Team 58 Presentation

Richard Wohlbold
Aljaz Medic
Aleksa Micanovic
Aleks Stepancic

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Typical Organization I

- **Algorithm that you consider (maybe 2 slides)**
  - State problem that it solves (input:…, output: …)
  - If possible visualize how it works or show high-level pseudocode
  - State asymptotic runtime
- **Cost analysis (cost measure, exact count)**
- **Baseline implementation (briefly explain), maybe show already performance plot and extract percentage of peak**
- **Optimizations you performed**
  - Briefly discuss major optimizations/code versions
  - Maybe explain the most interesting in a bit greater detail
  - Any analysis (e.g., profifling) you performed is interesting – show the result
  - If too much, explain only some things and just state the rest

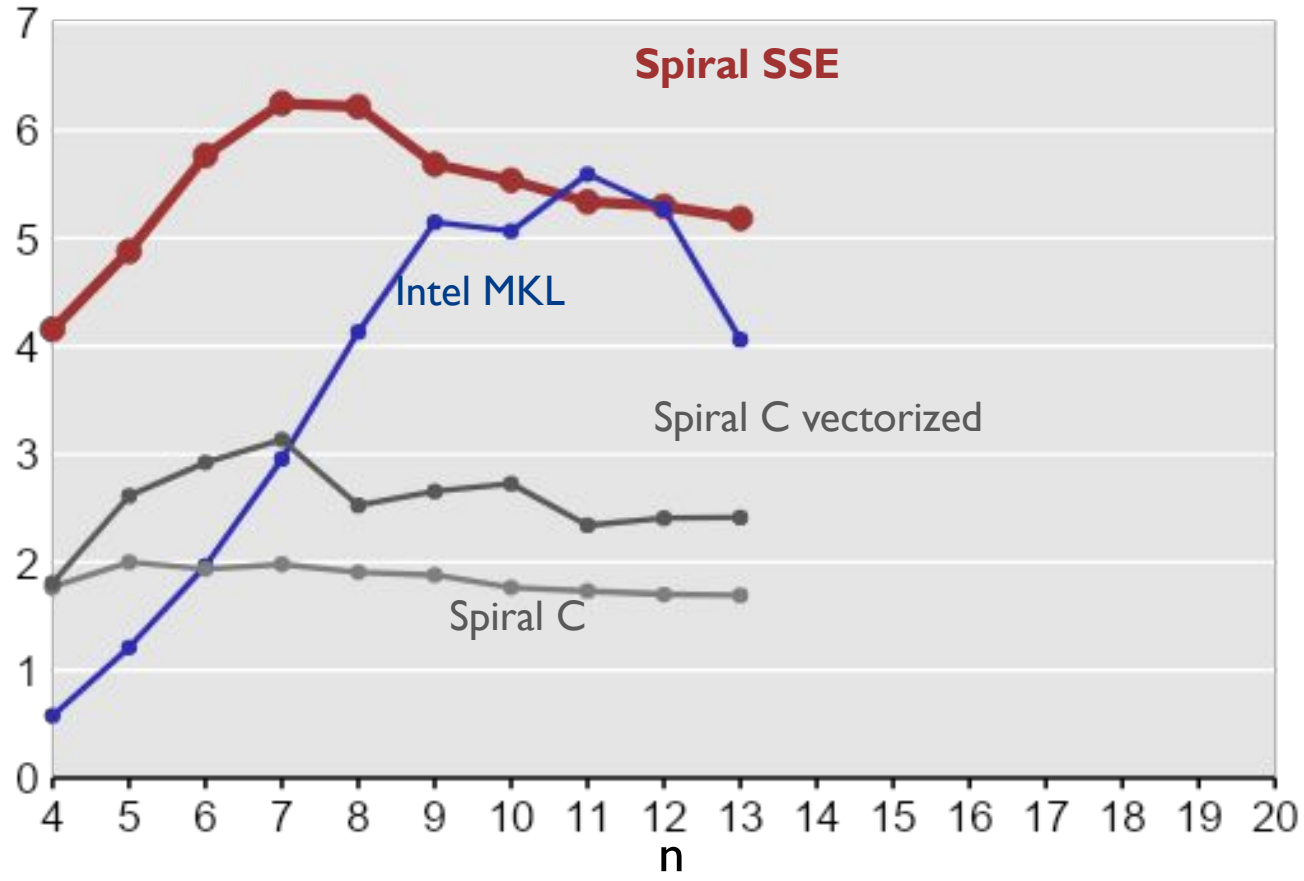
Typical Organization II

- **Experimental results**
  - Very brief: Experimental setup (platform, compiler)
  - Performance plot over a range of sizes with different code versions
  - Make sure you also push input size to the limit in the experiments
  - Extract overall speedup

- **Every project is different – so adapt as needed**
- **Focus on the most interesting things, don’t explain everything that will be in the final report.**

Try to Make Nice Plots

DFT 2^n (single precision) on Pentium 4, 2.53 GHz
[Gflop/s]

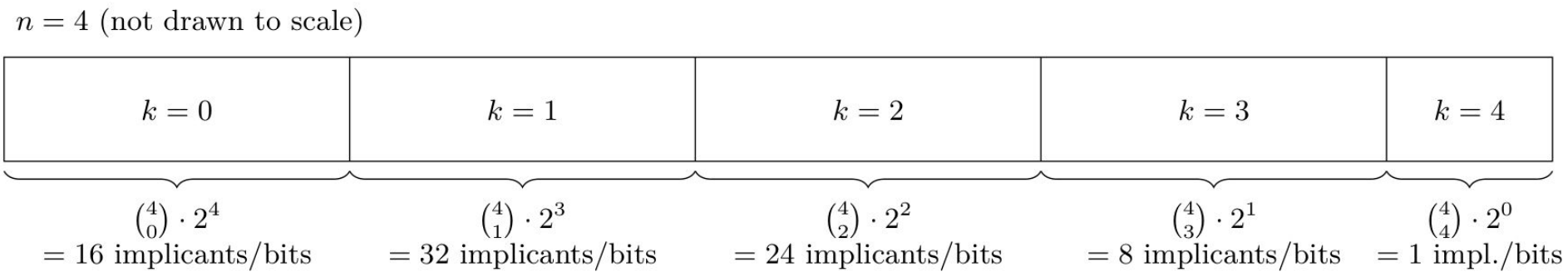


Prime Implicants for Quine-McCluskey

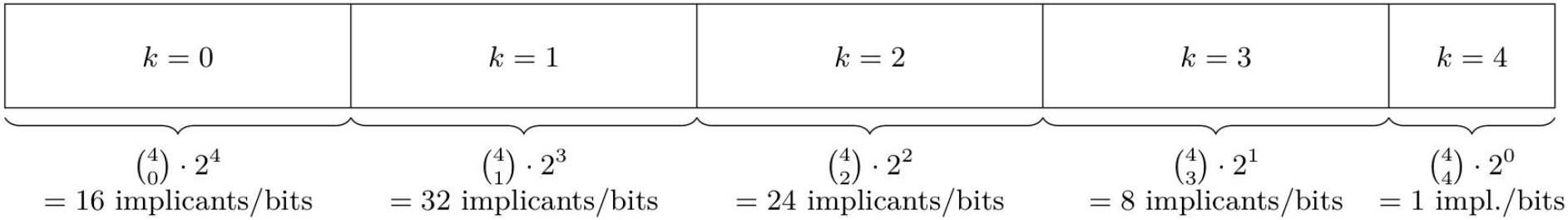
- The algorithm determines all prime implicants of a boolean function, given the initial set of minterms
- **Input:** number of bits and list of minterms
  (e.g. num_bits=7 0010011, 100101, …)
- **Output:** list of all prime implicants

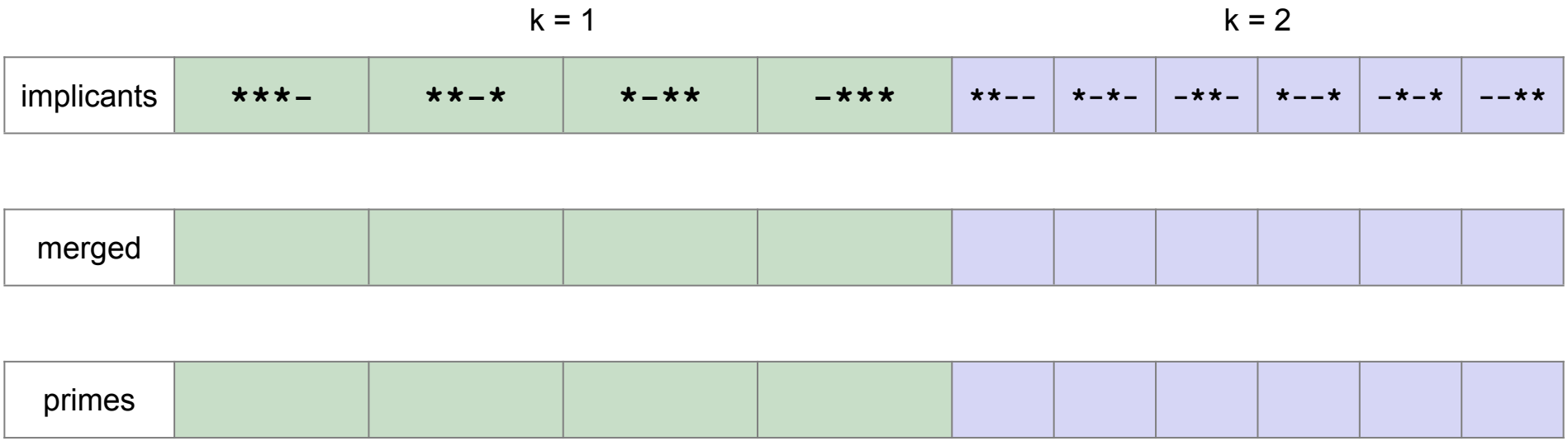
Prime Implicants for Quine-McCluskey

- **Every implicant has a bit that describes it in a 3^n array**
- **There are 3 such arrays: implicants, merged, primes**
- **Each is divided into n+1 sections made up of chunks**
  - For 0 <= k <= n, the kth section contains :
    - *all implicants with k*
    - *(n choose k) many chunks (all implicants have dashes in the same place)*
    - *chunks of size 2**(n-k) bits*
- **Chunks within the same section have a combinatorial ordering, based on the location of the dashes**
- **A single merge step takes one chunk of input and produces multiple chunks of output**

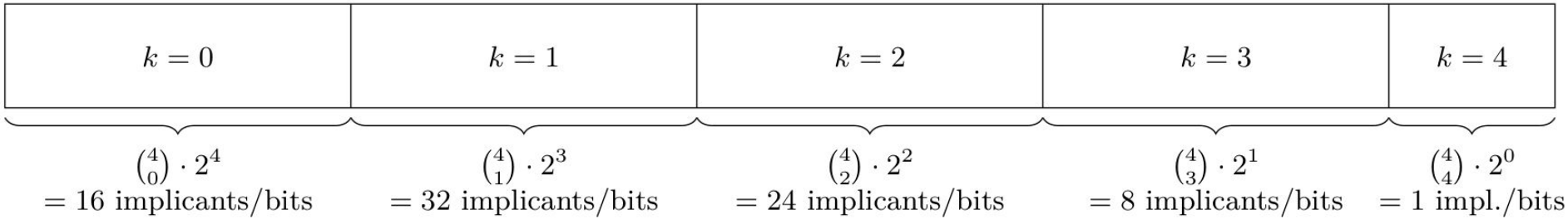


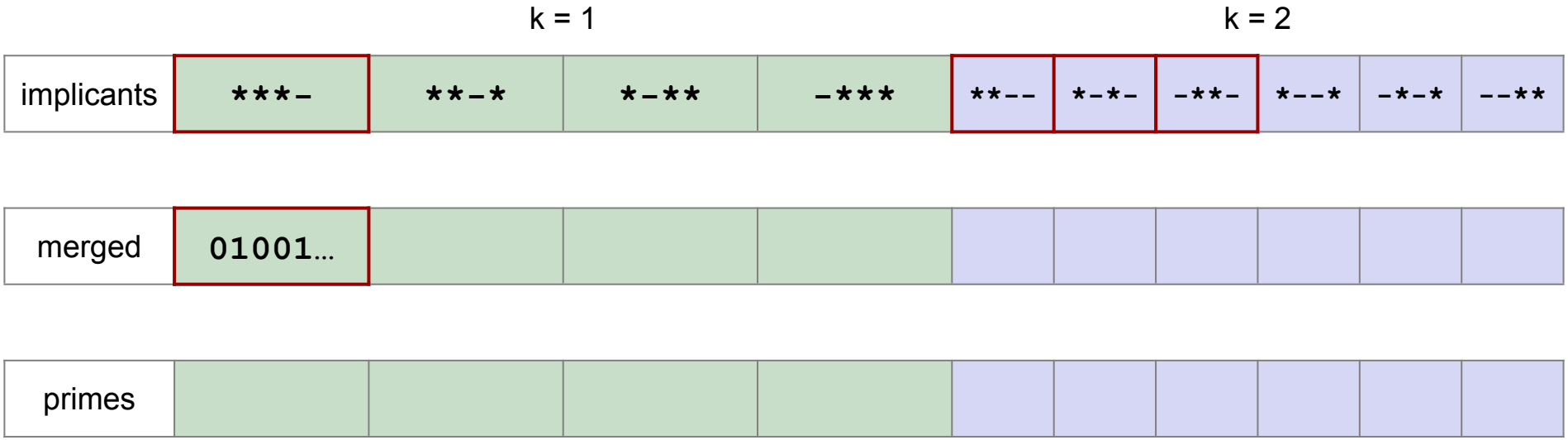
$n = 4$ (not drawn to scale)



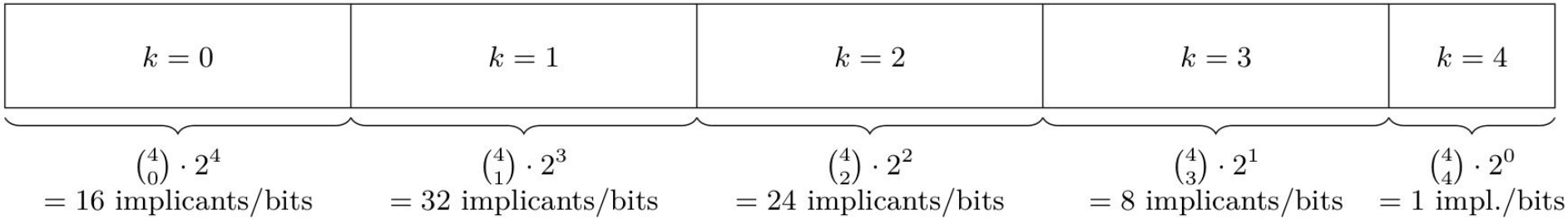


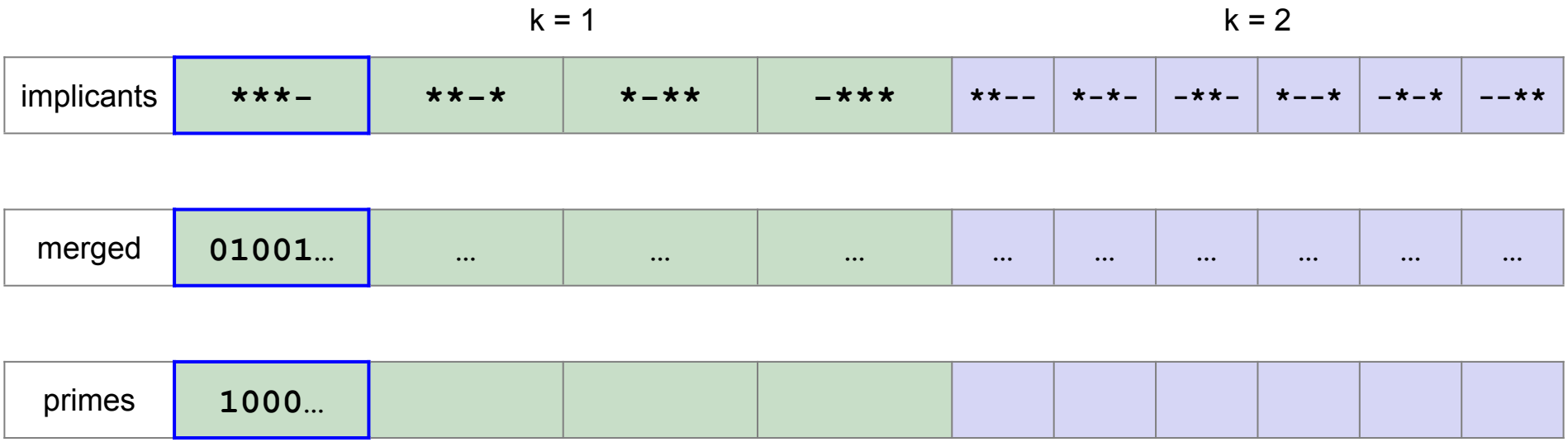
$n = 4$ (not drawn to scale)





$n = 4$ (not drawn to scale)





Dummy Baseline “Dense Byte”

- Two phase approach
  1. **Generation (Merge):** AND pair of implicants that differ by one bit
  2. **Scan:** Find implicants that are not merged
- **Drawback:**

  “Dense byte” stores **one implicant per byte**

Optimization #1 “64bit vectorization”

Improvement:

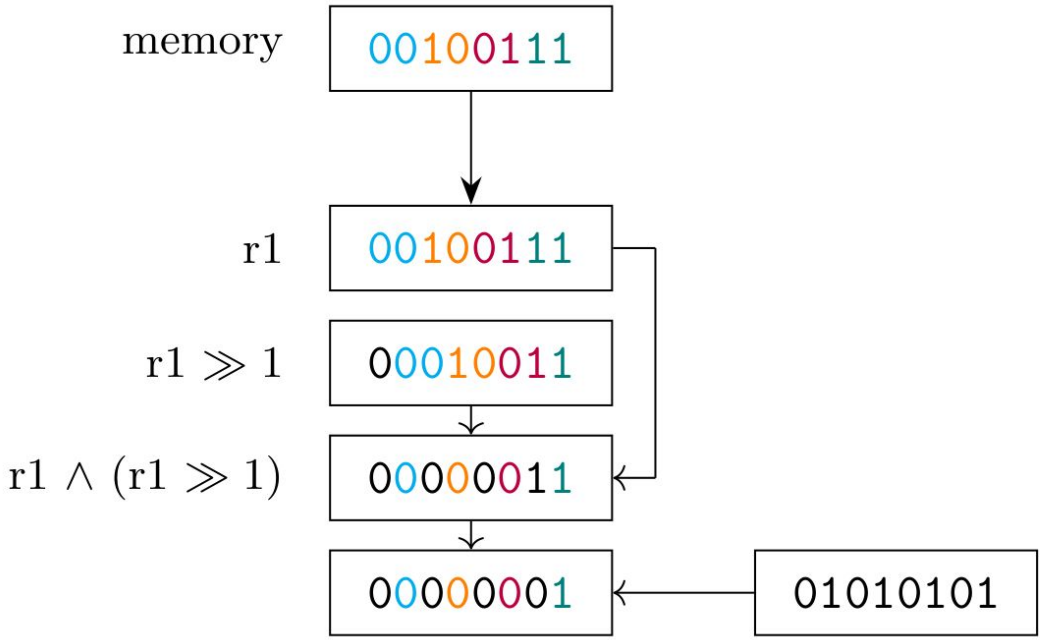
Store 64 implicants in **64 bit registers**

Block sizes >= 64, merging **EASY**

Problem:

Block sizes < 64, need to find approach to extract the merged bits within the register.

$n \geq 3$, block size: 1 bit, register width: 8 bits



How to reorganize bits into lower half, i.e. turn 01010101 into 00001111 ?

`unsigned __int64 _pext_u64 (unsigned __int64 a, unsigned __int64 mask)`

Synopsis

```
unsigned __int64 _pext_u64 (unsigned __int64 a, unsigned __int64 mask)
#include <immintrin.h>
Instruction: pext r64, r64, r64
CPUID Flags: BMI2
```

Description

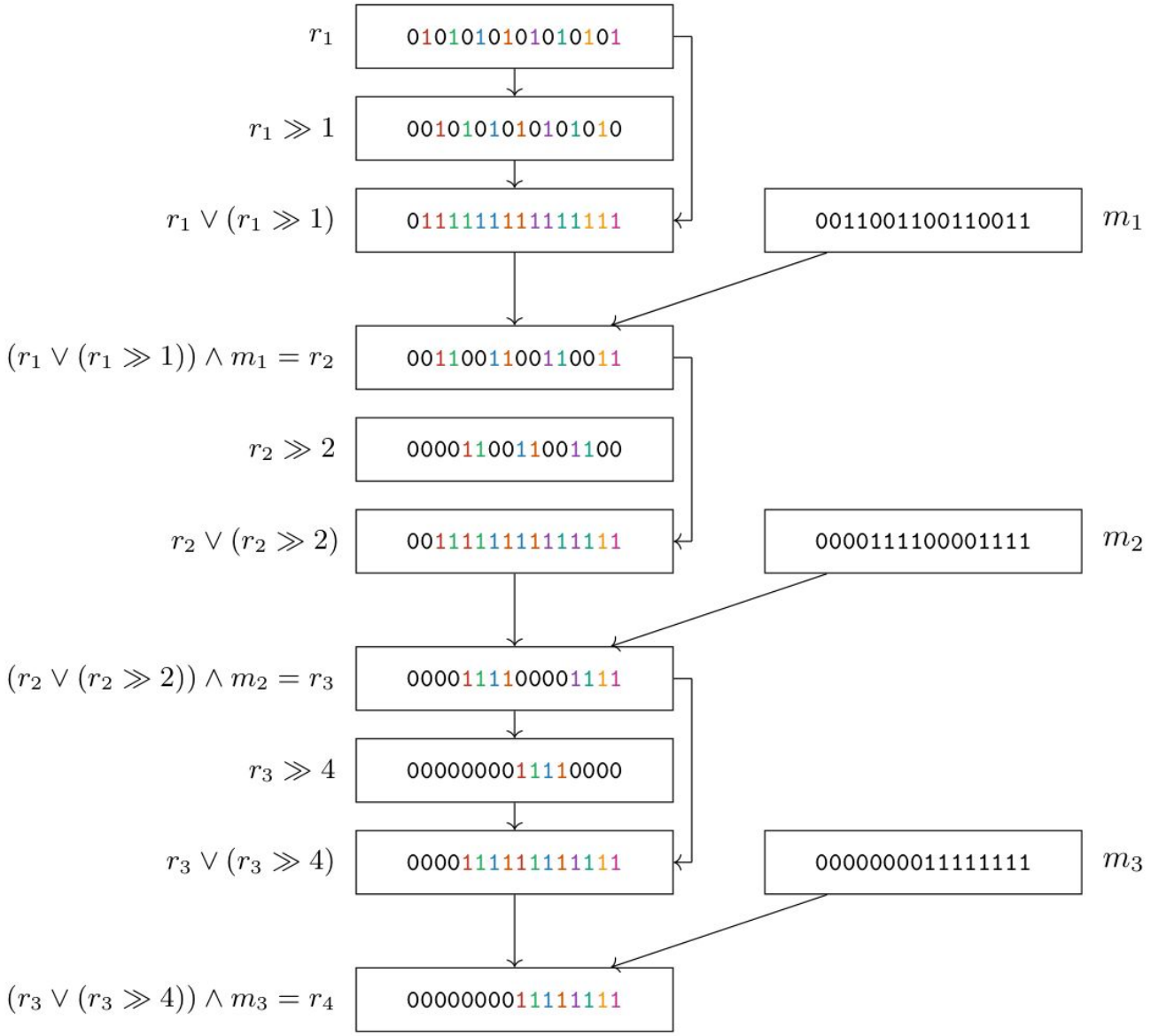
Extract bits from unsigned 64-bit integer `a` at the corresponding bit locations specified by `mask` to contiguous low bits in `dst`; the remaining upper bits in `dst` are set to zero.

Operation

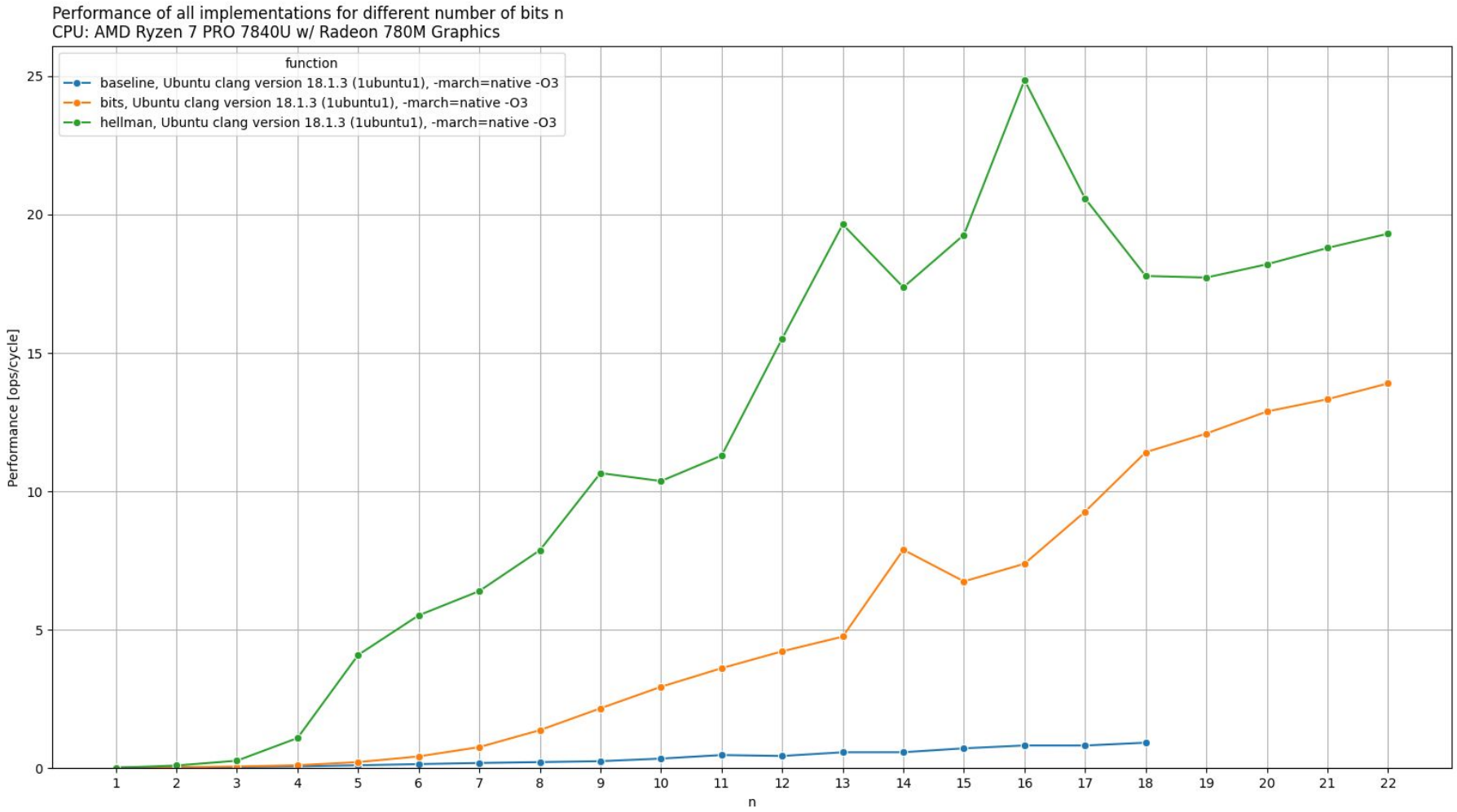
```
tmp := a
dst := 0
m := 0
k := 0
DO WHILE m < 64
        IF mask[m] == 1
                dst[k] := tmp[m]
                k := k + 1
        FI
        m := m + 1
OD
```

Latency and Throughput

Architecture	Latency	Throughput (CPI)
Alderlake	3	1
Icelake Intel Core	3	1
Icelake Xeon	3	1
Sapphire Rapids	3	1
Skylake	3	1


r_1
0101010101010101
$r_1 \gg 1$
0010101010101010
$r_1 \vee (r_1 \gg 1)$
0111111111111111
0011001100110011
m_1
$(r_1 \vee (r_1 \gg 1)) \wedge m_1 = r_2$
0011001100110011
$r_2 \gg 2$
0000110011001100
$r_2 \vee (r_2 \gg 2)$
0011111111111111
0000111100001111
m_2
$(r_2 \vee (r_2 \gg 2)) \wedge m_2 = r_3$
0000111100001111
$r_3 \gg 4$
0000000011110000
$r_3 \vee (r_3 \gg 4)$
0000111111111111
0000000011111111
m_3
$(r_3 \vee (r_3 \gg 4)) \wedge m_3 = r_4$
0000000011111111

PLOT #1 (BYTE, BITS, HELMAN)



Optimization #2 (single pass)

- **Additional space (3^n bits)** for a merge bitmap
- **Additional time and cache misses** for second prime-marking traversal through **implicants, merge, prime** bitmaps
- **Improvement:** in the first pass, populate **prime** bitmap

```
merged = implicants[i] AND implicants[i + (1 << k)]

primes[i] = primes[i] AND NOT merged
```

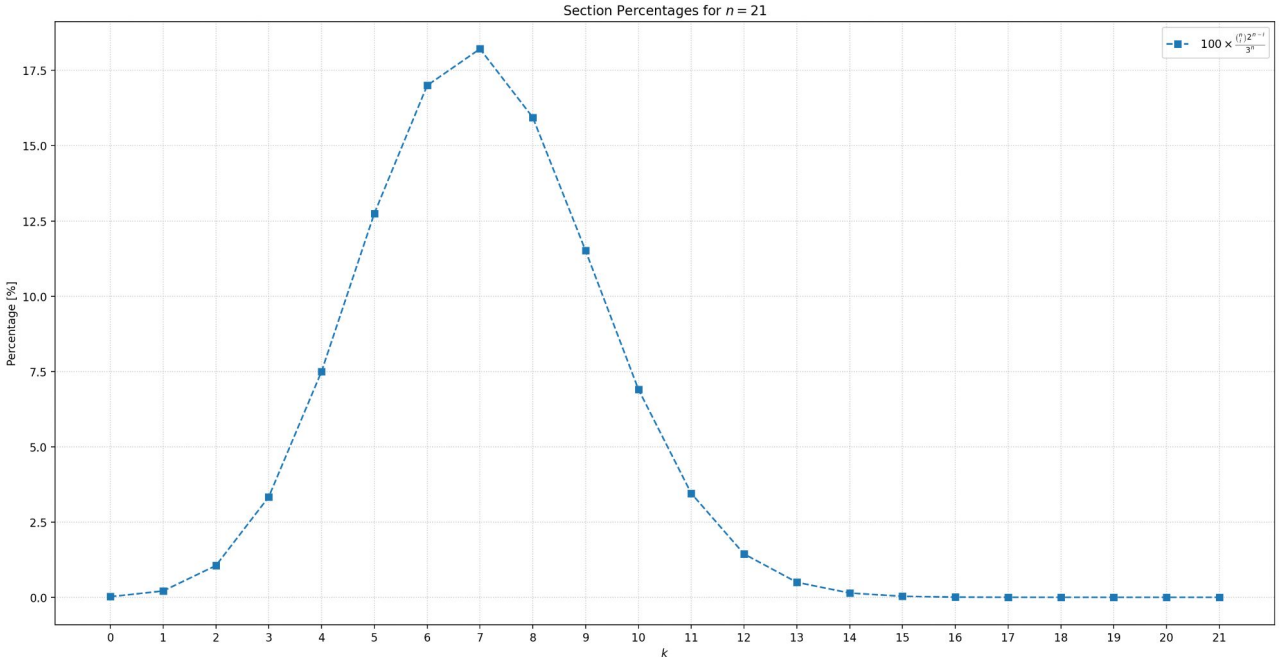
Cost Analysis

$$\#\text{Bit-Ops} = n3^n$$

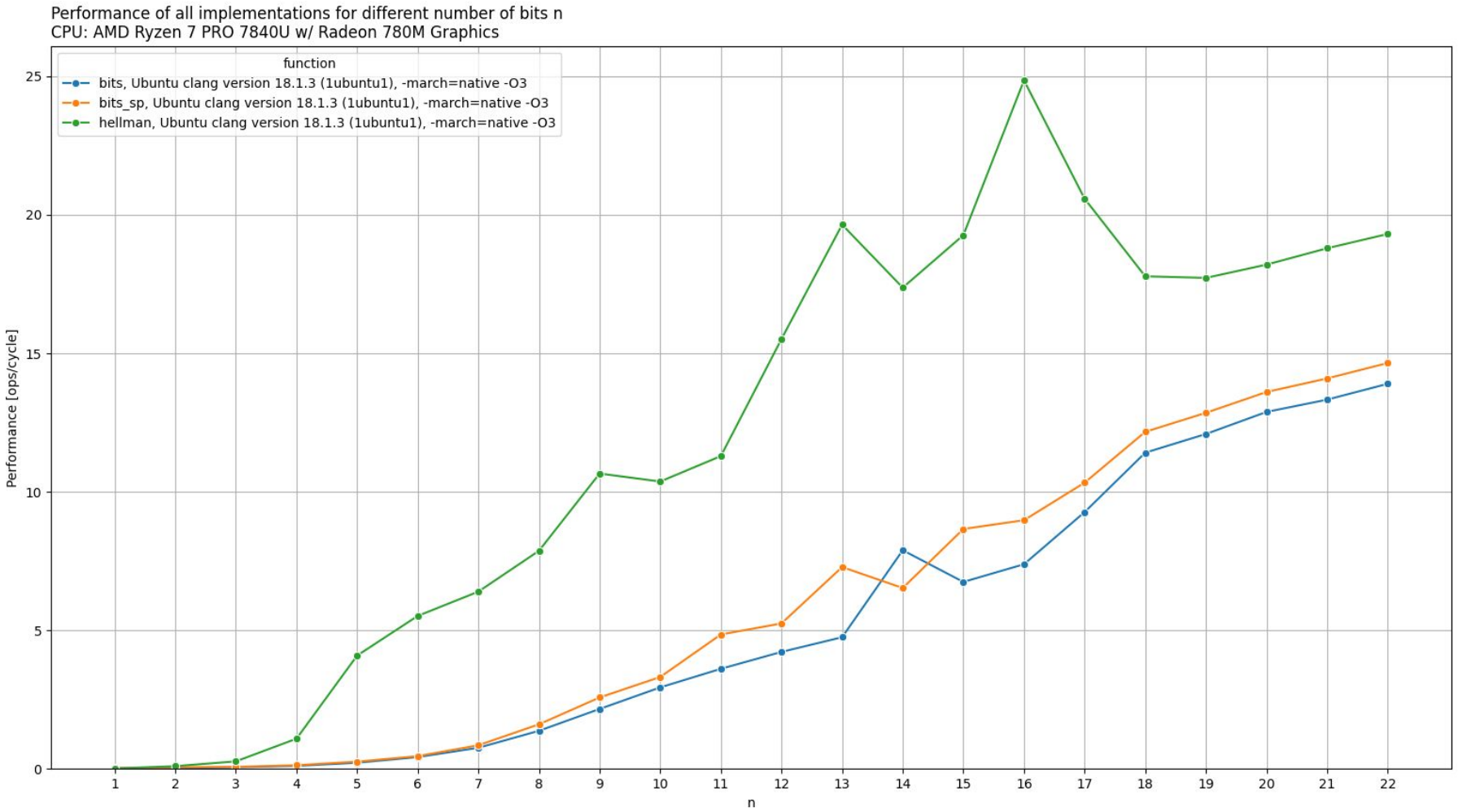
$$Q(n) \geq \frac{2 \cdot 3^n}{8} \text{bytes}$$

$$I(n) \leq n \cdot 4 \cdot 3^n$$

Biggest section for $k = \left\lfloor \frac{n+1}{3} \right\rfloor$



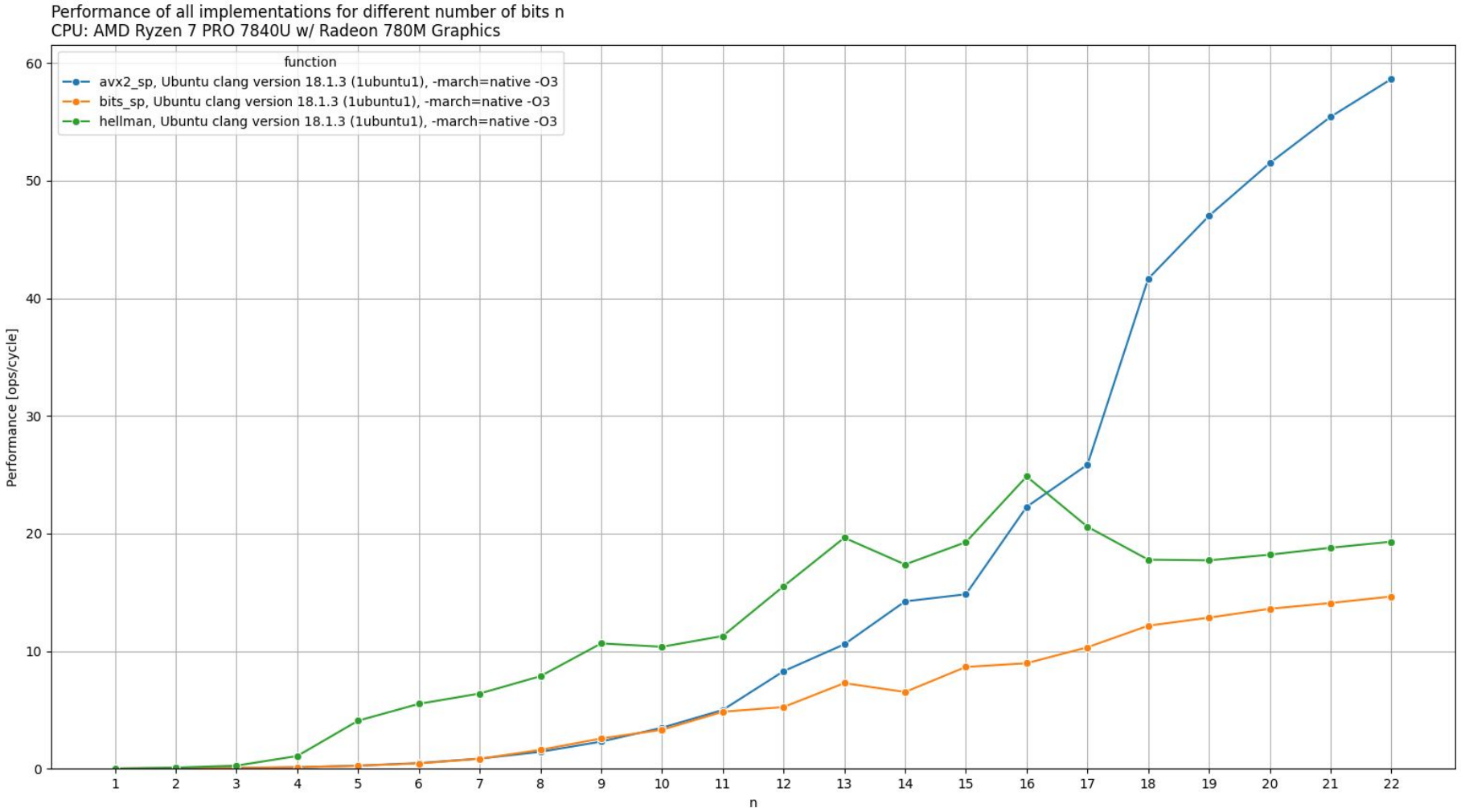
PLOT #2 (BITS, BITS_SP, HELLMAN)



Optimization #3 (AVX2 Vectorization)

- **Using 64-bit registers**
- **Improvement:** use 256-bit registers

PLOT #3 (bits_sp, avx2_sp, Hellman)



N=22 $\text{Speedup}_{\text{Helman}} = \mathbf{3.9}\text{x}$ $\text{Speedup}_{\text{bits\_sp}} = \mathbf{3.1}\text{x}$

Optimization #4 (SSA+ILP, Unroll)

- **Convoluted IF branches for inter-register merge**
  - No ILP
- **Improvement 1:** write straightforward code - add ILP

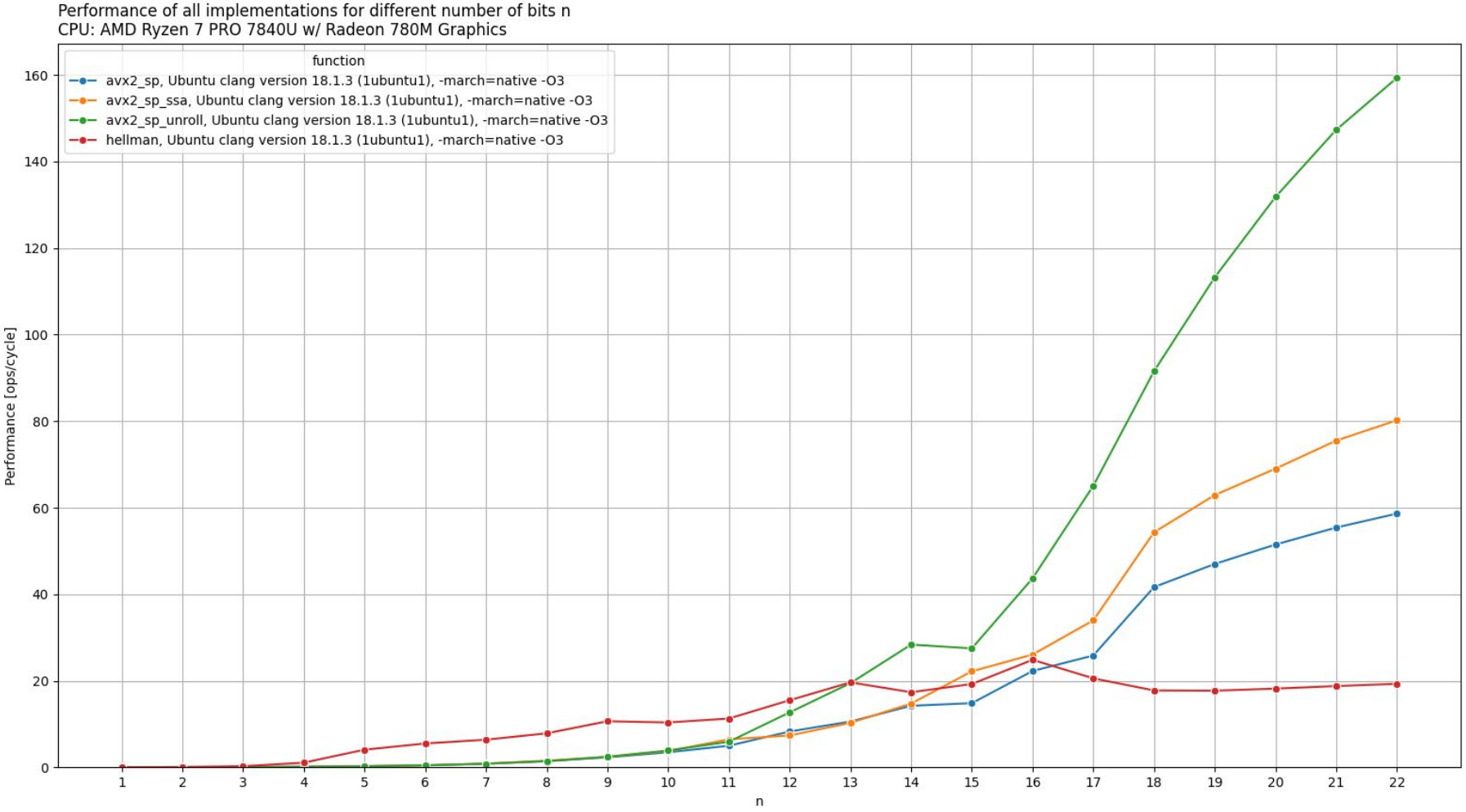
In **merge step** register will be **loaded multiple times** for each block length

$$\#\text{register\_loads} = k\frac{2^k}{\text{register\_size}} = k2^{k-8}$$

- **Suboptimal traversal order of inter-register merge in a chunk**
  - Bad temporal locality - multiple reads of same input bits
  - For large chunks can evict the whole cache
- **Improvement 2:** Inter-register merge processes all **possible block lengths** for a register before moving further + unroll loop for 4 inter-register steps

$$\#\text{register\_loads} = (k-7)2^{k-8}$$

PLOT #4 (avx2_sp, avx2_sp_ssa, avx2_sp_unroll, Hellman)

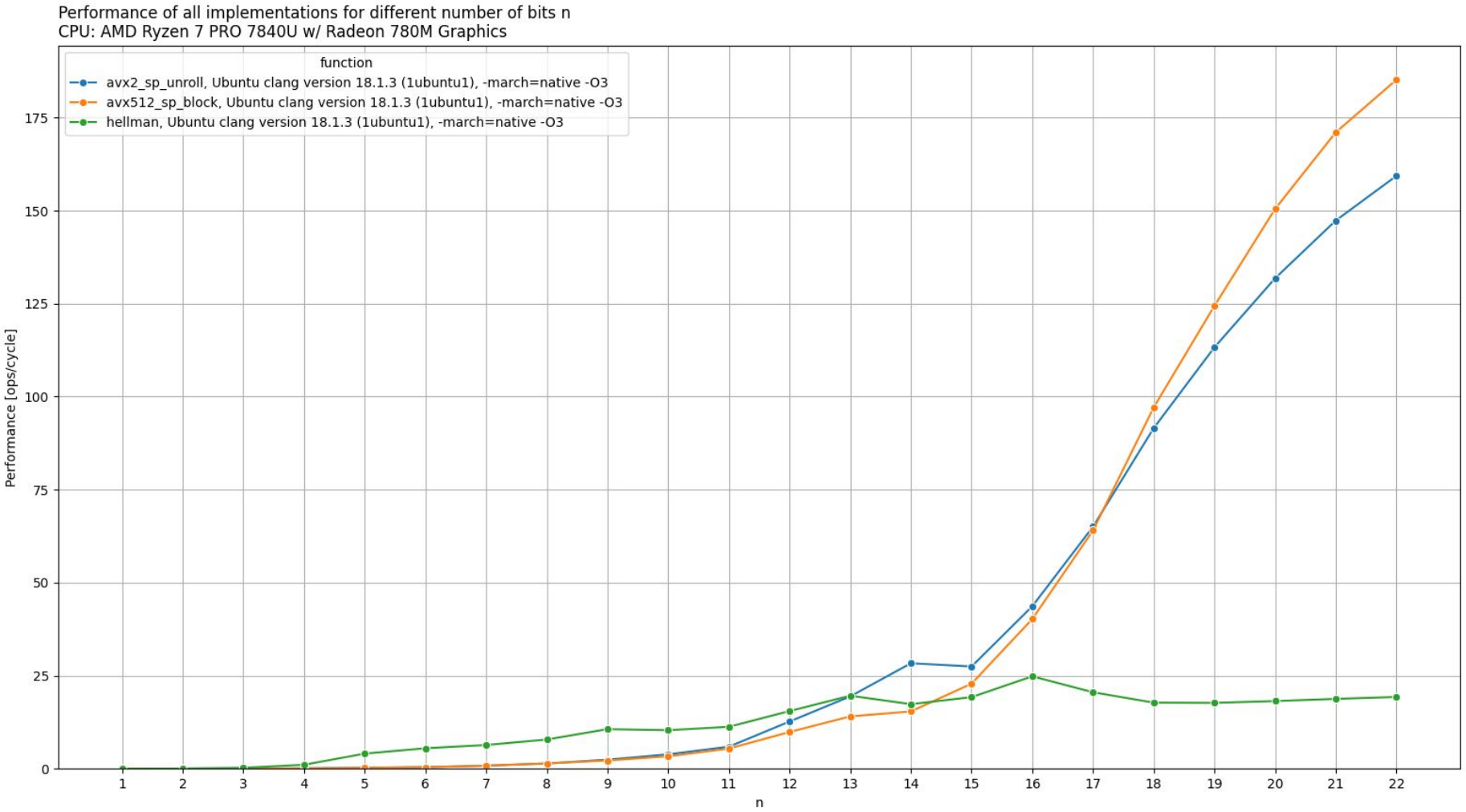


N=22 $\text{Speedup}_{\text{Hellman}} = \mathbf{8.0}\text{x}$ $\text{Speedup}_{\text{avx2\_sp}} = \mathbf{2.7}\text{x}$

Optimization #5 (AVX512)

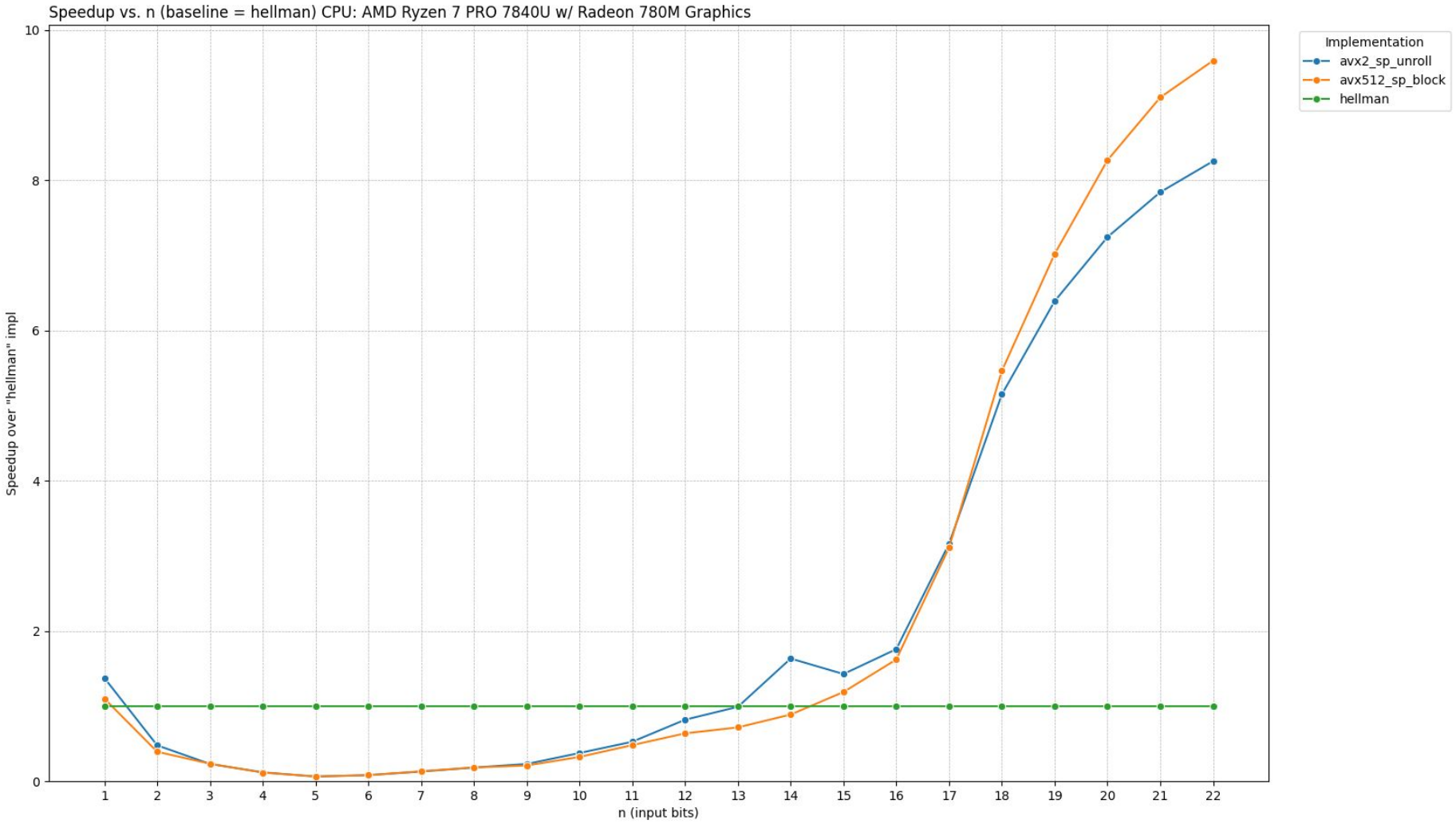
- **Using 256-bit registers**
- **Improvement:** use 512-bit registers

PLOT #5 (avx2_sp_unroll, avx512, Hellman)



N=22 $\text{Speedup}_{\text{Hellman}} = \mathbf{9.3}\text{x}$ $\text{Speedup}_{\text{avx2\_sp\_unroll}} = \mathbf{1.1}\text{x}$

SPEEDUP (avx2_sp_unroll & avx512 vs Hellman)

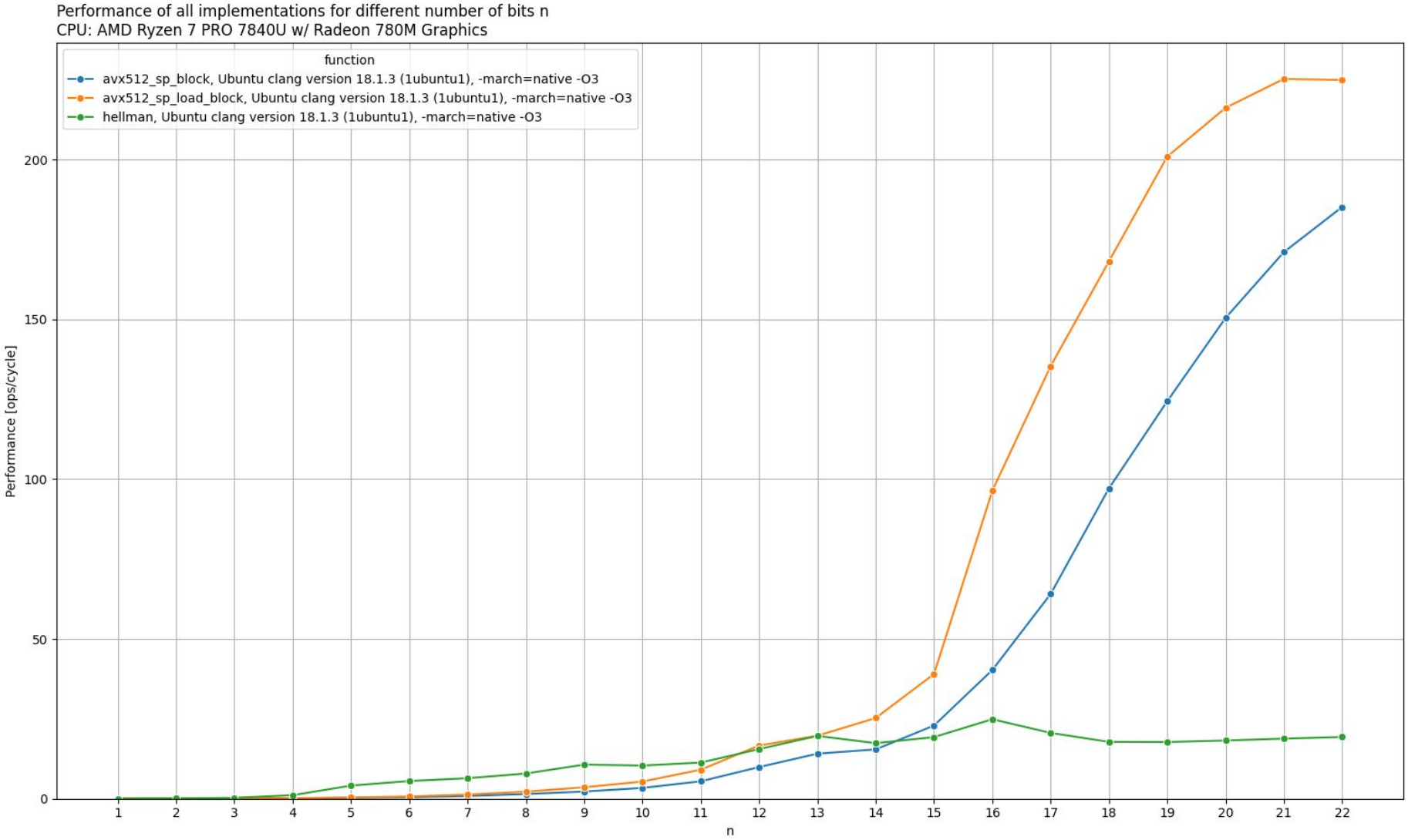


Final Optimization (loading traversal order)

- **The traversal order of implicants is being calculated, which adds cycles to the execution**
- **Improvement:** precompute the traversal order
- **Load additional 10 * 2^n bytes**
  - For n = 20, 21, 22 **negligible compared to bitmaps (< 1% of storage)**

```
// one linear pass
for (size_t i = 0; i < op_count; i++) {
    MergeOp *op = &ops[i];
    MERGE_FUNCTION(
        implicants,
        primes,
        op->in_idx,
        op->out_idx,
        op->rem_bits,
        op->first_diff
    );
}
```

FINAL PLOT #5 (avx512, avx512_load, Hellman)



SPEEDUP (avx512 & avx512_load vs Hellman)