OPTIMIZING PERFORMANCE OF THE QUINE-MCCLUSKEY ALGORITHM USING A NEW MEMORY LAYOUT

Aljaž Medič, Aleksa Mićanović, Aleks Stepančič, Richard Wohlbold

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

In this paper, we present performance optimizations for the Quine-McCluskey algorithm. We introduce a memory layout that groups related implicants to improve spatial locality. Based on this, we apply a series of algorithmic and low-level optimizations, which we evaluate through a set of experiments. Our non-vectorized implementation outperforms a vectorized reference implementation by 3.2x. We also report speedups of 11.7x, 12.4x, and 26.2x for hand-vectorized implementations on AVX2, AVX-512, and ARM NEON, respectively. We analyze the results and outline directions for further optimization.

1. INTRODUCTION

Motivation. In chip design, Boolean functions are used to specify the inputs and outputs of a given circuit, which can be implemented in many different ways. Circuits with fewer logic gates are generally preferable because they require less space and energy. Thus, one generally tries to find an efficient representation of the Boolean function with as few logic operators as possible. This process is called *Boolean function minimization* [1].

The Quine-McCluskey algorithm is a canonical algorithm for this problem. It computes all prime implicants and finds a globally minimal representation of a Boolean function [1]. Its exponential complexity makes it impractical for large-scale circuit design. However, the algorithm remains widely used in educational contexts, formal verification tools, and FPGA-based prototyping, where exact minimization is achievable and needed.

This algorithm consists of two parts: (1) finding all prime implicants and (2) finding a minimum set of prime implicants such that their logical OR implies the function. In this report, we focus on the first part, finding all the prime implicants. Our implementation considers the *dense* case, which means that the truth values of all implicants are stored.

The first part of the algorithm is exponential in the number of bits of the function n, requiring 3^n bits of space and $n3^n$ bit-operations of work. To find prime implicants, the algorithm repeatedly compares truth values of implicants whose bit-representations only differ in one digit to determine the truth value of an output implicant. Vectorization is able to leverage data parallelism in these comparisons. Additionally, depending on the way implicants are ordered in memory and the order of computations, locality and therefore cache-efficiency can differ among implementations.

Related Work. The best existing dense implementation that focuses on single-core performance we could find is DenseQMC [2]. Its memory layout represents implicants using a ternary number system, using many small bitsets, where implicants that only differ within the last 5 digits are stored within the same bitset. However, this approach leads to $n-4$ passes over the entire working set, thrashing the caches in the process. We call this implementation *hellman*, corresponding to the author's GitHub username.

Contribution. In this paper, we introduce an alternative memory layout to find prime implicants within the Quine-McCluskey algorithm. We introduce several performance optimizations for this approach, such as improving bit extraction, using a single-pass algorithm, improving the intra-register merge order, blocking for inter-register merging, pre-loading merge orders as well as vectorizing by hand for the AVX2, AVX-512 and NEON instruction sets. Our best optimizations are faster than the referenced existing implementation for bit-lengths $n > 10$. For large implicant sizes (e.g. $n = 22$) we obtain speedups of 3.2x, 11.7x, 12.4x and 26.2x when using compiler auto-vectorization, and with hand-vectorized implementations for AVX2, AVX-512 and NEON, respectively.

2. ALGORITHM

In this section, we summarize the algorithm, explain our memory layout, introduce our cost measure and perform cost analysis on the algorithm.

Definitions. Let $f : \{0,1\}^n \to \{0,1\}$ be an n-bit Boolean function. A *minterm* of f is an input $x \in \{0,1\}^n$ such that $f(x) = 1$. We commonly represent minterms as binary numbers. For example, for $n = 4$, we represent the minterm 1101 as 13.

An *implicant* of f is a formula $x \in \{0,1,-\}^n$ such

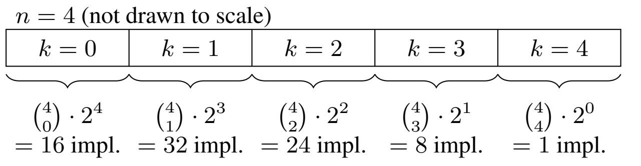
$n = 4$ (not drawn to scale)
$k = 0$
$k = 1$
$k = 2$
$k = 3$
$k = 4$
$\binom{4}{0} \cdot 2^4 = 16$ impl.
$\binom{4}{1} \cdot 2^3 = 32$ impl.
$\binom{4}{2} \cdot 2^2 = 24$ impl.
$\binom{4}{3} \cdot 2^1 = 8$ impl.
$\binom{4}{4} \cdot 2^0 = 1$ impl.

Fig. 1. Memory Layout for $n = 4$

that for all minterms $x' \in \{0,1\}^n$ x and x' agree on the non-dash positions of x, and $f(x') = 1$. We say that the implicant x' *covers* the minterm x. For example, if $01-0$ is an implicant of f, $f(0100) = f(0101) = 1$. In this paper, we say that an implicant is true if it is an implicant of f; we say that it is false if it is not an implicant of f.

A *prime implicant* of f is an implicant x that covers a set of minterms such that there exists no implicant x' that covers a superset of implicants. Intuitively, one cannot replace any 0 or 1 in x by $-$ and obtain another implicant of f.

Memory Layout. Given an n-bit Boolean function, we divide the memory into $n+1$ *sections*, indexed by $0 \leq k \leq n$, where k represents the number of dashes $(-)$ within an implicant. Each section is subdivided into *chunks* so that implicants within the same chunk have dashes at the same locations. For implicants on n bits and k dashes we can choose $\binom{n}{k}$ combinations of dash positions. We give chunks ordering using a combinatorial number system[1]. For example, for $n = 5, k = 3$, the first chunk is $**---$, followed by $*-*--$, $-**--$, $*---*-$, and so on.

Within a chunk in section k, there are $n-k$ non-dash variables which we denote by $*$. Therefore, each chunk in section k contains 2^{n-k} bits (for an example see Fig 1).

Merging. Recall that each chunk in section k contains exactly those implicants whose dash positions coincide. Consequently, the only pairs of implicants that differ in exactly one non-dash bit lie in the same chunk. We define a *merge* operation on a chunk of section k that, for each star position d, replaces that star with a dash, producing an implicant in section $k+1$. For example, for $n = 5, k = 3$, performing merge on chunk $*-*--$ can create implicants of the form $*----$ and $--*--$. This operation also marks implicants as merged, which is used to later determine which implicants are prime. During the merge, we call the quantity 2^d the *distance* between the implicants.

A given chunk can be produced by different input chunks. For simplicity and to improve spatial locality, an output chunk is produced from the input chunk that has a star $(*)$ instead of its first dash. Conversely, we write the merge result into the next section if and only if the output chunk is created by replacing leading star $(*)$ with a dash $(-)$. An example can be seen in Fig. 2. In the figure, note that all output chunks for an input chunk are next to each other; while merging, the algorithm writes output chunks sequentially.

[1]This means that the chunk containing implicants with n-bits and k-dashes on positions $0 \leq c_1 < c_2 < \cdots < c_k$ has index *binomnk* $- (\binom{c_k}{k} + \cdots + \binom{c_2}{k} + \binom{c_1}{1})$ [3].

Algorithm 1 shows the single-chunk merge step. There, I_j is the input chunk, O is the output section and M are the merged flags. The parameter $d_{\text{leading stars}}$ indicates when to write into an output chunk. In a first pass called *merge pass*, the algorithm loops over all sections k. For every chunk $j = 0, \ldots \binom{n}{k} - 1$ in section k, it invokes $\text{MergeImplicants}(k, I_j, O, M, d_{\text{leading stars}})$. In a second pass, called *reduce pass*, the algorithm declares any implicant *prime* if it is marked as true in I, but it has not been marked as merged in M.

Cost Measure. We define the cost measure $C(n)$ as the total number of bit operations for an n-bit function. We count each logical AND, OR and ANDNOT as a single bit operation, since these have the same cost on modern CPUs [4]. In this paper, we consider the theoretical cost implied by the first step of the described algorithm. Note that we ignore the 3^n bit operations induced by the reduce pass, since a single-pass implementation is possible (see section 3). For the analysis, we ignore bit operations used to rearrange bits that are due to implementation details. In all performance plots, we normalize runtime against $C(n)$.

Cost Analysis. In Algorithm 1, for a fixed section k, there are 2^{n-k-1} pairs of implicants to process at each of the k star positions. Each pair costs one AND and two OR bit operations, so a single call to MergeImplicants requires $3k2^{n-k-1}$ bit-operations. Summing over all $\binom{n}{k}$ chunks in sections $k = 0, \ldots, n$ gives the total cost of the *merge pass*:

$$C(n) = \sum_{k=0}^{n} \binom{n}{k} 3k2^{n-k-1} = n3^n.$$

In memory, all sections are laid out consecutively in two large bitmaps of size 3^n bits: One bitmap for the implicant truth values I and one for the merged flags M. In the best case, the algorithm loads 2×3^n bits from main memory into last-level cache, i.e. $Q(n) \geq 2 \times \frac{3^n \text{ bits}}{8 \text{ bits/byte}} = \frac{3^n}{4}$B. This yields an upper bound on the operational intensity of $I(n) = \frac{C(n)}{Q(n)} \leq 4n$. Since $I(n)$ is linear in n, the analysis suggests that the algorithm is compute-bound for large enough n.

3. OPTIMIZATIONS

In this section, we describe the optimizations we performed on the algorithm and their reasoning. We finish by mentioning optimizations which failed to deliver speedups.

Baseline (*baseline*). Our starting point represents each implicant as a full byte (a bool) in three separate arrays: *implicants* and *merged* which are used in the merge pass, as well as *primes* which is computed from the other two arrays in the reduce pass to mark unmerged true implicants as prime. It is used to verify the correctness of the algorithm, but pays a steep price in memory; each Boolean consumes 8 bits versus the ideal 1 bit per implicant. This also means that the algorithm runs out of memory for much smaller input sizes than the reference implementation (on a 32 GB machine, the largest implicant input size supported by the baseline is $n = 18$ bits).

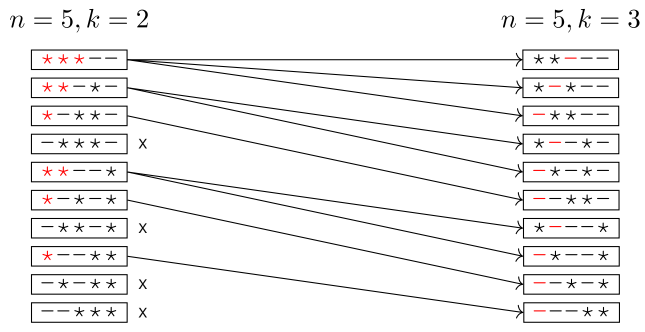

Fig. 2. Performing *merge* on a chunk outputs chunks where one of the leading stars is replaced by a dash. Input chunks are on the left, output chunks on the right. $n = 5, k = 2$.

Algorithm 1 MergeImplicants($k, I_j, O, M, d_{\text{leading stars}}$)

```
1:  for $d \leftarrow 0$ to $k - 1$ do
2:      for all pairs of terms $(t_1, t_2)$ differing only in bit $d$ do
3:          $r \leftarrow I_j[t_1] \wedge I_j[t_2]$        ▷ get result implicant
4:          $M[t_1] \leftarrow r \vee M[t_1]$        ▷ set merged flag for $t_1$
5:          $M[t_2] \leftarrow r \vee M[t_2]$        ▷ set merged flag for $t_2$
6:          if $d \geq d_{\text{leading stars}}$ then
7:              Write $r$ to $O$        ▷ write to output chunk
8:          end if
9:      end for
10: end for
```

64-bit "vectorization" (*bits*). In the *bits* implementation, we replace Boolean arrays with bitmaps of size 3^n. Each bit encodes a truth value for one implicant. We process implicants by loading them into 64-bit registers.

When the distance (offset) between bits representing pairs of implicants to merge is $2^d \geq 64$, we can use two 64-bit registers to compare 64 pairs of implicants in parallel. We call this process *inter-register merge*.

For smaller distances $2^d < 64$, both bits lie within the same register. Given a distance of 2^d and a register r, the merge is done by computing $r \wedge (r \gg 2^d) \wedge m$ where m is a mask that removes the non-result bits. The relevant result bits are now interleaved with zeros in the output (see Fig 3). One approach is to extract them by a series of shifts, ORs and masking ANDs (see Fig 4). We call this process *intra-register merge*.

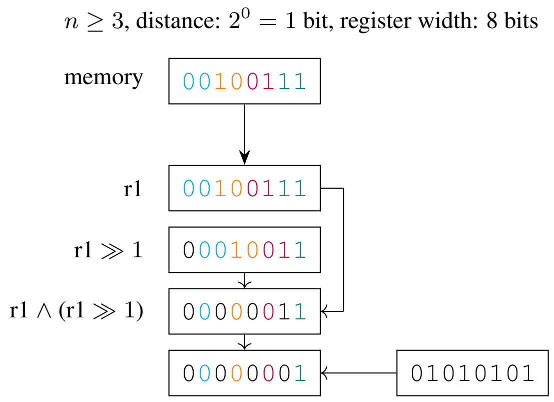

Fig. 3. Intra-register merges for small pair distances interleave result (colored) bits with zeros.

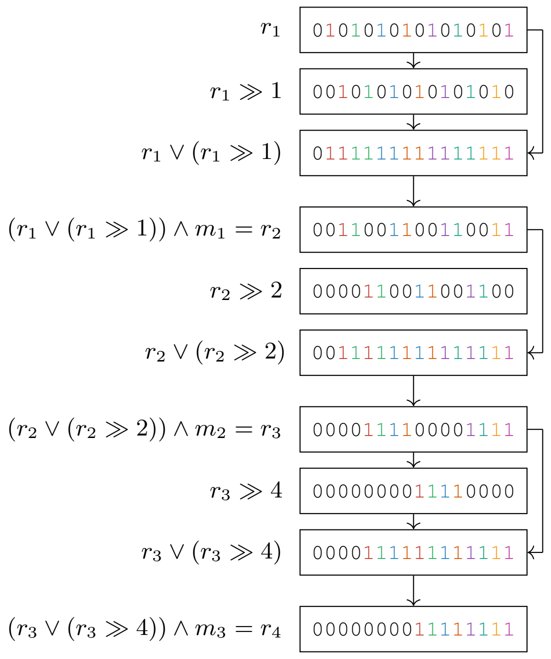

Fig. 4. Bit extraction: repeatedly shifting, ORing and masking extracts interleaved bits from a 16-bit register for distance $2^d = 1$

Improved Bit Extraction (*pext*). While extracting interleaved bits, a lot of cycles are wasted on shifts and masks that do not contribute useful work. On the x86 architecture, within the BMI2 instruction set, there is a specialized bit extraction instruction called `pext` (parallel bits extract) [4]. It takes a mask as an input to efficiently extract interleaved bits, placing the results into the lower half of the register. The *pext* implementation applies this instruction for small merge distances, replacing the earlier bit extraction method based on multiple shifts, ORs and masks.

Single Pass (*pext_sp*). The algorithm as described in Section 2 operates in two passes: a merge and a reduce pass. For larger implicant sizes (e.g., $n = 20$), the three bitmaps occupy: $3{\cdot}3^{20}$ bits ≈ 1.3 GB, which exceeds LLC capacity. As a result, the additional traversal of two-pass approach significantly increases the number of cache misses the algorithm incurs (see Fig. 9), adding at least one LLC miss per cache line of the *implicant*, *merged* and *primes* bitmaps.

To reduce memory traffic, we eliminate the merged bitmap entirely and maintain only a prime bitmap. The key idea is to fuse *reduce* pass into the *merge* pass. First at the start of $\mathrm{MergeImplicants}$, all true implicants in a chunk are marked prime. During the merge with result r, for input implicants t_i, t_j, the following update is performed:

$$\mathrm{prime}[t_i] \leftarrow \mathrm{prime}[t_i] \wedge \neg r$$
$$\mathrm{prime}[t_j] \leftarrow \mathrm{prime}[t_j] \wedge \neg r.$$

The update replaces the OR operations used to mark merged flags in the Algorithm 1. Since ANDNOT corresponds to one operation within our cost measure, the loop has the same number of operations and the effective overall cost measure stays the same.

By performing the reduce right after merging, the second pass is eliminated completely – besides reducing the number of cache misses for each cache line by 3 (one for each bitmap), it also allows us to run the algorithm for larger input sizes..

Improved Intra-Register Merge Order (*pext_sp_intra*). For every merge step that merges pairs with a distance $2^d < 64$, we load the entire input chunk register by register, perform merging for that distance 2^d on each one and then start over from the chunk beginning for the next distance 2^{d+1}. This means that in order to process a whole chunk of size 2^k implicants, every implicant is loaded k times from the cache into registers. For large chunk sizes, the implicants might be evicted from cache and re-loaded in the process.

Therefore, in the *pext_sp_intra* implementation, for distances $2^d < 64$ within a chunk, we load one register at a time and process it for all distances $2^d \in \{1, 2, 4, 8, 16, 32\}$ before moving to the next register. This reduces data movement between cache and registers: For a chunk of 2^k implicants, each cache line is now loaded $(k-5)$ times into registers. The new ordering also improves instruction-level parallelism (ILP), as there are few true dependencies between distances — only the final update to the *prime* bitmap. As a result, the CPU's execution units are kept busier, leading to better overall utilization.

In this implementation, outputs are written in six separate output regions, one per block size. Within each region, the writes are sequential in 32-bit units. Because each of these six output regions fits into a small number of cache lines that are fully written, spatial and temporal locality are preserved.

Blocking for Inter-Register Merging (*pext_sp_inter*). In the standard algorithm, merging a chunk of 2^k implicants processes one distance 2^d at a time, requiring each implicant to be loaded up to $k - 5$ times.

With inter-register blocking, we load multiple registers and process every r consecutive distances in parallel. For example when $r = 2$, for distances 2^6 and 2^7, we load: $i_1 = I[0..63]$, $i_2 = I[64..127]$, $i_3 = I[128..191]$, $i_4 = I[192..255]$ and perform the comparisons: (i_1, i_2), (i_3, i_4) for distance 2^6 and (i_1, i_3), (i_2, i_4) for distance 2^7.

This can be extended to 8 or 16 registers, processing 3 or 4 distances at once. For large n, where big-distance merges dominate, blocking significantly reduces memory traffic: the number of loads for the same implicant becomes $\lceil \frac{k-6}{r} \rceil + 1$. It also allows more computations to be performed per data load.

Pre-Loading Merge Order (*pext_sp_load*). As implicants are being merged, the implicant input and output indices, which follow the combinatorial order from section 2, as well as the $d_{\text{leading stars}}$ parameter are calculated at runtime. The overhead of this calculation is smaller for larger n, but still around 10% for $n = 22$; it can be removed by pre-computing the merge order indices.

Vectorization. Finally, using bigger register sizes than 64-bits, such as AVX2 256-bit registers and AVX-512 512-bit registers has multiple benefits. It improves data parallelism, since vector bit operations take the same time as on regular 64-bit registers, as well as amortize the cost from intra-register merging, since more implicants are being merged at once. We implemented each vectorization step separately using Intel intrinsics since the intrinsics don't map exactly to one another.

Since vectorization changes the register sizes, intra-register merging is now performed for distances $2^d < 256$ and $2^d < 512$ for AVX2 and AVX-512 respectively. This also improves the number of times each implicant bit is loaded into registers during a merge step to $\lceil \frac{k-8}{r} \rceil + 1$ and $\lceil \frac{k-9}{r} \rceil + 1$ for AVX2 and AVX-512 respectively, when blocking for r distances as before. However, for larger registers there is no equivalent of the `pext` instruction, so we use a mix of the earlier bit-extraction method shown in Fig. 4 with avx2 shuffles and avx512 permutes respectively.

Failed Optimization: Increasing ILP for Intra-Register Merge (*pext_sp_intra_ilp*). Computing the updated *primes* bitmap during an intra-register merge has a sequential dependency, making the *pext_sp_intra* implementation not able to take full advantage of ILP. In the implementation *pext_sp_intra_ilp*, the algorithm loads 4 registers (throughput of ANDN instructions on AMD Zen 4) and performs the intra-register merges on them for all small distances $2^d < 64$, trying to take full advantage of ILP. Upon implementing this version and comparing it to the *pext_sp_intra* optimization step, we did not see a significant improvement. We concluded that in the *pext_sp_intra* implementation, there are enough operations in the bit extraction version already to saturate the available pipeline slots.

Failed Optimization: Different Traversal Order. The algorithm does not require implementations to traverse chunks sequentially. This iteration order corresponds to a breadth-first search in a tree. Instead, a depth-first approach can be used, processing freshly written child chunks immediately after the parent chunk has been processed. This hopes to improve temporal locality.

We tried implementing this optimization, but we didn't see any significant speedup. Our explanation is that even though the sequential ordering incurs more LLC misses, these do not significantly slow down the program. This could be due to the high operational intensity which makes the algorithm access data that is in cache multiple times, so the traversal order does not matter too much.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the different optimization steps of our implementations through a series of experiments. We first detail our experiment setup, including the hardware configuration and range of input sizes used for benchmarking; afterwards, we discuss the experiments and the obtained results.

Experimental Setup. We ran all of our experiments on an AMD Ryzen 7 PRO 7840U 3.3 GHz processor, whose Zen 4 architecture has support for both AVX2 and AVX-512 SIMD. The machine has 32GB of RAM and the following cache sizes: 256KB per-core L1d cache, 8MB per-core L2 cache, and 16MB of shared L3 cache. For evaluating the NEON implementation, we used an Apple M4 Pro processor with 24GB of RAM.

We measure the number of cycles using the `rdtsc` instruction (hardware counters). Before performing the actual measurements, we perform one warmup run on different data to warm up the instruction cache, branch predictor etc. All our measurements are performed on cold data caches - we measure 5 runs and take the median value of cycles, cache misses and accesses.

Since the algorithm requires a lot of memory, the operating system (OS) does not by default back all allocated pages with physical memory. Instead, on the first page access, the kernel zeroes a physical page and maps it to the address space of the process. To avoid this from interfering with the measurements, after allocation, we make sure to access every page once to make sure it is backed by the OS before starting the measurements.

For compilation, we tested two different compilers: *gcc 13.3.0* with *-Ofast* and *clang 18.1.3* with *-O3* optimization flags. The highest implicant size n for which implicants could be allocated on this machine was $n = 23$, as larger sizes do not fit into RAM.

The experimental measurements use randomly initialized data for all implicant sizes $n \in [1, 23]$. Each implementation has passed tests on sparse (50%) and densely populated ($90\%, 99\%, 100\%$) test cases for each implicant size.

Different Compilers. Our tests showed that *gcc* generally achieves better performance when compiling and auto-vectorizing code (*bits** and *pext** implementations) whereas code that was vectorized by hand generally performs better if compiled with *clang* (*avx2**, *avx512**, *neon** implementations). The performance advantage of *gcc* spans from 45% on the *bits* implementation, to almost identical performance on some of the later non-SIMD optimizations, but there is no general rule. In this section, we always use the compiler which performs best on a given implementation and disable compiler auto-vectorization unless stated otherwise. The *hellman* implementation is always auto-vectorized, since it was designed for 256-bit vector registers.

Fundamental Optimizations. We first evaluate a set of fundamental initial optimizations, i.e. the implementations *baseline*, *bits*, *pext*, *pext_sp* and compare their performance against the reference implementation *hellman*. See figure 5 for the performance plot. Note that the *hellman* implementation uses a different algorithm based on a ternary number system and therefore its cost measure is also different. We still use the previously defined cost measure to normalize its runtime for comparison to our implementations.

The performance of our *baseline* implementation is very low, as it requires at least 8x more memory than the other implementations. For $n \leq 16$, the performance of *bits*, *pext* and *pext_sp* is lower than the reference implementation as they have more overhead when merging small chunks. This overhead includes special cases for handling small chunks that don't fit into a register, as well as calculation of the input and output chunk indices and the $d_{\text{leading stars}}$ parameter. However, because the runtime of the *bits* implementation for $n \leq 16$ is less than $100\,\text{ms}$, we argue that this performance difference does not matter in practice.

For $n > 16$, the performance of *hellman* drops due to it doing multiple passes over its working set and its working set not fitting inside LLC anymore. This is not a problem for our implementation since in our memory layout, there is only one pass over the entire working set. For these relevant larger input sizes, the *bits* implementation has comparable performance to the (vectorized) *hellman* reference implementation, while the *pext* implementation already achieves better performance. We also observe that using a single-pass algorithm does not seem to give much performance improvement over *pext*. However, removing one of the bitmaps allows us to run single-pass implementations up to $n = 23$. It also significantly helps further vectorized implementa-

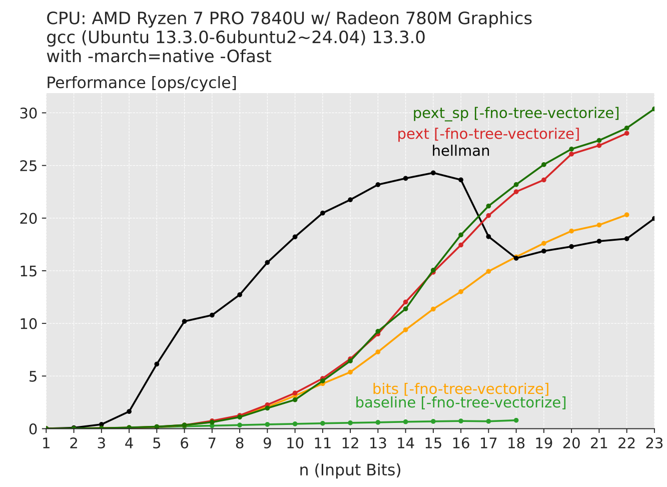

Fig. 5. Performance Plot of Fundamental Set of Optimizations: *bits*, *pext*, *pext_sp*.

tions.

Improved Merge Order. We now evaluate the second set of optimizations: intra-register merge order, inter-register merge order blocking and pre-loading the merge order. This corresponds to the optimizations *pext_sp_intra*, *pext_sp_inter2* and *pext_sp_load_inter2*. We compare the performance of these implementations against the reference implementation *hellman*. We chose a block size of 2 for the inter-register-blocked version of *pext_sp* since it resulted in the best performance in our measurements. See Fig 6 for the performance plot.

We observe that improving the intra-register merge order (*pext_sp_intra*) provides a speedup of 1.7x over *pext_sp*. We presume that this speedup is mainly due to improved ILP during intra-register merging, with a second factor being the reduced number of loads from cache/memory of each implicant. By blocking in the inter-register merging and thereby processing two distances at the same time, we get another speedup of 1.2x. Note that the speedups obtained for inter-register blocking are larger if we vectorize for SIMD (see next paragraph). Finally, we observe that pre-loading the traversal order results in a large speedup for smaller input sizes ($n \leq 16$) since it eliminates a lot of overhead in the implementations; the optimization provides less relative speedup for larger n.

Vectorization. We performed measurements to compare the best auto-vectorized implementations based on *pext* with the best hand-vectorized implementations based on AVX2, AVX-512 and NEON. In our measurements, we found that blocking by a factor of 2 in AVX2 and AVX-512, and by a factor of 4 in NEON yielded the best results. We show the performance of the AVX implementations as well as the reference implementation in Fig 7. We tested the perfor-

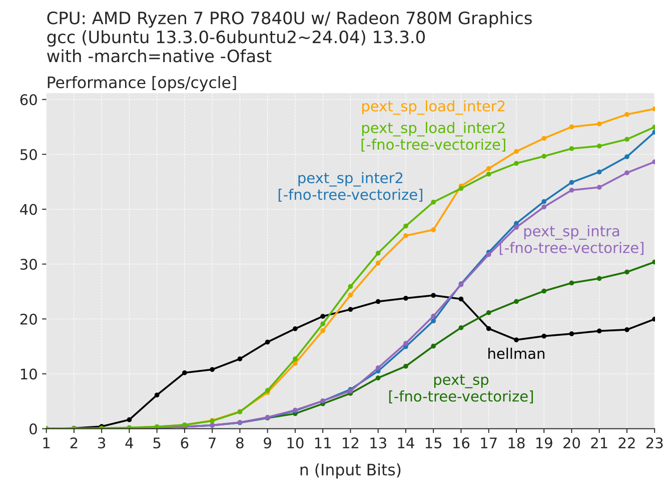

Fig. 6. Performance plot for second set of optimizations: Intra-Register Merging (*pext_sp_intra*), blocking for inter-register merging (auto-vectorized, non-vectorized *pext_sp_inter2*), pre-loading merge order (*pext_sp_load_inter2*).

mance of the NEON implementation on an Apple M4 Pro processor with 24GB of RAM (see Fig 8).

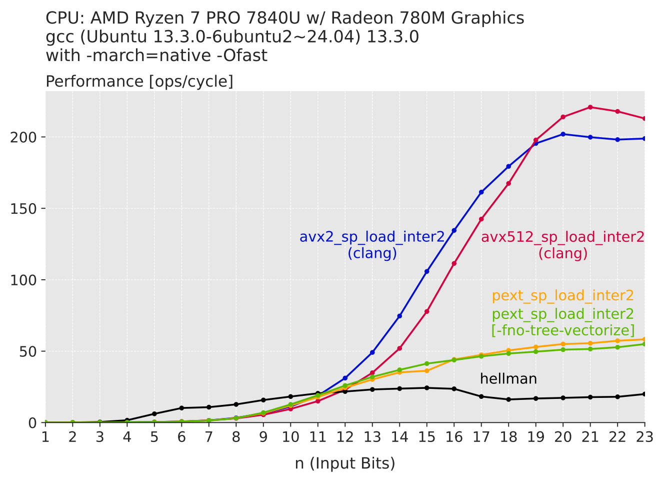

Fig. 7. Performance plot of Vectorized Implementations: auto-vectorized (*pext_sp_load_inter2*), AVX2 (*avx2_sp_load_inter2*), AVX-512 (*avx512_sp_load_inter2*)

We observe that there is a large difference between the auto-vectorized version based on *pext* and the hand-vectorized version based on AVX2, yielding a speedup of 3.7x. We also observe that the speedup between AVX2 and AVX-512 is much smaller, around 1.1x. Overall, we get speedups of 3.2x, 11.7x, 12.4x and 26.2x over the reference implementation for the pext, AVX2, AVX-512 and NEON versions

($n = 22$) and speedups of 67.5x, 230x, 202x, 83x over the baseline implementation ($n = 17$).

For NEON, we see that optimizations generalize well for this architecture. We observed a significantly larger speedup compared to the reference implementation, likely because the memory layout of the *hellman* implementation does not vectorize as efficiently for the 128-bit register width of NEON.

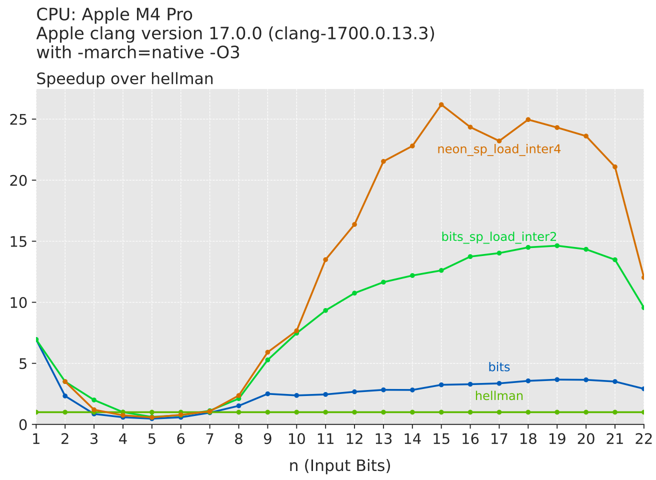

Fig. 8. Performance Plot of Vectorized Implementation for NEON: *neon_sp_load_inter4* Apple M4 Pro

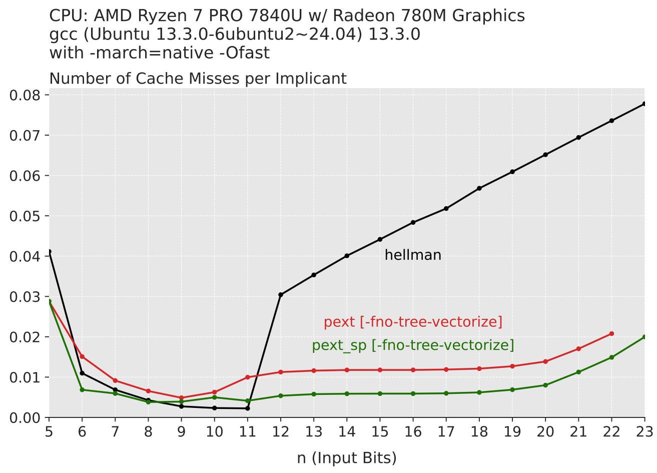

Fig. 9. Number of Cache Misses per Implicant for Implementations *hellman*, and non-vectorized *pext*, *pext_sp*

Cache Miss Rate. One of the reasons for our different memory layout was the better locality. To check whether this was actually the case, we measured the number of L1 data cache misses of the reference implementation as well as the non-vectorized implementations *pext* and *pext_sp*. We normalized the resulting number against the number of implicants and plotted the results (see Fig 9).

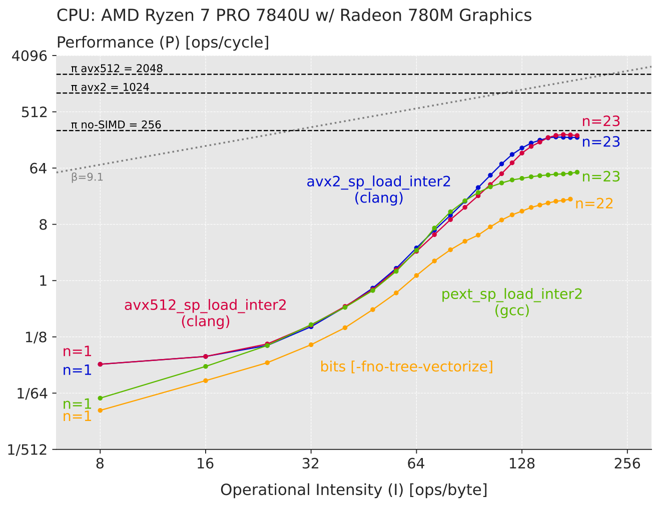

Fig. 10. Roofline Plot for selected implementations

We observe that for $n \geq 12$, our implementations have a much lower number of cache misses per implicant than the reference implementation where the number of cache misses grows approximately linearly with n. However, we can see that for $n \geq 21$, the number of cache misses per implicant starts to increase for our implementations as well. We presume that this is due to single chunks not fitting into L1 data cache anymore. In that case, capacity cache misses occur during a single merge step and the algorithm's performance decreases.

Roofline Plot. We used the best implementations for each vectorization target to create a roofline plot shown in Figure 10. We observe that as assumed before, the algorithm is compute-bound for large enough n. However, for AVX-512, we do not have enough RAM to run the program on an input size which would be compute-bound. We also observe that the actual performance still falls short of the theoretical performance roofline. Our explanation is that this is because our implementation performs more bit operations than the theoretical algorithm in the intra-register case. Further, the inter-register blocking may not be effective enough in reducing data movement between cache and registers.

5. CONCLUSION

We showed how the newly introduced memory layout results in better cache efficiency for the Quine-McCluskey algorithm. We achieved significant speedups for all vectorization targets. Future work could optimize merge steps for chunks that are larger than L1 cache as well as dynamically sparsify sections with few true implicants.

6. CONTRIBUTIONS OF TEAM MEMBERS

- **Aljaž.** Implemented *pext_sp_inter* blocking with modifiable block sizes. Worked on intra-register blocking ILP optimization and pre-loading merge order. Also worked on failed optimizations for DFS traversal and increasing ILP in inter register blocking.

- **Aleksa.** Implemented *pext_sp_intra* register blocking optimization, as well as worked on different unfruitful versions of the ILP optimizations. Focused on AVX-512 vectorization, translated the code to a different set of intrinsics for intra-register merging.

- **Aleks.** Vectorized for ARM NEON, worked on preloading merge order, intra-register blocking and DFS optimization. Worked on theoretical cost analysis.

- **Richard.** Conceptualized memory layout, implemented first versions of *baseline*, *bits*, *pext*, *avx2* implementations. Implemented initial versions of single-pass approach as well as inter-register blocking approach for the *avx2* vectorization target.

7. REFERENCES

[1] E. J. McCluskey Jr., "Minimization of boolean functions," *Bell System Technical Journal*, vol. 35, no. 6, pp. 1417–1444, 1956.

[2] Aleksei Udovenko, "DenseQMC: an efficient bit-slice implementation of the Quine-McCluskey algorithm," *arXiv preprint*, 2023.

[3] Edwin F. Beckenbach, *Applied Combinatorial Mathematics*, Wiley, New York, 1964.

[4] Intel, "Intrinsics Guide," 2025.