OPTIMIZATIONS ON THE PENTAGON ABSTRACT DOMAIN

Erik Henriksson
ETH Zürich
Zürich, Switzerland

Boris Peltekov
ETH Zürich
Zürich, Switzerland

ABSTRACT

In this work we present an implementation of join on the Pentagon Abstract Domain. We explain the basics of the operation, the improvements which is possible of the naive implementation and conclude with our resulting SIMD vectorized version. Our work gives insight in the huge benefits one can get by optimizing data storage and utilizing the vector instructions which are available on modern Intel processors. We hope this work will be useful to improve the performance on future program analyzers.

1. INTRODUCTION

Many huge bugs in computer science history has been caused by accessing array out-of-bounds, especially in the form of buffer overflow errors. The domain of abstract intrepretation has therefore gotten attention from researchers and new domains as the octagon and polyhedra domain has been invented. These allows to also capture relations between variables, something which is very useful when you want to prove invariants about array accesses. The "classical" interval domain is an example of a domain which is not precise enough to prove this. The main property you have to sacrifice for preciseness is speed.

Motivation. Logonzzo et al. proposes a new domain, Pentagon Abstract Domain, which allows for capturing intervals and strict upper bounds between variables. We have looked into this domain to optimize the most critical operations on the Intel Core architecture. We have found the join operation or more specifically the transitive closure performed inside a join to be the most critical and has thus focused our work on that operation.

Related work. There is a paper from Püschel et al. which explains how to do tiling and vectorization of the Floyd-Warshall algorithm, which also we use in the transitive closure. However, since this work is done on general graphs and our work is done on the special case where all edges have equal length, there is considerable amount of optimization. There is also an excellent manual from Intel, Intel 64 and IA-32 Architectures Optimization Reference Manual[1], which has proven very useful for us.

2. BACKGROUND: ABSTRACT INTERPRETATION AND TRANSITIVE CLOSURE

In the field of Abstract Interpretation the "join" operation between two abstract domains is one of the most commonly used, because it is responsible for correctly merging the inferred information coming from two different program flow paths. Since most programs heavily rely on branching, the "join" operation must be perfected both in terms of speed and correctness – it has to preserve as much information as possible, but also overapproximate the information coming from the merging paths. Note that there is an instantiation of the abstract domain at every program point and information is exchanged between them – the join operation is one example of such exchange. More specifically, every Pentagon domain [2] consists of two subdomains – interval and strict upper bounds (SUB). The interval subdomain contains a mapping from a variable to an interval of possible values, which the variable may take. In case when no bounds can be inferred, the variable is mapped to an artificial interval called TOP. The SUB domain is a set of strict inequalities between variables with the requirement that all of them must hold at the given program point. Note that no inequalities between expressions are allowed.

2.1. Mechanics of Pentagon join

Before joining the two domains, a closure operation is performed on each one of them. Its purpose is to enrich (in terms of precision) and exchange the available information between the SUB and interval subdomains. The closure can be described in 3 steps:

1. Infer stricter interval bounds given some inequalities. For example if $a \in [2, 8]$, $b \in [0, 6]$ and $a < b$ one could deduce that $a \in [2, 5]$ and $b \in [3, 6]$.

2. Infer inequalities given some intervals. For example if $a \in [2, 5]$ and $b \in [8, 10]$ then one could safely infer that $a < b$.

3. Transitive closure of SUB domain. If $a < b$ and $b < c$ then $a < c$.

It must process the SUB domain until reaching a stable point. More applications of transitive closure would not change the data. Once closure is applied to both domains, the join is delegated to interval and SUB domains respectively. They strictly adhere to the mantra of overapproximation – the result should be sound in the sense that no possible data assignment is missed. In case of intervals – if $a \in [2, 5]$ in the first domain and $a \in [7, 12]$ in the second, the join should be at least $[2, 12]$. Of course, further inflation of the result would not affect the soundness, but the precision will suffer, therefore it is undesireable. The same applies to SUB join – if an inequality is present in only one of the domains, it should not be included in the result.

2.2. Cost analysis

From now on we assume that the SUB domain is encoded as n by n matrix of 0's and 1's where n is the number of variables. 1 on ith row and jth column would designate that ith variable is less than jth variable. Steps 1 and 2 of the closure take $O(n^2)$. Step 3 takes $\Theta(n^3)$. It can be implemented using the Floyd-Warshall algorithm [3] augmented with logical operations 1 AND and 1 OR in the innermost loop, thus the exact number of logical operations is $2n^3$. We do not include in the cost measure the operations needed for updating the loop indices. The interval join is linear and the SUB join takes $O(n^2)$. In conclusion, the costliest step is the transitive closure of the SUB domain, therefore we concentrated our efforts to optimize it, taking the Floyd-Warshall algorithm as a starting point. Taking into account that in a single Pentagon join the transitive closure is performed twice, the total cost of a Pentagon join is $4n^3 + O(n^2)$ operations.

3. OPTIMIZED TRANSITIVE CLOSURE

3.1. Naive implementation - STL

We started with implementation of transitive closure using STL library. It uses STL maps and sets to encode that variable x is less than every variable in a given set. It is very concise in terms of memory footprint – it doesn't waste any space and supports runtime-varying number of variables, but a major downside is that the STL maps and sets use Red-Black trees, which generally do not exhibit any spacial locality. The STL implementation proved to be much slower than the others and there is no much to do about it, because, as the profiling investigation confirmed, most of the time is spent in STL internals.

3.2. Better naïve implementation Dense matrix (DM)

For the DM implementation we switched to different representation of the SUB domain n by n matrix of 1's and 0's. On top of it runs the standart Floyd-Warshall (FW) algorithm (with conditionals after the 2nd and 3rd loop). Because of these conditionals the op count may be less than $2n^3$. The DM and STL implementation yielded the same results, thus we used them as a basis for our verification framework. For the sake of comparison, the DM proved to be much faster than the STL implementation, because the data is packed in only one array and there is no extra auxiliary information stored (not the case with STL primitives).

3.3. Opportunities for improvement

The first step is to replace the conditionals from FW with logical operations. This streamlines computation and paves the way for further optimizations (bit-packing and SIMD processing). The downside is that the op count now is exactly $2n^3$. After this step the FW superficially resembles the matrix-matrix multiplication (MMM), for which there are widely known and efficient optimizations such as loop reordering, tiling for cache/registers, loop unrolling. After a closer look to FW, it's evident that the k loop should be the outermost one and i and j loops do not depend on each other – therefore they can be reordered.

It is known that putting i as the outermost loop is superior in MMM, but this is not possible in our case, which explains why MMM-like performance is not achieveable. The algorithm with k,i,j loop order shows spacial locality, because for every k-step the matrix is traversed in row-major order which coincides with the memory layout of the array (in C). Between successive iterations of k temporal locality is exhibited, but since k is the outermost loop and inside it the whole matrix is traversed, there is a benefit only when the whole matrix fits in last-level cache (n is small). Because of this fact, tiling for cache is crucial for achieving top performance in cases when n is large. Due to the independence of i and j loops, we introduced tiling on them (with configurable parameters) and observed tangible performance improvement.

3.4. Tiling for cache

The restriction on the k loop is imposed by data dependencies (otherwise the algorithm would not compute the full transitive closure), but as shown in [4], it can be relaxed through creating a FW version (FWabc) which works on 3 mutually distinct matrices (two source and one destination) rather than 1. In this special version the loops can be freely reordered and tiled. In order to unleash the potential of this optimization, tiling of the input matrix is needed. As next step of optimization process we implemented tiled version of FW (FWT, with configurable parameters) following the approach described in [4]. The essence of it is to call FWabc whenever possible (when 3 distinct submatrices are encountered) during the processing of smaller submatrices.

We observed about 4x improvement compared to naïve DM for large n (there is no big difference for small n, because the workset of DM fits in the cache). After that we started experimenting with different tile sizes – the only restriction is that n should be divisible by the tile size. The workset of DM consists of the whole matrix, whereas the workset of FWT is 3 tiles. We experimented with various workset sizes and found that crossing the L1 cache threshold doesn't affect performance, but when the workset gets bigger than the L2 cache, the performance is severely degraded. Based on our benchmarks, we decided not to implement two levels of tiling, but rather concentrate on loop unrolling and fine-tuning the innermost basic block.

Now we examine the operational intensity of DM and FWT to see if they are memory bound. We measure the memory traffic by approximating the number of cache misses. Let c be the cache size, s - cache block size (64 bytes in case of Core 2), b - block size (in FWT) and $M = \frac{n}{b}$ is the number of tiles in one row. From now on we assume that $n >> c$. In DM, if k and i are fixed, there will be $\frac{2n}{s} + 1$ misses. In total there will be $\frac{2n^3}{s} + n^2$ misses. The number of ops is $2n^3$, so the operational intensity is $I(n) = \frac{2n^3}{2n^3+sn^2} = \frac{2n}{2n+s} \in O(n)$. In FWabc, if $c > 3b^2$(i.e. the workset fits in cache), the operational intensity will be $\frac{2n^3}{3b^2}$, because the data will be loaded just once from memory. Following the scheme of FWT in [4], we count the number of tiles that need to be loaded during the computation and use it to calculate the cache misses and the operational intensity. If k is fixed, the first three phases load $2(M-1)+1$ tiles and phase 4 loads $(M-1)(2M-1)$ tiles. Loading of each tile costs $\frac{b^2}{s}$ cache misses. In total there will be $(2M^3 - M^2)\frac{b^2}{s}$ misses and after substitution of $M = \frac{n}{b}$ the result is $\frac{2n^3}{bs} - \frac{n^2}{s}$ misses. Note that the op count in FWT is still $2n^3$. So if we choose $b = \sqrt{\frac{c}{3}}$, the amount of bytes transferred will be at most $\frac{2\sqrt{3}n^3}{\sqrt{c}}$ bytes. The gain is approximately $2\sqrt{c}$ and the operational intensity is $O(\sqrt{c})$. This explains the performance advantage of the tiled version and also confirms that it is not memory bound.

3.5. Loop unrolling and scalar replacement

During the development of all variants of Pentagon join, we used configurable compile-time unrolling parameters and compiler flags and pragmas to hint the compiler which loops to unroll. The motivation for this decision is the possibility to automatically find good values for the unrolling factors. After finishing the bit-packed and SIMD implementations, we searched for the "sweet spots" of unrolling parameters using scripts. Taking them as a reference point and considering the number of available registers and the architecture of the Intel Core 2 processor, we implemented hand-written single static assignment (SSA) basic blocks using scalar replacement to avoid pointer aliasing. In principle some of the code (namely FWabc) operates on distinct arrays, so we used this fact during the development of the basic blocks. We strived to utilize the ideas found in micro-MMM (register blocking and reordering for instruction level parallelism), but there is a lower degree of register reuse in Floyd-Warshall than in matrix-matrix multiplication.

3.6. Bitpacking

An important step is to notice that we use boolean matrices and therefore can make use of bit-packing to reduce both memory footprint and increase the number of ops/cycle the processor can perform. The main idea is to represent every element by a bit instead of an integer. We also had to convert some of the binary operations to use the bit-wise variant. This is of course also possible in a SIMD implementation using the SIMD variants of the regular bit-wise operations.

3.7. Theoretical performance analysis

The objective of the theoretical analysis is to approximate the upper bound on performance of our SIMD implementation achievable on a particular piece of hardware and to pin-point the bottlenecks. Now we will examine our SIMD implementation with all previously described optimizations applied. The loop order inside the tiles is k,i,j – so no distinction between FW and FWabc is needed in this regard. There are three major places where a bottleneck can occur:

1. CPU's arithmetic resources - number of ALU's, distibution of the operations over the execution ports etc.
2. CPU-cache bandwidth (and sometimes latency)
3. Cache-RAM bandwidth

In order to update a single 128-bit chunk of data C[i,j:j+128] two logical SSE operations are needed – 1 AND and 1 OR. Despite that the OR depends on the AND, high throughput can still be achieved, because these operations for different j and fixed k,i can be done in parallel. So given the unrolling on j and hardware capable of out-of-order execution, this dependency would not be an issue. Our test hardware has 2 (Core 2 Duo) or 3 (Ivy Bridge) execution ports capable of processing a SSE logical operation each cycle. Considering only this, the peak performance should be 256 or 384 ops per cycle respectively. However there are other obstacles which prevent achieving this.

Now lets focus on the memory behaviour of the algorithm. Inside the i loop one particular bit (A[i,k]) has to be extracted and its value spread along a SSE register. Then this value is used during the whole j loop. Therefore, assuming enough work done in the j loop, the effect of this

memory load is negligible. Inside the j loop there are 2 SSE loads and 1 SSE store that need to be done. However the Sandy Bridge has only 1 load and 1 store port, so it is capable of only 1 load and 1 store per cycle. Consequently this limits the performance to 128 ops per cycle. We tried to alternate the dataflow of the algorithm, but we could not evade the stream-processing nature of the inside the j loop – two 128-bit values are fetched, 2 SSE operations are applied and then the result is written back. Despite that the distinct iterations of j are completely independent both in terms of memory usage and arithmetic operations, the bottleneck is the L1 cache bandwidth. No further data or register reuse is possible given our implementation layout. The positive side of it is that the data is accessed linearly row-wise and would not cause any unnecessary cache misses.

As discussed in section 3.4, the tiled FW is not memory bound and the slow RAM-cache bus will not impede its performance. As a result of the shown reasoning we conclude that no more than 128 ops per cycle can be sustained on our test hardware. Not suprisingly, the results confirm that – about 110 ops per cycle were achieved in our benchmarks. We attribute the performance gap to the fact that the execution ports for the logical SSE operations are shared with the ALUs responsible for incrementing the indices and calculating the memory addresses.

4. EXPERIMENTAL RESULTS

We ran all the tests on an Intel Core i7-3615QM, 2.3 GHz, Mac OS X 10.9 64-bit OS. We have written a benchmark suite which we used for our benchmarks. We used GCC 4.9 with -O3 optimizations enabled.

We analyzed performance for different config values we use in the code to find the sweet-spot for this particular CPU. The benchmark suite is named benchmark.cpp and can be found in the repository.

We have implemented four different variants of the algorithm; as described in the paper using STL maps, using dense matrix, using bit-packed matrix and using SIMD bit-packed matrix.

Our algorithm was implemented using C++ and we have benchmarked it to see the difference between the algorithm as described in the paper and our different implementations.

Figure (1) shows our final results. We can see the difference from STL Map (as described in the paper) to our fastest solution is about 10 000x. The highest improvement is by going to dense matrices, but there is still a solid 100x to the final SIMD implementation.

Figure (2) shows performance for different block sizes of the register blocking. We have 16 SIMD registers available and it should thus be logical if we saw a bump or similar in that range. The GCC compiler does however make very clever optimizations which we did not get rid of which makes the result better and better with larger register loops due to more freedom given to the compiler in reordering and allocating registers.

Figure (3) shows the effect of different tiling sizes for the cache. We clearly see 1024 is the sweet spot here, and that it does not matter that much for the 32-bit bitpacked int version.

Figure (4) shows the effect unrolling of the register loops has on performance. We have experimented a bit and the one showed is the best one we could achieve. We are still 25% below the GCC compiler, which also does unrolling and is a lot better than us on this.

5. CONCLUSION

Abstract interpretation in conjuction with Pentagon domain has proven itself as an efficient tool for capturing many common programming errors. But in order to be practical, the most costly operations in the domain have to be scalable. In this paper we used various techniques for optimizing the "join" operation. As a result the optimized implementation is within 35% of the theoretical maximum on our test hardware and is over 150x faster than the straighforward naïve approach (DM implementation). Some of the optimizations are also applicable in other areas in which fast transitive closure is needed.

6. REFERENCES

[1] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 248966-018 edition, March 2009.

[2] Francesco Logozzo and Manuel Fähndrich, "Pentagons: a weakly relational abstract domain for the efficient validation of array accesses," in *SAC*, Roger L. Wainwright and Hisham Haddad, Eds. 2008, pp. 184–188, ACM.

[3] Robert W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345–, June 1962.

[4] Sung-Chul Han, Franz Franchetti, and Markus Püschel, "Program generation for the all-pairs shortest path problem," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, USA, 2006, PACT '06, pp. 222–232, ACM.

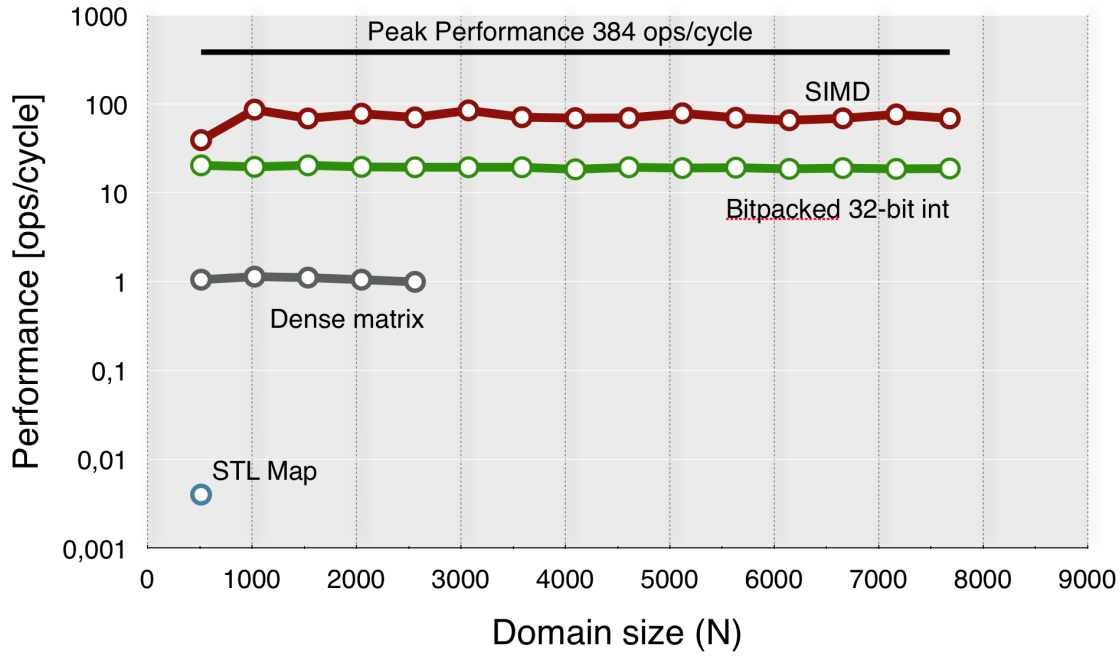

Fig. 1. Comparison of the four different implementations for different domain sizes

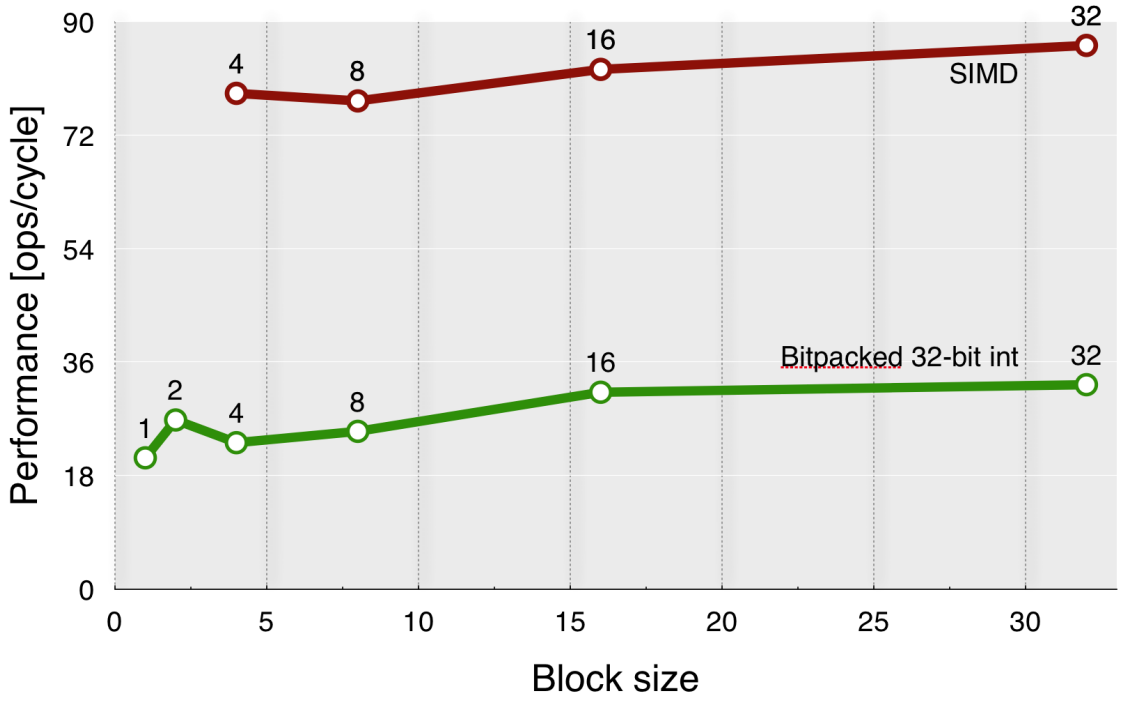

Fig. 2. Comparison of bit-packed 32bit integer to bitpacked 128bit SIMD vector for different block sizes of the register blocking

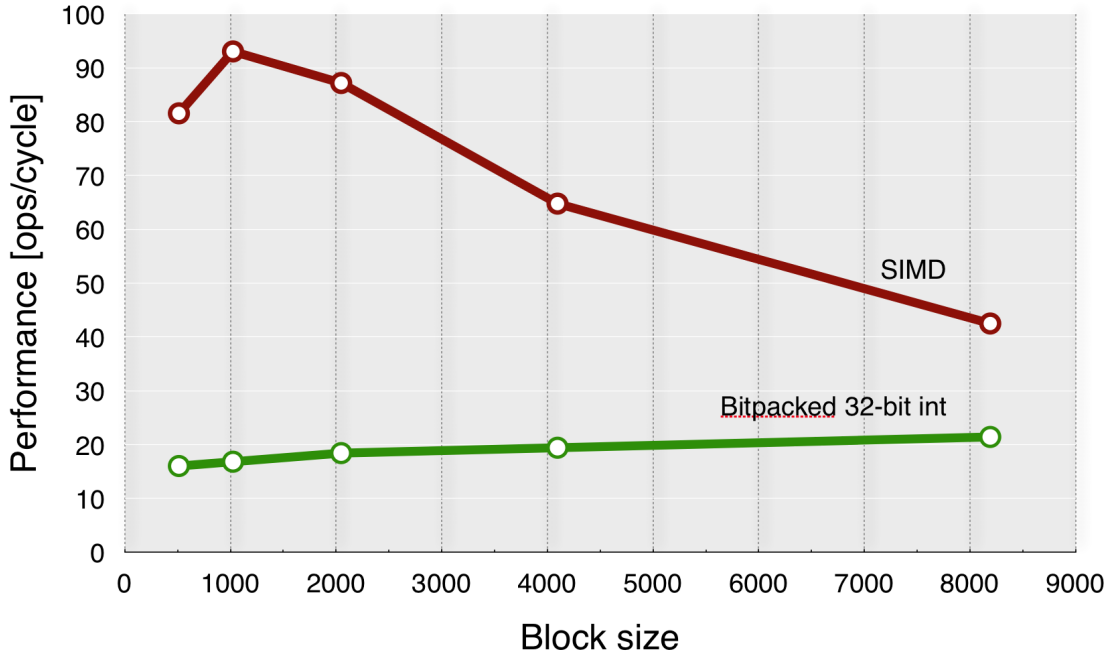

Fig. 3. Comparison of bit-packed 32bit integer to bitpacked 128bit SIMD vector for different cache block sizes

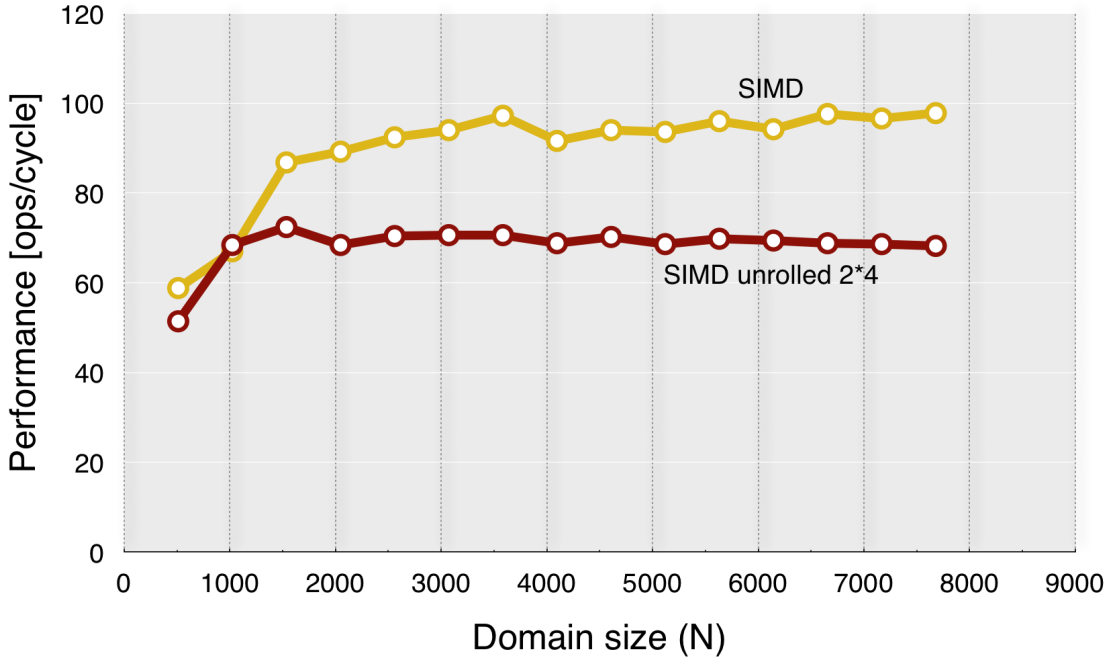

Fig. 4. Comparison of 128-bit SIMD for different unroll factors. Note: The yellow non-unrolled version is partly unrolled by GCC