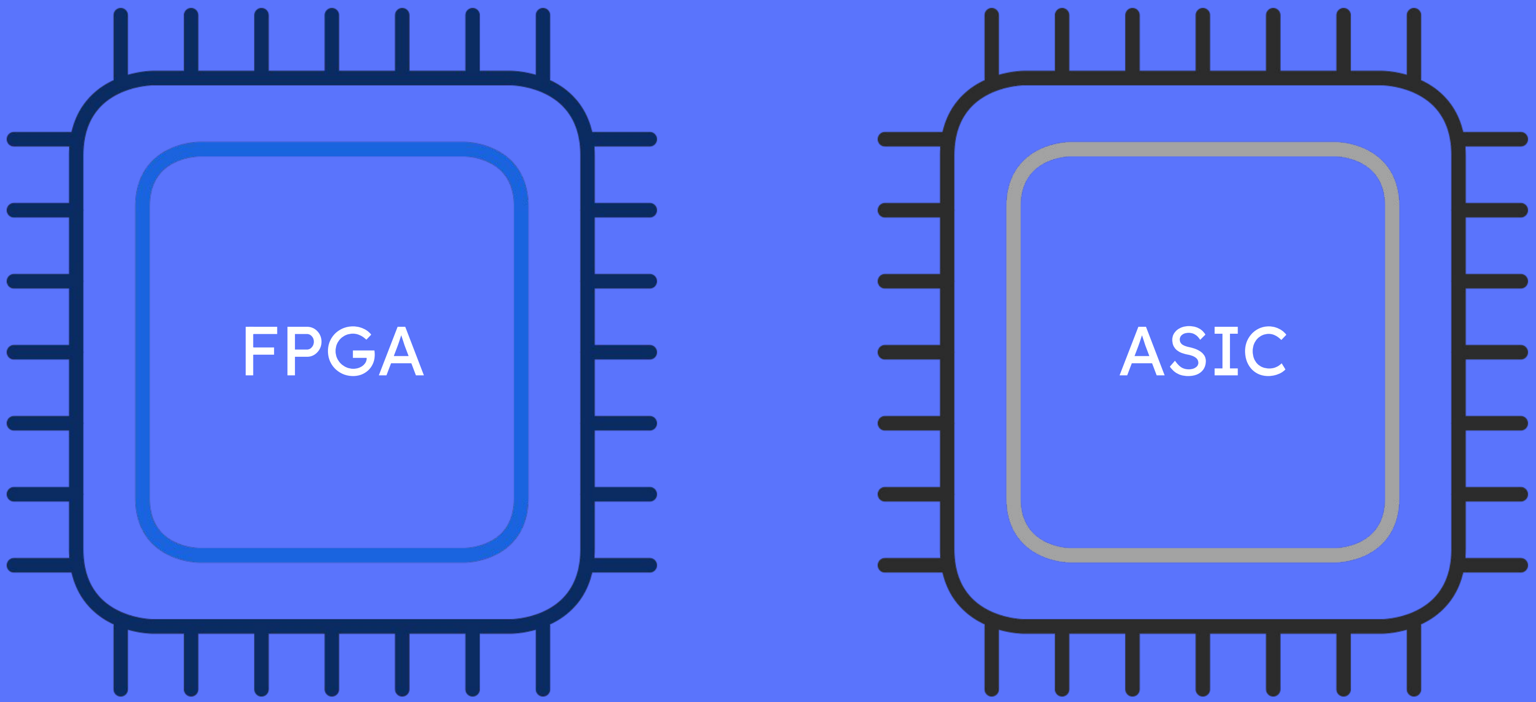
# GPU Data Transfer



# GPU Memory Access



# Other Hardware



# 03

# Case Study

01. **Sequential algorithm**

02. **Parallel algorithm**

03. **Comparison**

# Batch inverse

It's a Stwo component we'll parallelize.

- Its purpose is to calculate the multiplicative inverse in the field for a batch of numbers.

# Sequential algorithm

We'll first review the sequential implementation.

- Since field inversion is an expensive operation (extended euclidean algorithm), we use something called Montgomery's trick.

# Montgomery's trick

$$a_1, a_2, a_3, a_4 \in F_p$$

# Montgomery’s trick

$$\beta_1 = a_1$$

$$\beta_2 = a_1 \times a_2$$

$$\beta_3 = a_1 \times a_2 \times a_3$$

$$\beta_4 = a_1 \times a_2 \times a_3 \times a_4$$

# Montgomery's trick

$$\beta_1 = a_1$$

$$\beta_2 = \beta_1 \times a_2$$

$$\beta_3 = \beta_2 \times a_3$$

$$\beta_4 = \beta_3 \times a_4$$

# Montgomery’s trick

$$\beta_4^{-1} \longleftarrow eea(\beta_4)$$

# *Intuitively

$$a_4^{-1} = \beta_4^{-1} \times \beta_3$$

$$= \frac{1}{a_1 \times a_2 \times a_3 \times a_4} \times a_1 \times a_2 \times a_3$$

$$= \frac{1}{a_4}$$

# *Intuitively

$$\beta_3^{-1} = \beta_4^{-1} \times a_4$$

$$= \frac{1}{a_1 \times a_2 \times a_3 \times a_4} \times a_4$$

$$= \frac{1}{a_1 \times a_2 \times a_3}$$

# Montgomery's trick

$$a_4^{-1} = \beta_4^{-1} \times \beta_3$$
$$\beta_3^{-1} = \beta_4^{-1} \times a_4$$

$$a_3^{-1} = \beta_3^{-1} \times \beta_2$$
$$\beta_2^{-1} = \beta_3^{-1} \times a_3$$

$$a_2^{-1} = \beta_2^{-1} \times \beta_1$$
$$\beta_1^{-1} = \beta_2^{-1} \times a_2$$

$$a_1^{-1} = \beta_1^{-1}$$

# Montgomery’s trick

For an array of size *n* it replaces *n* field inversions with *3n* field multiplications + *1* field inversion.

- *n* multiplications for the acc. products
- *1* inversion of the accumulated product
- *2* multiplications to invert each element which warrants the change

# Montgomery’s trick

It cannot be parallelized as is.

Calculating any of the accumulated products requires the previous value.

So we’ll adapt the algorithm for its parallelization.

# Adapted trick

$$a_1, a_2, a_3, a_4 \in F_p$$

But suppose the amount is a power of two.

# Adapted trick

$$a_1 \quad a_2 \qquad a_3 \quad a_4$$

$$\beta_{1,2} = a_1 \times a_2 \qquad \beta_{3,4} = a_3 \times a_4$$

$$\beta_{1,4} = a_1 \times a_2 \times a_3 \times a_4$$

# Adapted trick

- In the same level of the tree, all multiplications are independent.
  They can be parallelized.

# Adapted trick

$$\beta_{1,4}^{-1} \longleftarrow eea(\beta_{1,4})$$

# Adapted trick

$$\beta_{1,4}^{-1}$$

$$\beta_{1,2}^{-1} = \beta_{1,4}^{-1} \times \beta_{3,4}$$

# *Intuitively

$$\beta_{1,2}^{-1} = \beta_{1,4}^{-1} \times \beta_{3,4}$$

$$= \frac{1}{a_1 \times a_2 \times a_3 \times a_4} \times a_3 \times a_4$$

$$= \frac{1}{a_1 \times a_2}$$

# Adapted trick

$$\beta_{1,4}^{-1}$$

$$\beta_{1,2}^{-1} = \beta_{1,4}^{-1} \times \beta_{3,4} \qquad \beta_{3,4}^{-1} = \beta_{1,4}^{-1} \times \beta_{1,2}$$

$$a_1^{-1} = \beta_{1,2}^{-1} \times a_2 \qquad a_3^{-1} = \beta_{3,4}^{-1} \times a_4$$

$$a_2^{-1} = \beta_{1,2}^{-1} \times a_1 \qquad a_4^{-1} = \beta_{3,4}^{-1} \times a_3$$

# Adapted trick

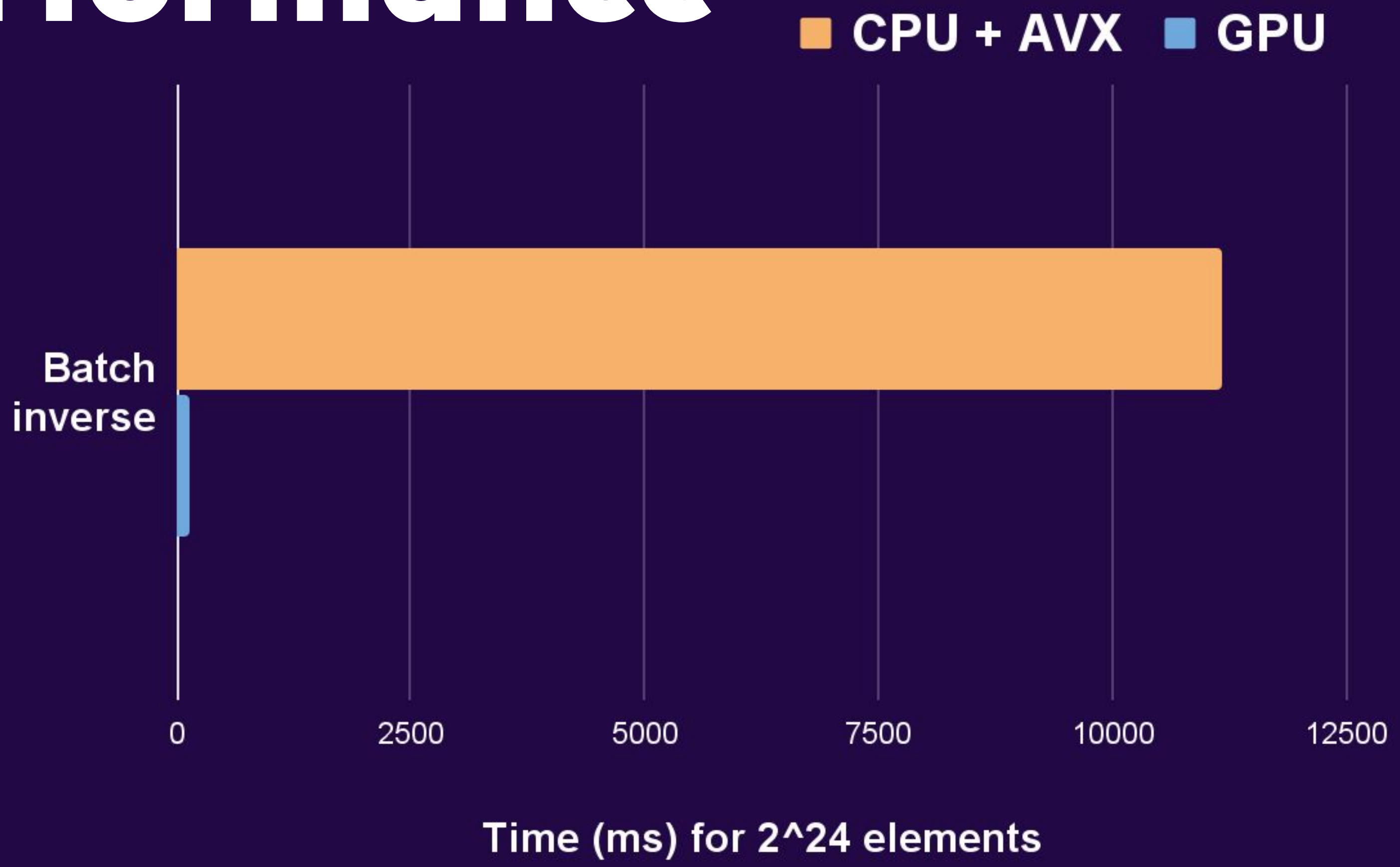
- Once again, each level of this tree can be parallelized.

# Adapted trick

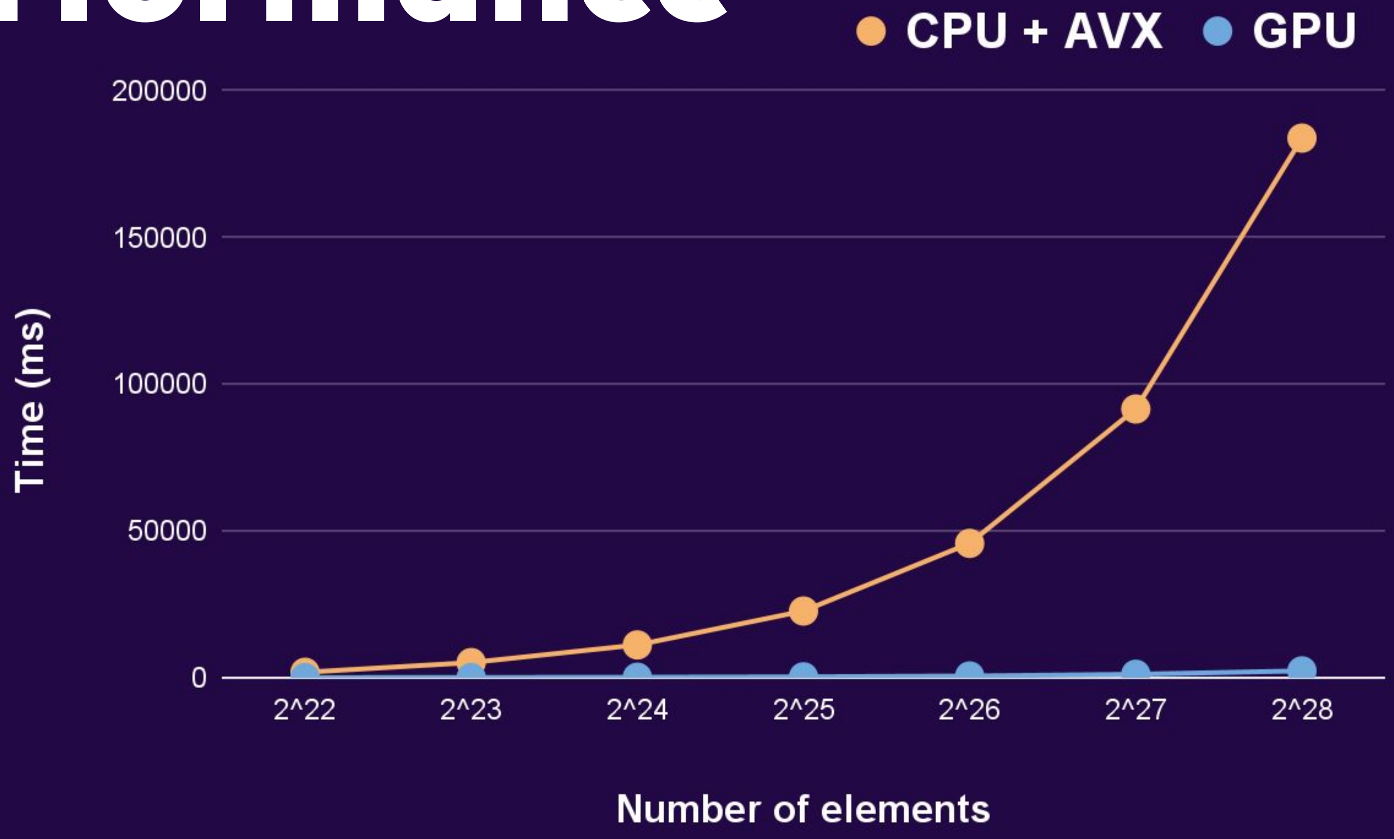
Even more CUDA-specific optimizations can be applied to it:

1. Use of shared memory
2. Use of warps to invert 32 elements at once
3. Use of consecutive memory access

# Performance



# Performance

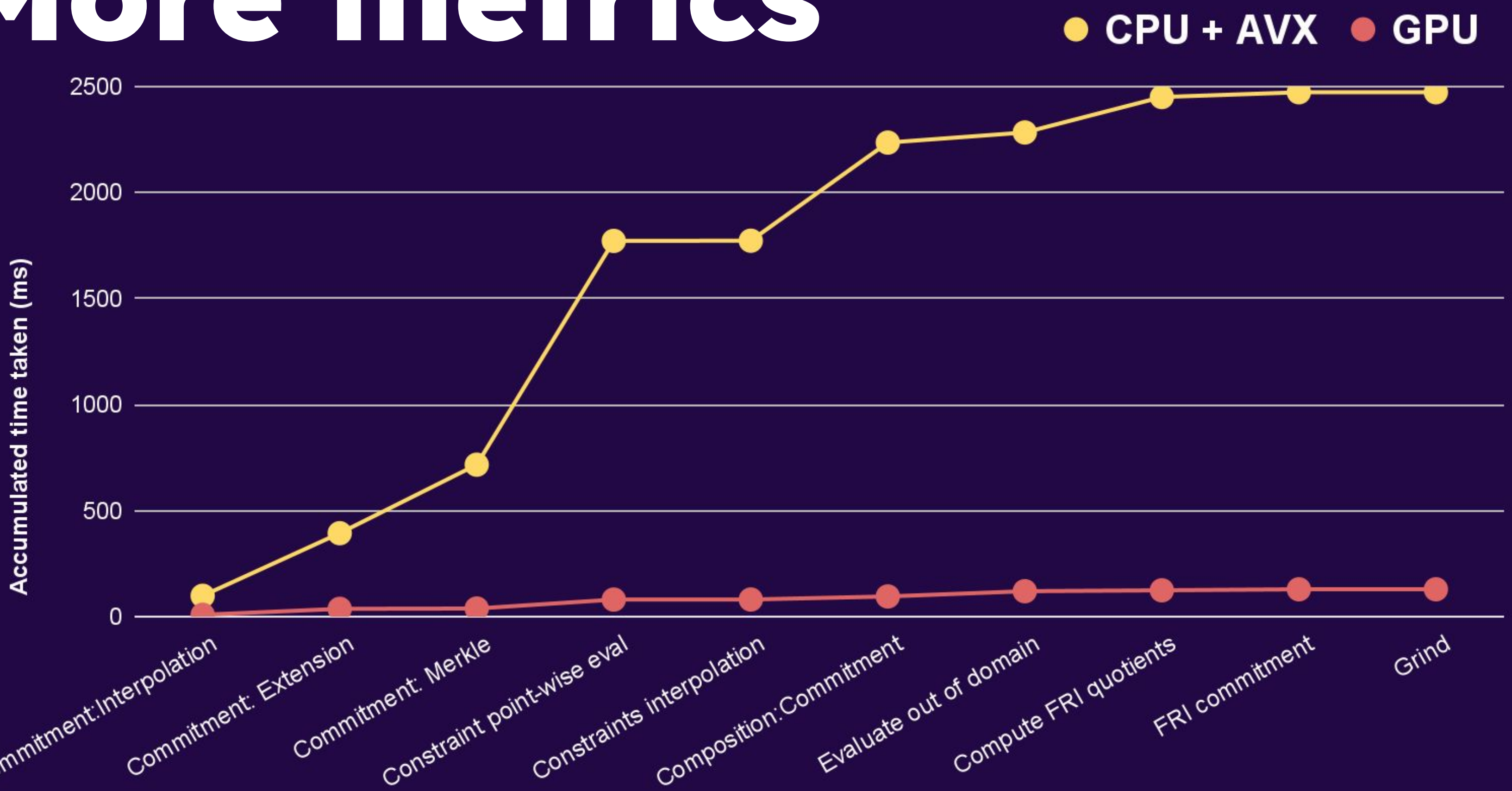


# What we learned

1. GPU parallelization is a very powerful tool to make our algorithms quicker and cheaper to run.

2. A simple heuristic goes a long way in harnessing the power of both high-performance and commodity hardware GPUs.

# More metrics




* Wide Fibonacci with Blake, with 2^(10) columns of size 2^(16) each.

# Summary

1

Circle STARKs utilize smaller fields to balance security and efficiency

2

Harnessing GPU power is nowadays vital for the computation of cryptography primitives.

3

We should keep an eye on future hardware acceleration trends.

# Due thanks

# Thank you!

## Contact

@nethermindeth

@eryxcoop