PRIME IMPLICANTS FOR QUINE-MCCLUSKEY

Peter Nutter
Gerald Prendi
Grzegorz Swiader
Leyla Yaayladere

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We present a performant implementation for the prime implicant generation step of the Quine-McCluskey algorithm, a fundamental task in Boolean logic minimization. Our approach builds upon the dense bit-slicing implementation by Aleksei Udovenko [1] and introduces novel computation layouts we call CROSS-DIMENSIONAL and CROSS-LAYER. This significantly improves temporal data locality, mitigating the memory bandwidth bottleneck that constrains performance for large problem sizes. Combining the locality-aware memory access pattern with AVX vectorization and other loop unrolling, our implementation demonstrates a considerable performance increase. On our test platform, it outperforms the reference implementation, achieving a speedup of up to 2.06x for problem size $n = 23$. Our roofline analysis confirms that our method achieves higher operational intensity, making more effective use of the processor's computational resources.

1. INTRODUCTION

Motivation. The minimization of Boolean functions is a cornerstone of digital logic design, with direct applications in circuit synthesis, artificial intelligence, and other computational fields. The Quine-McCluskey (Q-M) algorithm is a tabular method that provides a deterministic way to find the minimal sum-of-products form of a Boolean function. The algorithm's complexity is exponential, so its runtime and memory footprint grow exponentially with the number of input variables n. This exponential growth makes the performance of its implementation a critical concern, especially when dealing with the large-scale problems common in modern applications. Consequently, developing high-performance implementations that efficiently handle large n is of significant practical importance.

Related Work. Several approaches have been proposed to accelerate the Q-M algorithm. One of these methods, which we use as our baseline, is the "dense" implementation by Aleksei Udovenko, referred to as "Hellman" in the rest of the paper [1]. Hellman's algorithm leverages bit-slicing, a technique that uses bit-parallelism to perform comparisons between many terms simultaneously, leading to a substantial speedup over naive implementations. However, as we will show, for large problem sizes, its performance becomes limited by memory bandwidth due to its memory access patterns, which exhibit poor data locality.

Contribution. In this paper, we present a series of optimizations that overcome the data movement limitations of existing high-performance Q-M implementations. Our main contribution is a novel computation strategy we term the CROSS-DIMENSIONAL & CROSS-LAYER approach. This method reorders the computation and merges multiple steps of the original algorithm to improve data locality. We detail our optimization journey, starting from the baseline.

2. BACKGROUND ON THE ALGORITHM

This section discusses the background of the algorithm, details of the reference implementation, and shows our cost measure.

General Quine-McCluskey Method. The algorithm begins with a truth table and iteratively finds and merges implicants that differ by exactly one bit. For example, two terms t_1 and t_2 merge if their bitwise XOR, $t_1 \oplus t_2$, is a power of two. The resulting term contains a "don't care" in the differing bit position. This process continues until no more merges are possible. The remaining implicants are called the prime implicants.

Baseline Algorithm. The implementation we compare against is the DENSE algorithm from hellman's paper [1]. The next few paragraphs explain the baseline's implementation details, also relevant for our versions of the algorithm.

Ternary Representation. Every implicant over n variables can be written as a word of length n over the alphabet $0, 1, -$, where '-' denotes "don't-care." Interpreting this word as a base-3 number - with digits $0 \to 0, 1 \to 1, - \to 2$

- maps the 3^n possible implicants to a range of indices. To denote presence of implicants, bits in a large bit-vector S are set to 1.

Bit-Slicing and Data Partitioning. To manage this bit vector and enable parallel processing, hellman combines data partitioning with a technique called *bit-slicing*. First, he partitions the n variables into a *bottom layer* of $h = 5$ variables and a *top layer* of the remaining $n_h = n - h$. The choice of $h = 5$ is deliberate: 3^5 is the power of three closest to a power of two - 2^8. This enables processing all $3^5 = 243$ bits in parallel by bitwise logical instructions.

Block Indexing. These 243-bit blocks are stored contiguously in the main array S. The specific values of the top-layer variables determine the block's index. As a result, processing a dimension d in the top layer requires stepping through S with a stride of 3^d. In these top-layer phases, blocks are always accessed in triples $\langle s, t, u \rangle$ with indices $\langle b, b + stride, b + 2 \cdot stride \rangle$.

Algorithmic Phases. Hellman's algorithm and implementation has two logical phases in two logical layers. First, hellman iterates through the n_h top-layer dimensions, processing triples of blocks in the (1) **top-layer merge**. Next, within each of the 3^{n_h} blocks, for the h bottom-layer dimensions, he processes first the (2) **bottom-layer merge** and then the (3) **bottom-layer reduce**. Lastly, he iterates through the n_h top-layer dimensions again, processing the (4) **top-layer reduce** phase.

The merge operation on a triple is defined by

$$(\chi_0, \chi_1, \chi_*) \mapsto (\chi_0, \chi_1, \chi_* \vee (\chi_0 \wedge \chi_1)),$$

while the reduce is given by

$$(\chi_0, \chi_1, \chi_*) \mapsto (\chi_0 \wedge \neg\chi_*,\ \chi_1 \wedge \neg\chi_*,\ \chi_*).$$

Cost Measure and Analysis. We define a cost model based on the number of bitwise operations performed during the **merge** and **reduce** phases, counting only the core binary operations: AND, OR, and fused AND–NOT. Bitwise shifts and control logic are excluded from the model. The merge step is counted as one `AND` and one `OR`, while the reduce step is counted as two `AND-NOT` operations. The total cost measure is $C(n) = C_\wedge \cdot N_\wedge + C_\vee \cdot N_\vee + C_{\wedge\neg} \cdot N_{\wedge\neg}$, with operation counts determined by the algorithm's deterministic passes through top and bottom layers regardless of input, since it checks all minterms.

$$N_{\text{top}} = \underbrace{3^{n-6}}_{\text{Iterates blocks by 3}} \cdot \underbrace{(n-5)}_{\text{top dimensions}} \cdot \underbrace{256}_{\text{bits in a block}}$$

$$N_{\text{bot}} = \underbrace{3^{n-5}}_{\text{Iterates all blocks}} \cdot \underbrace{5}_{\text{bottom dimensions}} \cdot \underbrace{256}_{\text{bits in a block}}$$

$$N_\wedge = N_{\text{top}} + N_{\text{bot}}$$

$$N_\vee = N_{\text{top}} + N_{\text{bot}}$$

$$N_{\wedge\neg} = 2 \cdot (N_{\text{top}} + N_{\text{bot}})$$

3. OPTIMIZATION PERFORMED

In this section, we describe the optimizations we performed over the baseline implementation. These include amongst other things, reducing the operation count, two kinds of temporal locality improvements and using AVX instructions in an explicit load-compute-store SSA style to achieve better ILP. Where appropriate, we visualize the optimizations on graphs and give formulas quantifying the improvements. These optimizations are parameterizable, lending themselves to code generation.

The baseline exhibits poor locality. Figure 1 shows that the reference implementation exhibits no temporal data locality. It makes $(n-5)$, 1, and $(n-5)$ strides across the entire table in the top layer merge, bottom layer, and top layer reduce respectively, for a total of $2n - 9$ full strides. The size of the table is exponential in n ($3^{n-5} \cdot 2^5$ bytes), so for sufficiently large n the table no longer fits in cache, making the computation memory-bound.

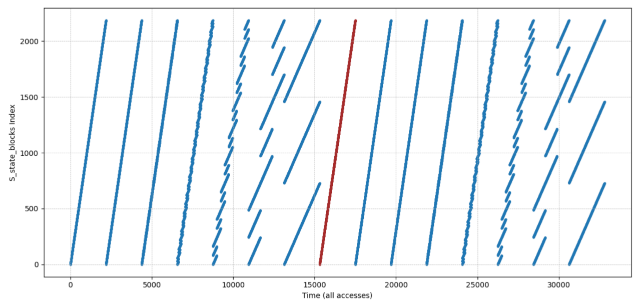


Fig. 1: Access pattern for hellman's DENSE for $n = 12$. The red line denotes accsses in the bottom layer.

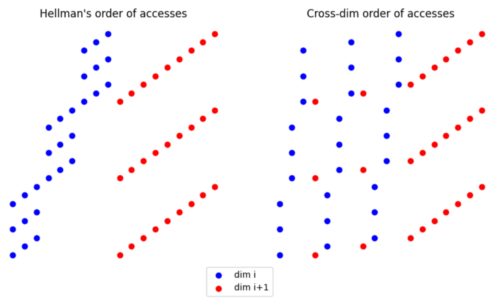


Fig. 2: The idea behind CROSS-DIM access pattern. We process triples from dimension $i + 1$ as soon as all relevant triples from dimension i have been processed.

Alternative approach: Tree algorithm. Our initial attempt to address the locality problem involved a complete restructuring of the data layout. We designed a "tree algorithm" where implicants were grouped into bins based on their number of "don't care" symbols and the positions of those symbols. This approach seemed elegant, as merges would only occur between specific, well-defined bins, potentially improving locality. However, the implementation proved conceptually complex, and the irregular data structures made efficient vectorization difficult. We were unable to outperform the baseline and therefore returned to optimizing the dense, bit-slicing approach.

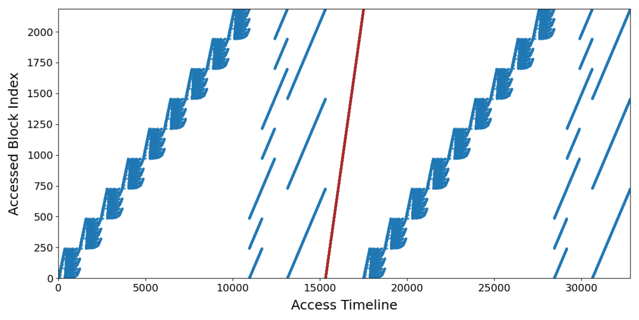


Fig. 3: Access pattern for CROSS-DIM for $n = 12, t = 5$.

Cross-dimensional locality. We apply a blocking optimization for the top layer computations to split the whole table into smaller working sets that fit into cache. The core idea is vizualised for working across 2 dimensions in Figure 2, but the same pattern can be applied across any number of dimensions t at a time. Figure 3 shows the memory access pattern for $t = 5$, the value we found best for this parameter. In this section we explain the case where $t = 2$ for simplicity. Listings 1 and 2 are the snippets of code corresponding to memory access patterns in Figures 1 and 2.

```
// go over full table
for (size_t b = 0; b < n_blocks; b += step) {
    // process 1 dimension only
    for (size_t c = 0; c < shift; c++) {
        TOP_MERGE(S, b + c, shift);
    }
}
```

Listing 1: Baseline: One dimension per stride

There are two interesting ideas in Listing 2:

1. The ordering of the loops: we go step by step through dimension $d + 1$ (b += step1), and within it, we do a 1-stride (c++), processing dimension d when needed (c $<$ shift0). This loop order is good because it minimises the size of the working set. Within an iteration on c, we access 3^2 (or 3^t in the general case) blocks.

2. Loop specialization. Instead of using an if statement to check c $<$ shift0, we split the loop into two specialized ones: one where $c \in [0, shift0)$, and another where $c \in [shift0, shift1)$ to get better code locality and less branching.

This optimization reduces the number of memory strides roughly by almost a factor of t. A more rigorous analysis is given at the end of the ”smoothing” paragraph.

```
// go over full table
for (size_t b = 0; b < n_blocks; b += step1) {
    size_t c = 0;
    // process 2 dimensions together
    for (; c < shift0; c++) {
        size_t base1 = b + c;
        int    i;
        for (i = 0; i < TOP_MERGE_ITER; i++) {
            size_t base0 = base1 + i * step0;
            TOP_MERGE(S, base0, shift0);
        }
        TOP_MERGE(S, base1, shift1);
    }
    // process the remaining elements
    for (; c < shift1; c++) {
        size_t base1 = b + c;
        TOP_MERGE(S, base1, shift1);
    }
}
```

Listing 2: Cross-dim: Two dimensions per stride

Reducing the operations count. Listing 2 makes use of a constant TOP_MERGE_ITER. From Figure 2 we can see that to process a triple from dimension $d+1$, we need to process 3 triples from dimension d. By a happy accident, we first coded the top-layer merge loop to range over $i \in [0, 2)$ instead of $i \in [0, 3)$, but the code still passed the tests for correctness. This reduction works only for the merge phase of the algorithm, but not for reduce. Figure 4 shows the access patterns for the merge phase with skipped computation (on the left of the red lines), and for the reduce phase with full computation (to the right of the red lines).

We don’t have a proof of correctness, but test-based evidence supports the claim that this reduction preserves correctness of the algorithm. It also proved difficult for us to analyze the number of operations performed by this new version of the algorithm, so we don’t have a formal cost analysis. Instead, we instrumented the code to count the number of operations at runtime.

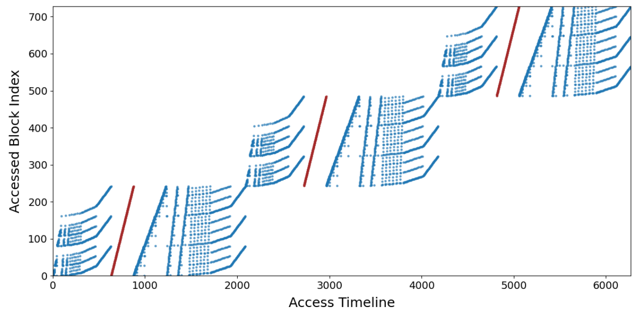


Fig. 4: Access pattern with skipping unnecessary merges (to the left of the red lines) and without skipping computation for reduces (to the right of the red lines).

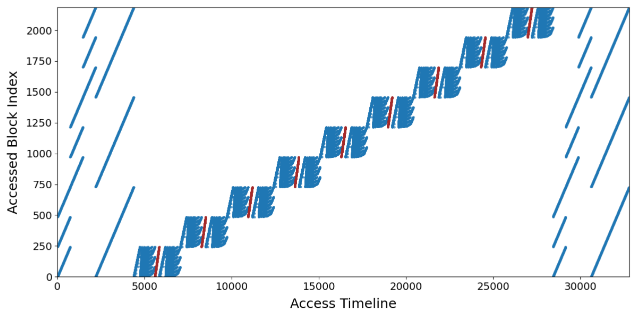


Fig. 5: Access pattern for CROSS-DIM-CROSS-LAYER for $n = 12$, $t = 5$, $b = 5$, without skipping merges.

Cross-dimensional smoothing. Fixing t to work across t top-layer dimensions at a time, we have $(n-5) \bmod t$ dimensions left over. To finish processing them, we can fall back on the one-by-one stride of the reference implementation. A slightly more sophisticated solution is to smooth this part out, writing (generating) code for work across t, $t-1$, …, 1 dimensions at a time. When working across t dimensions at a time with smoothing, the number of full strides in a top layer becomes $\lceil \frac{n-5}{t} \rceil$, for $2 \lceil \frac{n-5}{t} \rceil + 1$ memory strides in the full computation. For example, for $n = 23$ and $t = 5$, we improve from $2 \cdot (23-5) + 1 = 37$ to $2 \lceil \frac{23-5}{5} \rceil + 1 = 9$ full memory strides.

Bottom-layer ILP. In the bottom layer, the most expensive operations are the bitshifts. There is no AVX instruction that treats the entire 256-bit register as one value and shifts it by an immediate number of bits. Thus, to hand-roll an AVX implementation of bitshifts, we need to write a sequence of permutations and packed 64-bit integer ands, shifts, and ors. A single bitshift then has quite a large latency. Interleaving these bitshifts for multiple independent blocks can give a good speedup by exploiting ILP.

Cross-layer locality. Figure 5 shows another idea: we can move some of the dimensions in the top layer to be computed in the same memory stride as the bottom layer. For simplicity of implementation, we move the lowest b dimensions from the top layer to be processed alongside the bottom layer. This also has the benefit of having the blocks consecutive in memory - preparing for the optimization described in the previous paragraph. This cross-layer optimization improves slightly on the number of full memory strides: given n, t, b the number of strides will be $\lceil \frac{n-5-b}{t} \rceil$ in the top layer, and 1 in the bottom layer, for a total of $2 \lceil \frac{n-5-b}{t} \rceil + 1$ full memory strides. When $b = t$, it removes an additional 2 strides (1 per top layer).

Code generation. The optimizations listed above are all parameterizable. We wrote a python script to generate the code for different parameters. For practical reasons we limited ourselves to measuring code for parameters under 8.

4. EXPERIMENTAL RESULTS

This section presents the empirical evaluation of our optimizations. We first explain our experimental setup and methodology. We then present and analyze the performance results, comparing our optimized implementations against the baseline. We report speedups given by each individual optimization. Finally, we use cache-aware metrics and a roofline model to provide deeper insight into the source of the performance gains.

Experimental Setup. All experiments were conducted on an **AMD Ryzen 7 4800H (Renoir)** processor. This processor is a Zen2 microarchitecture and has a base clock speed of 2.9GHz. The Ryzen 7 4800H processor has a cache hierarchy of:

L1d	32KiB, 8-way associative, 64B line size, per-core,
L2	512KiB, 8-way associative, 64B line size, per-core,
L3	8MiB, 16-way associative, 64B line size, shared,

which differs from the Renoir specification in the 8MiB (instead of 4MiB) L3 cache. Zen2 processors have a Floating Point execution unit with 4 ports that can perform AVX instructions. Given that we're not using FMA instructions and we count bit operations, the peak performance of a Zen2 processor is 1024 bitops per cycle.

The code was compiled using **GCC 14.3.1** with the flags **-O3 -march=native -mavx2 -std=c++17**. To ensure accurate and stable measurements, we used the rdtsc instruction to count CPU cycles and a high_resolution_clock from the standard library to measure the seconds independently.

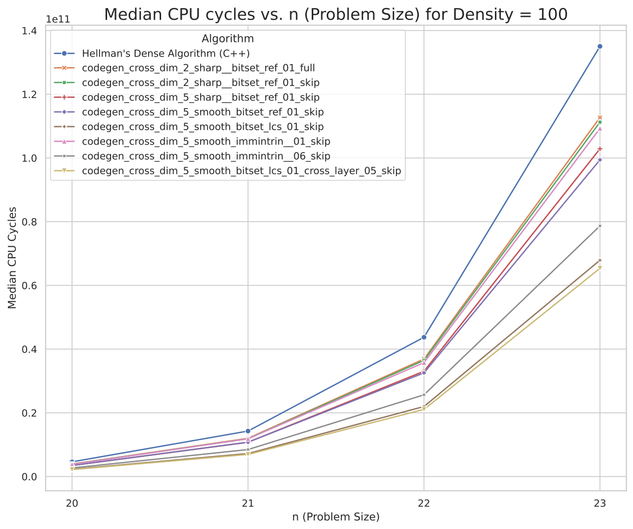


Fig. 6: Best versions of our code after each optimization.

Each data point represents the median of 100 runs, preceded by several warmup runs. We evaluated performance across a range of problem sizes from $n = 12$ to $n = 23$ and for various function densities, though we primarily present results for density 100 and $20 \leq n \leq 23$.

Results. Figure 6 outlines the runtimes of the best versions of our code after each optimization step. In short: we first reduce the operations count with the MERGE-SKIP optimization, we search for the best parameter t for CROSS-DIM, then apply top-layer SMOOTHING. For the bottom layer, we compare different versions of unrolling. Finally, we search for a good parameter b for the CROSS-LAYER optimization. The best parameters turn out to be $t = b = 5$, and using C++ bitsets with no manual unrolling in the bottom layer. The best speedup we obtain for $n = 23$ is 2.066.

Merge-skip. We apply this optimization first because it changes the number of operations from the reference implementation. Working over $t = 2$ dimensions at a time in the top layer, we get a 1.2137 speedup over the baseline, as shown in figure 7. Note that this speedup includes both the change from hellman to CROSS-DIM 2 and the MERGE-SKIP. The more significant gain in this speedup comes from the CROSS-DIM optimization because of the reference implementation's bad memory access pattern.

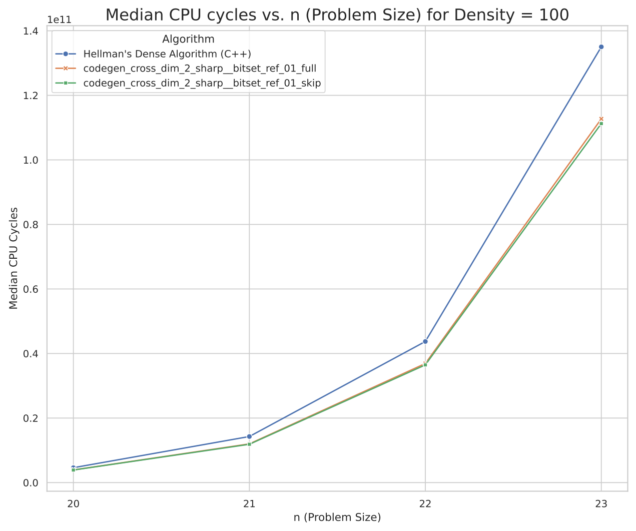


Fig. 7: Runtime comparison of the reference implementation against the CROSS-DIM (data movement) and MERGE-SKIP (compute) optimization ideas.

Cross-dim parameters. Given t, the working set for this optimization is 3^t bitset blocks. Increasing t causes the algorithm to make fewer full memory strides, but with an increased working set size. Given that the L1d cache size is 32KiB and the cache line size is 64B, the maximum t that still has the working set fully in cache is $t = 5$ for just over 15KiB. However, the MERGE-SKIP optimization reduces the size of the working set for the merge phase. Experimental results showed us that for $t = 6$, the extra L1d cache misses from the reduce phase are offset by the reduction in the number of memory strides. However, we saw this only when CROSS-DIM was the only optimization applied. Combining CROSS-DIM with later optimizations showed that $t = 5$ is still the better parameter over $t = 6$. The speedup we get for $t = 5$ over the previous best – $t = 2$ is 1.08.

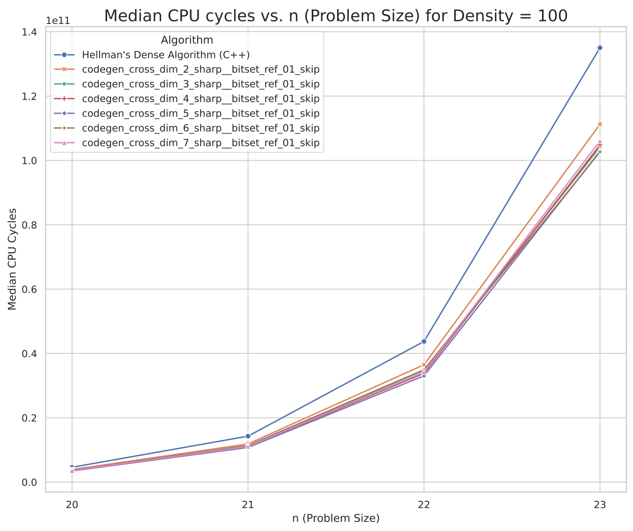


Fig. 8: Runtime comparison for different parameters for the CROSS-DIM optimization. The best value for this parameter is $t = 5$.

Cache and Locality Analysis. To further analyze the source of our performance gain, we plot Operations per Cycle (Ops/Cycle) versus input size in Figure 9. The vertical lines indicate the approximate problem sizes where the data footprint of the algorithm exceeds the L1, L2, and L3 caches of our test machine. The performance of the baseline algorithm drops sharply as the working set spills out of the caches. In contrast, our CROSS-DIMENSIONAL approach maintains a significantly higher Ops/Cycle rate, demonstrating its superior cache utilization. The performance of all implementations drops after the L3 cache is exceeded, but our optimized versions maintain a clear advantage, confirming they are less affected by main memory latency.

Smoothing. It is difficult to determine the exact speedup this optimization gives. When $n - 5 - b$ is a multiple of t, there are no extra strides that SMOOTHING could remove. But when $n - 5 - b \equiv t - 1 \pmod{t}$, it can remove $t - 2$ strides from each top layer. For $n = 20$, $t = 5$, $b = 0$ we obtain no speedup, but for $n = 23$, $t = 5$, $b = 0$, the speedup was 1.035. For a different choice of parameters – $n = 22$, $t = 6$, $b = 0$ ("best case") – we achieved a slightly better speedup of 1.05.

Unrolling. We found that 'hellman' was doing something seemingly harmless in the bottom layer: iterating over

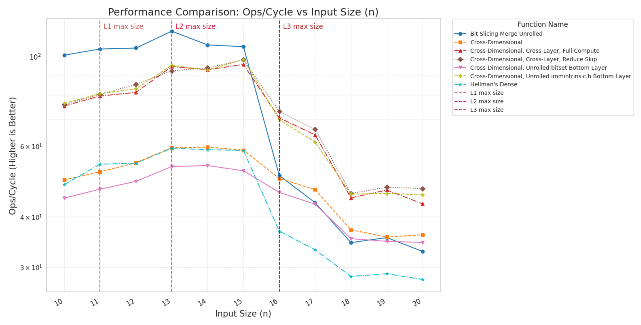


Fig. 9: Operations per Cycle vs. Input Size (n). The performance of our locality-aware methods degrades less severely as the data size exceeds the machine's cache capacities.

the blocks by reference. This causes C++ to reflect each individual change in memory, making the computation reach out to memory 10 times per block. Changing this loop to an index-based, explicit load-compute-store style lets the compiler apply full unrolling and ILP optimizations. Our hand-rolled implementations outperform hellman's bitset implementation, but an explicit load-compute-store use of the C++ bitset still outperforms our version as seen in figure 10. It is interesting that manual unrolling for small unroll factors prevents the compiler from optimizing further, as shown by the degradation in performance.

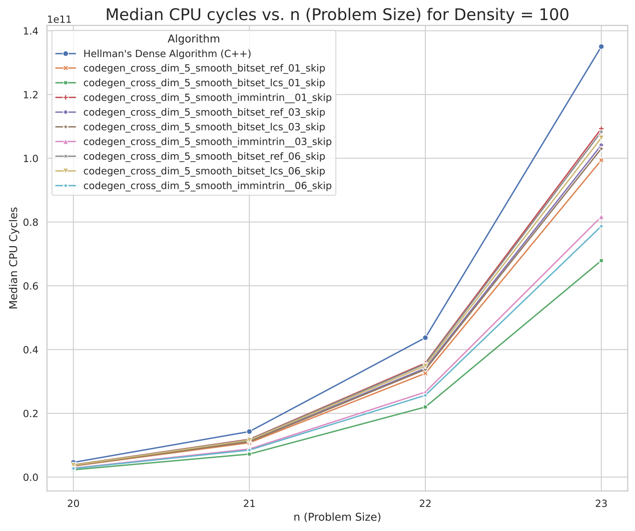


Fig. 10: Different unrolling styles and factors for the bottom-layer computation. Our hand-rolled AVX code is slightly outperformed by code generated from C++ bitsets.

Cross-layer parameters. As figure 11 shows, we found that the best parameter b is 5. We are not quite sure why it's not 6. While the working set is 3^b bitset blocks, all of them are consecutive, fitting 2 bitset blocks per cache line. For $b = 6$, it would be just under 23KiB. We are at least sure that for $b = 7$, the size of the bottom-layer working set is over 68KiB, causing a number of L1 cache misses and degrading performance.

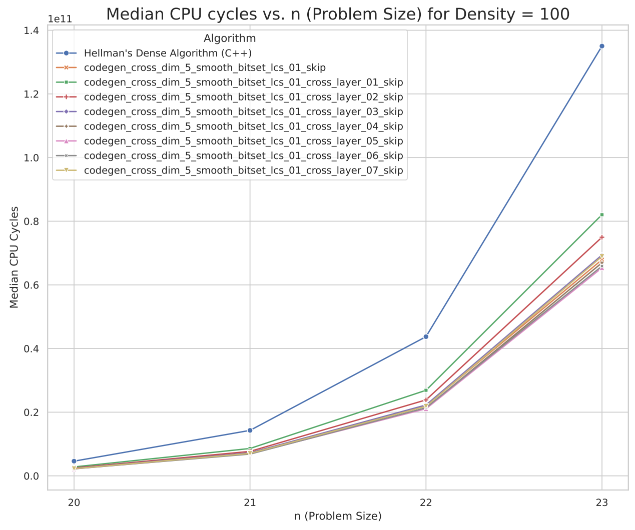


Fig. 11: Results for investigating the best parameter b for the CROSS-LAYER optimization. For $b = 7$, the performance degrades compared to $b = 5$ due to L1 capacity misses.

Roofline Analysis. Figure 12 presents a roofline model analysis of our best implementations against the baseline for $n = 16$ and $n = 19$. This model plots achieved performance (ops/cycle) against operational intensity (ops/byte) for versions of our algorithm without the MERGE-SKIP optimization. The plot clearly shows that our CROSS-DIM and CROSS-LAYER implementations are shifted to the right, so they achieve higher operational intensity than the baseline. This means our methods perform more arithmetic operations for each byte of data transferred from memory. By improving data reuse, our optimizations successfully move the computation from being purely memory-bound to the compute-bound roof of the machine, leading to more efficient use of the available hardware resources.

Reducing the Operations Count. As mentioned in Section 3, we discovered an optimization that reduces the number of merge operations in each top-layer dimension. Because this fundamentally alters the algorithm's operational count, a direct comparison on the same performance/roofline plots would be misleading. Therefore, we first evaluate it separately by measuring its runtime improvement over our best locality-optimized version, CROSS-LAYER. This optimization provided a further speedup of approximately **1.21** for $n = 23$ over the baseline. A visualization of the skipped computation paths can be seen in the memory access pattern in Figure 13.

To assess the hardware efficiency of this new algorithm on its own merits, we present its roofline analysis in Fig-

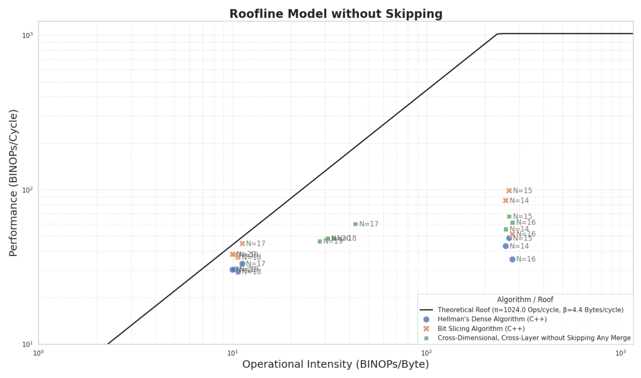


Fig. 12: Roofline model for n=14 to n=20. Our optimized algorithms achieve higher operational intensity, indicating better memory system utilization.

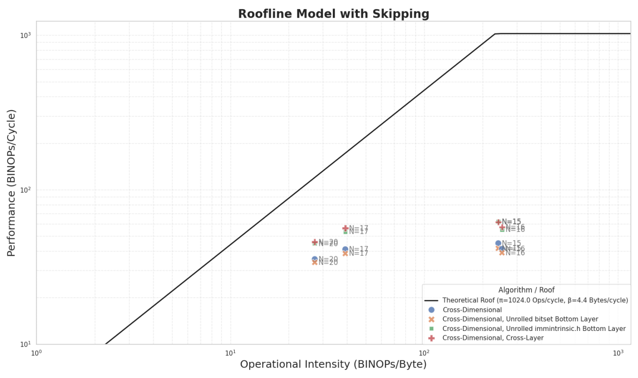
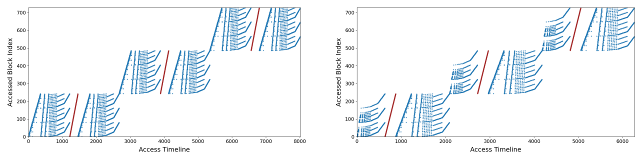


Fig. 14: Roofline model for the CROSS-LAYER-SKIP algorithm from n=14 to n=20. Some n ommited to make the plot more readable. The plot characterizes the hardware utilization of the algorithm with the reduced operation count.

ure 14. The plot shows that for smaller problem sizes like $n = 15-16$, the CROSS-LAYER-SKIP algorithm operates in the compute-bound region. As the problem size increases to $n = 19$, the operational intensity decreases slightly, and the performance moves closer to the memory bandwidth ceiling, indicating that even with a reduced workload, the algorithm becomes memory-bound as the data set exceeds the cache capacity.



(a) Without skip optimization (b) With skip optimization

Fig. 13: Zoomed-in memory access patterns in the cross-layer part of the algorithm for input size $n = 12$.

5. CONCLUSION

In this paper, we addressed the primary performance bottleneck in the dense bit-slicing implementation of hellman's Quine-McCluskey algorithm: poor data locality in the top-layer processing phases. Our main contribution is a data localization technique, the CROSS-DIMENSIONAL approach, which reorders the computation to maximize cache reuse. By processing multiple dimensions at a time on a small, contiguous block of data before it is evicted from the cache, our method dramatically reduces the number of costly main memory accesses.

Our final implementation demonstrates a clear and significant performance improvement over the Hellman baseline, achieving a speedup of up to 2x ($n = 23$).

Future Work. There are some optimizations that we haven't tried in the time limit of the course. The ones that we considered but haven't worked towards are:

- blocking for CROSS-DIM targetting L3 cache,
- blocking for CROSS-LAYER targetting L3 cache,
- changing the storage layout,
- skipping more computation in the merge phase.

6. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Peter. After setting up the benchmarking, plotting, and example generation infrastructure, I implemented the initial dense version (DENSE ALGORITHM) without bitslicing, then vectorized it (DENSE-VECTORIZED). Developed the bitslicing version and attempted to port it to AVX (BIT SLICING AVX) encountered challenges with AVX bitshifts, so collaborated with Grzegorz to complete it. Assisted Leyla in getting the tree-algorithm operational. Introduced macros, and precomputed power tables for further code optimization. Later, focused on the roofline plot and simulated cache hit/miss behavior for roofline analysis.

Leyla. Implemented TREE-BASED dense algorithm to exploit locality and skip some merge operations; performance was limited due to complex index calculations and poor vectorization. Ported DENSE-VECTORIZED algorithm to AVX-512, but no gains due to memory bound. Analyzed memory patterns and proposed BIT-SLICING-SNAKE (bidirectional pass) to exploit edge locality which had limited impact. With Grzegorz, proposed combining top and bottom layer passes (CROSS-DIM); assisted him in extending this to blocking with proof-checking, debugging, and analyzing memory access patterns of his implementations. Counted operations for cost and performance analysis.

Grzegorz. Came up with the CROSS-DIM and CROSS-LAYER ideas. Found MERGE-SKIP by complete accident. Together with Leyla, implemented these versions of the algorithms. With help from Gerald, wrote a codegen script to search for the best parameters for each of the optimizations. Hand-rolled the AVX 256-bit register bitshifts, which Gerald then specialized for particular shifts. Tried manual bottom-layer unrolling. Found that compilers unroll better and at less pain.

Gerald. Worked on unrolling the BitSlicing implementation's bottom layers and merging the two layers. Introduced macros for unrolling, and function call removal. Optimized the custom bitshifts implementation for different shift-sizes, and optimized the vectorized bottom layers to reduce the number of operations. Did some occasional profiling with Intel Advisor, AMD uProf and worked the perfomance plot with caches. Helped Peter and Leyla with the rooflines. Assisted Grzegorz with brainstorming and debugging the codegen script.

7. REFERENCES

[1] Aleksei Udovenko, "DenseQMC: an efficient bit-slice implementation of the quine-McCluskey algorithm," Cryptology ePrint Archive, Paper 2023/201, 2023.