OPTIMISING SEQUENTIAL PERFORMANCE OF BELIEF PROPAGATION FOR TOP-N RECOMMENDATION

Henry Trinh, Richard Hladík, Stephanie Ruhstaller, Sarah Tröndle

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Recommender systems are omnipresent in today's online shopping and streaming services. A scalable alternative to matrix factorization is belief propagation as it supports incremental updates of the model. This paper presents an efficient and fast belief propagation implementation in C for single-core, applying various optimisation ideas and techniques. We evaluate the performance on the MovieLens dataset and achieve a speedup of at least $100\times$. Moreover, the effects of the optimisations on speedup grow with the input size.

1. INTRODUCTION

Motivation. In the age of digitalization a majority of people spend their time on different platforms for entertainment, online-shopping and other fields. With the number of items increasing tremendously, the consumer is challenged to choose a small set of items that meets his interests and preferences. To tackle this problem, recommender systems were invented. Since recommender systems play a big role in e-commerce profitwise, many companies have an incentive creating recommender systems that are accurate, robust, scalable and fast. However, these characteristics are hard to fulfil with incomplete, uncertain, inconsistent and/or intentionally contaminated data. Further, since new data keeps coming in frequent intervals, collaborative filtering techniques like matrix factorization require solving the entire problem again and impose computational limitations for large-scale deployment.

Ayday et al. [1] and Ha et al. [2] propose a different approach where the relation of the users and items are represented as graphical models. The idea is to calculate the marginal probability distribution of the nodes which corresponds to predicting the ratings of the items for users. Since computing the marginal probability is intractable in general, they opted to approximate it with the *belief propagation* (BP) algorithm which grows linearly with the number of nodes (users/items) and therefore does not have the computational constraint such as the matrix factorization algorithm.

Many papers about BP focus on the algorithmic site for optimisation. Our focus is a fast and efficient C implementation of the BP that takes into account the memory hierarchy and vectorisation.

Contribution. We provide a BP implementation based on Ha et al. [2] optimised for sequential performance. The applied optimisations range from algorithmic changes over standard C optimisation techniques that include unrolling loops and scalar replacement to decreasing memory usage and utilising SIMD instructions. Our BP implementation outperforms the libDAI library [3] when applied to the specific problem of top-N recommendation.

Related Work. BP is also used in decoders for polar codes. For better performance, Ren et al. [4] propose an efficient early termination scheme based on convergence properties of *log-likelihood ratio* messages of BP polar decoders. They showed that for an iteration limit of 30, their method reduces the number of iterations to 8–9 on average, with only a negligible reconstruction degradation.

There are also works that focus on parallelisation. Mendiburu et al. [5] show one of the first parallel designs for BP with the Message Passing Interface paradigm, while Zheng et al. [6] propose novel approaches to parallel message computation in BP using GPU parallelization.

Our focus in this paper is on single-core performance where we do not concern ourselves with the convergence properties of BP and only consider running the algorithm for a fixed number of iterations.

2. BELIEF PROPAGATION & TOP-N RECOMMENDATION

In this section, we start by formulating belief propagation as described by Ha et al. [2]. Then we describe what role it plays in top-N recommendation, and provide its cost analysis.

2.1. Belief propagation

Belief propagation is a general method for performing inference on graphical models that comes in many flavours and shapes. For the purpose of this article, we use *belief propagation* solely to refer to the algorithm of Ha et al. [2, Section 2.1], which we describe in the following paragraphs.

The input of BP is a bipartite undirected graph $G = (V, E)$. Each of the vertices $v \in V$ can be in one of S *states*, labelled $x_1, \ldots, x_S \in X$. There is a joint probability mass function $p : X^V \to [0, 1]$ (assumed intractable and directly inaccessible to us) that captures the probability of each possible assignment of states to vertices.[1] The goal of the algorithm is, for each vertex v_i, to calculate its marginal distribution $p_i : X \to [0, 1]$:

$$p_i(x_j) = \sum_{\mathbf{s} \in X^V : s_i = x_j} p(\mathbf{s}) = \Pr_p[s_i = x_j]$$

To model p, the algorithm is also given a collection of node potentials $\{ \phi_i \}_{i \in V}$, with $\phi_i : X \to [0, 1]$, and a collection of edge potentials $\{ \psi_{ij} \}_{ij \in E}$, with $\psi_{ij} : X \times X \to [0, 1]$. Intuitively, node potentials capture our priors about which states each vertex inclines to, while the edge potentials capture the influence the state of a vertex has on the state of its neighbours.

In classical BP, it is assumed that p can be explicitly expressed in terms of ϕ_i and ψ_{ij}. In our case, where p itself is inherently more fuzzy, we make no such claim, and content ourselves with saying that ϕ_i and ψ_{ij} allow us to approximately model p.

Message passing. The BP algorithm itself consists of two procedures: *propagation* and *belief calculation*.

Propagation happens for a predefined number of iterations. For each vertex i and each its neighbour j, we compute a *message* $m_{ij} : X \to \mathbb{R}^+$ as follows:

$$m_{ij}(x_c) = \sum_{x_d \in X} \phi_i(x_d) \psi_{ij}(x_d, x_c) \cdot \prod_{k \in N(i) \setminus j} m_{ki}(x_d).$$

In the actual algorithm, we then also normalise m_{ij} so that $\sum_{x_c} m_{ij}(x_c) = 1$. It can be shown that this does not change the output, but can help with precision issues. Initially, all messages are set to all-ones, i.e., $m_{ij}(\cdot) = 1$. Algorithm 1 contains the pseudocode of the propagation procedure.

Belief calculation happens at the end of the algorithm. For each vertex i, we compute its *belief vector* $b_i : X \to [0, 1]$ as follows:

$$b'_i(x_c) = \prod_{j \in N(i)} m_{ji}(x_c)$$
$$b_i = b'_i / \|b'_i\|_1.$$

Again, we have normalised each b_i so that $\sum_{x_c} b_i(x_c) = 1$.

Finally, the algorithm outputs b_i as the approximation of the sought marginal probability distribution p_i.

Algorithm 1: Vanilla belief propagation

```
Input: $G = (V, E), \{ \phi_i \}_{i \in V}, \{ \psi_{ij} \}_{ij \in E}$
Output: $\{ m_{ij} \}_{ij \in E}$
$m_{ij}(x) \leftarrow 1 \quad \forall ij \in E, x \in X$
for $iter = 0, \ldots, \mathcal{I}$ do
    for $i \in V$ do
        for $j \in N(i)$ do
            for $x_c \in X$ do
                $m'_{ij}(x_c) \leftarrow 0$
                for $x_d \in X$ do
                    $M_j^d \leftarrow \prod_{k \in N(i) \setminus j} m_{ki}(x_d)$
                    $m'_{ij}(x_c) \mathrel{+}= \phi_i(x_d) \psi_{ij}(x_d, x_c) M_j^d$
            $m'_{ij} \leftarrow m'_{ij} / \|m'_{ij}\|_1$
    Swap $m$ and $m'$.
```

2.2. Top-N recommendation

Now we describe the full algorithm of Ha et al. [2] and what role BP plays there.

Top-N recommendation. The task of top-N recommendation is as follows: we are given a list of triplets of the form (userID, itemID, rating), as well as an integer $r \in \mathbb{N}$ and a special user ID t (the so-called *target* user). Based on that, we want to recommend r items to user t such that t is as likely as possible to like those items. In our concrete instantiation, items are actually movies and ratings are integers in $\{1, \ldots, 5\}$.

Application of BP. Ha et al. [2] solve top-N recommendation by converting the input ratings into a suitable graph that can be then passed on to BP. They describe in detail how this construction can be tuned. Our focus is on optimizing the BP part of the process, but we implemented and now describe the rest of it for completeness.

The ratings are represented as a bipartite graph, with users and movies as vertices and ratings as edges. Only edges whose rating satisfies a certain threshold are kept (different thresholding methods are described in the paper; we use the threshold that keeps edges above user's average rating). BP is used to model the probability that user t likes a vertex. Thus, the possible states of each vertex are $X = \{\text{LIKE, DISLIKE}\} = \{L, D\}$. (Note that user vertices are also modelled, although in the end we are only interested in movie vertices.)

Potentials ϕ_i are set depending on target user's rating of i (if applicable, otherwise $\phi_i(L) = \phi_i(D) = 0.5$). For ψ_{ij}, we set independent of i, j: $\psi_{ij}(L, L) = \psi_{ij}(D, D) = 0.5 + \alpha$; $\psi_{ij}(L, D) = \psi_{ij}(D, L) = 0.5 - \alpha$, where α is a small positive constant. In our actual implementation however, we had to use $\alpha = 0.3$, as setting $\alpha = 0.0001$ as suggested in the paper produced beliefs very close to 0.5.

After constructing the graph in this fashion, BP is run

[1] We use the notation X^V to denote the space of all $|V|$-dimensional vectors $\mathbf{s}$ indexed by elements of V where $s_v \in X$ for each component s_v. Given $v_i \in V$, we also sometimes write s_i instead of s_{v_i} for brevity.

for a fixed number of iterations, or until the change of the message values between iterations is less than some predefined threshold. Finally, the beliefs b_i are computed, the movie vertices sorted according to $b_i(\text{LIKE})$ and the r vertices with the largest $b_i(\text{LIKE})$ are output.

Cost analysis. As mentioned in Section 3, our work deals solely with optimising Algorithm 1, thus, we only focus on it in our cost analysis. We have chosen the number of flops as the cost measure. Analysing Algorithm 1 yields

$$4 \cdot (\deg(i)(\deg(i) - 2) + 3\deg(i)) = 4\deg(i)^2 + 4\deg(i)$$

flops for each $i \in V$ on lines 4–9, plus $3\deg(i)$ flops for each $i \in V$ on line 10. In total, we get

$$\begin{aligned} C(G) &= \sum_{i \in V} \left(4\deg(i)^2 + 7\deg(i)\right) \\ &= 14m + 4\sum_{i \in V} \deg(i)^2 \\ &\geq 14m + 16m^2/n \\ &= 14m + 8m \cdot \overline{\deg}, \end{aligned}$$

where $\overline{\deg}$ is the average degree and we used respectively 1) the fact that $\sum_{i \in V} \deg(i) = 2|E|$, 2) the Cauchy-Schwarz inequality $\langle \mathbf{x}, \mathbf{y} \rangle \leq \|\mathbf{x}\| \cdot \|\mathbf{y}\|$ for $x_i = \deg(i)$ and $y_i = 1$, and 3) the fact that $2m = n \cdot \overline{\deg}$.

2.3. Faster belief propagation

Because the context is convenient, we now describe the theory behind our algorithmic optimisation that is later discussed in Section 3. Looking closer at Algorithm 1, we can see that for a given i, the calculation of M_j^d is very similar for different j. We can exploit that to insert

$$M^d \leftarrow \prod_{k \in N(i)} m_{ki}(x_d)$$

before line 4, and replace line 8 with

$$M_j^d \leftarrow M^d / m_{ji}(x_d),$$

thus replacing $\deg(i) - 2$ multiplications with one division. We can also move the multiplication by $\phi_i(x_d)\psi_{ij}(x_d, x_c)$ outside the j-loop since the potentials do not depend on j.

This drastically decreases the flop count to $C(G) = 26m + 8n$.

3. OPTIMISATION STEPS

This section presents all optimisation steps seen in Fig. 1. Initial measurements have quickly shown that BP is by far the largest bottleneck of the top-N recommendation algorithm. Hence, our work focuses solely on optimising the propagate procedure described in Algorithm 1.

Baseline ⟨1⟩. This project has two baselines. The first makes use of the C++ library libDAI, which implements belief propagation on a factor graph. The second baseline, and basis for all optimisation, was implemented from scratch and strictly follows the belief propagation algorithm presented in Section 2. It was implemented in a way that tries to maximise sequential array access and uses a CSR (compressed sparse row) inspired format for storing incoming messages.

Namely, we store a list `off` of vertex offsets. The incoming messages m_{ji} of vertex i are stored in array `in` contiguously at indices $\texttt{off}[i], \ldots, \texttt{off}[i{+}1]{-}1$. We additionally maintain an array `out` of indices into `in`, such that if $\texttt{in}[k]$ represents message m_{ij}, then $\texttt{in}[\texttt{out}[k]]$ represents message m_{ji}. This way, calculating the product of incoming messages can be done contiguously, and random access is only needed when sending a message to a neighbour.

Low-hanging fruit ⟨2⟩. The first optimisations applied on top of the baseline implementation were scalar replacement and writing out explicitly the inner two by two loops starting on line 5 of Algorithm 1. This optimisation was then used as basis for all following steps.

Using scalar replacement for values that are accessed through pointers should allow the compiler to use more optimisations since the possibility of aliasing is excluded, and can possibly save memory accesses when a pointer is dereferenced once instead of multiple times. Writing out the inner two by two loop saves on control flow instructions in addition to saving computation as well as enabling the compiler more predictability.

Precompute message products ⟨3⟩. In the next step, we switched to the asymptotically faster algorithm introduced in Section 2.3. This improved the time complexity from $\mathcal{O}(m \cdot \overline{\deg})$ to $\mathcal{O}(m + n)$, while decreasing the flop count to $C(G) = 26m + 8n$. On the other hand, it introduced additional division operations which have worse latency and throughput.

Graph padding. In anticipation of unrolling and SIMD, optional graph padding was added to optimisation ⟨3⟩. The idea behind it was to add $\mathcal{O}(n)$ dummy edges to the graph that would not have any influence on the result, but would make the number of neighbours of a node divisible by 4 or 8 respectively. As a consequence, vectorised code could use aligned instead of unaligned loads. Besides that, no small sequential leftover loop would be needed when the j-loop from Algorithm 1 is unrolled and vectorised. To see for every unrolling and vectorisation step whether the additional nodes or the leftover loop are the bigger overhead, the graph padding could be activated using a compiler flag (by default deactivated). The graph padding was included in all successive optimisation steps.

Unroll inner loop ⟨4 & 10⟩. Unrolling the inner loop, the j-loop in Algorithm 1, was done once on basis of op-

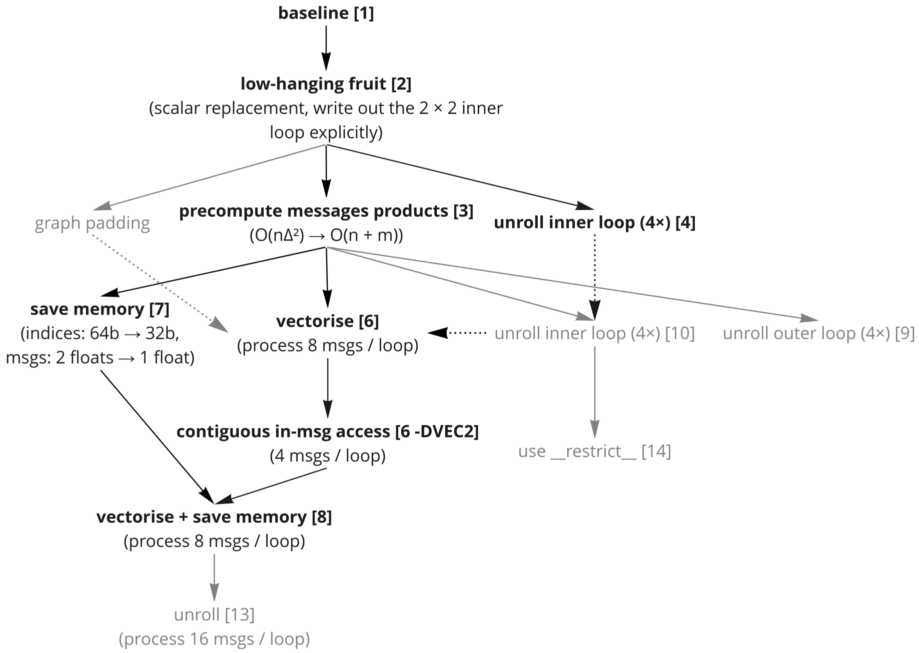
baseline [1]
low-hanging fruit [2]
(scalar replacement, write out the 2×2 inner loop explicitly)
graph padding
precompute messages products [3]
($O(n\Delta^2) \to O(n + m)$)
unroll inner loop (4×) [4]
save memory [7]
(indices: 64b → 32b,
msgs: 2 floats → 1 float)
vectorise [6]
(process 8 msgs / loop)
unroll inner loop (4×) [10]
unroll outer loop (4×) [9]
contiguous in-msg access [6 -DVEC2]
(4 msgs / loop)
use __restrict__ [14]
vectorise + save memory [8]
(process 8 msgs / loop)
unroll [13]
(process 16 msgs / loop)

Fig. 1. Diagram of optimisation steps

timisation $\langle 2 \rangle$ and once on basis of optimisation $\langle 3 \rangle$. In both implementations, the loop was unrolled by a factor of 4. This allowed for more scalar replacement. Since the for-loop iterations are independent, computing all of them at once allows for more instruction level parallelism (ILP). The difference in the two unrolling implementations is that thanks to optimisation $\langle 3 \rangle$, the second version unrolls a loop with much fewer computations, which means that ILP can be expected to have less impact.

Unroll outer loop $\langle 9 \rangle$. In a different implementation, based on optimisation $\langle 3 \rangle$, we examined what happens if the outer loop is unrolled. Specially, the i-loop from Algorithm 1 was unrolled by a factor of 4. Since with optimisation $\langle 3 \rangle$ the computation of messages product was pushed to the outer loop, it was of interest to see if unrolling this loop would allow for increased ILP. Or if, with 4 inner loops run during one unrolled iteration of the outer loop, the loss of locality would result in decreased performance.

Vectorise $\langle 6 \rangle$. SIMD was introduced on top of $\langle 3 \rangle$. First, Intel AVX2 intrinsics for vector instructions were introduced that very closely followed the instruction sequence of scalar computation. This way, 8 messages were processed together per loop iteration.

In a second step, the vector instructions were optimised with shuffling in-messages after loading them into vector registers to have more sequential access. Additionally, the number of messages handled during one iteration was reduced to 4.

Even though, looking at the assembly compiled from optimisation $\langle 3 \rangle$, it is visible that the compiler already introduced some vector instructions, writing them explicitly should allow for a greater use of SIMD.

Save memory $\langle 7 \rangle$. Since it was reasonable to assume that memory accesses would also be a limiting factor in performance, the number of unnecessarily loaded bytes were reduced. This was done in two places.

The LIKE and DISLIKE states have to add up to 1 according to the algorithm definition. This means, instead of storing two states, we could store only 1 and compute the other with a simple subtraction, saving a memory access.

In a second step, the type of indices was changed from `size_t` to `uint32_t`. This is sufficient for any input that needs less than 48 GB of memory.

This compaction allowed for more values being read when loading 1 cache block, making memory access more efficient.

Combine vectorisation and saving memory$\langle 8 \rangle$. In the next step, optimisation $\langle 6 \rangle$ and $\langle 7 \rangle$ were combined. This optimisation was expected to work well together, since it combines greater computational efficiency with greater memory efficiency. Additionally, the number of messages processed per iteration was increased to 8, to account for the increased data efficiency.

Unroll $\langle 13 \rangle$. On basis of optimisation $\langle 8 \rangle$ another unrolling was tested. Now, the number of messages processed per iteration was increased from 8 to 16. The idea was to see if we could make more use of optimisation $\langle 7 \rangle$ to increase ILP or if this would lead to a loss of locality and slow down

Processor	Intel® Core™ i5-7200U
Code name	Kaby Lake [7]
Cores	2 [7]
Threads	4 [7]
Frequency	2.50GHz (Turbo Boost disabled)
L1 cache	2 x 32 KB 8-way set associative instruction cache [8] 2 x 32 KB 8-way set associative data cache [8]
L2 cache	2 x 256 KB 4-way set associative cache [8]
L3 cache	3 MB 12-way set associative shared cache [7, 9]
RAM	2 × 8 GB DDR4 2133 MT/s

Table 1. Experimental platform

computation.

Use __restrict__ ⟨14⟩. In a next step, based on optimisation ⟨10⟩, the `__restrict__` keyword was used to ensure that the compiler did not make any assumptions about memory aliasing and therefore hindering some optimisations.

4. EXPERIMENTAL EVALUATION

This section starts by describing the experimental setup in the first part. In the second part, we show the results of our experiments evaluating the optimisations steps described in the previous section.

4.1. Experimental setup

This section provides details about the experimental setup by describing the platform and compiler used, the test inputs and how measurements were performed.

Platform. All the experiments were conducted on an Intel® Core™ i5-7200U @2.50GHz machine which is described in Table 1. The processor specification states a maximum memory bandwidth of 34.1 GB/s for two channels [7]. Halving this number gives 17.05 GB/s per channel in theory. For comparison, Table 2 shows the best memory throughput rates actually measured on this platform for a single core using John D. McCalpin's STREAM memory benchmark [10, 11]. Taking the average over all functions in Table 2 gives 15'054.4 MB/s which is around 15 GB/s. Dividing this 15 GB/s by 2.5 GHz, results thus in an average memory bandwidth $\beta = 6$ bytes/cycle. Furthermore, the bandwidths for L1, L2 and L3 caches are 81, 29 and 18 bytes/cycle according to [8].

Compiler flags. The program was compiled using GCC 12.1.0 with `g++ -Ofast -march=native`.

Input data. Test inputs were based on the MovieLens dataset [12]. Various sizes were generated by taking the ratings of users with identifier smaller than some number for different numbers and then normalising / desparsifying the user and movie identifiers separately, mapping them to $\{1, \dots, \#\text{users}\}$ and $\{1, \dots, \#\text{movies}\}$ respectively. This section's plots display the results of either a small or a big dataset composed from inputs generated this way. The small dataset represents graphs with ≈1k–10k vertices and ≈1k–80k edges based on the MovieLens Latest Small dataset [13]. The big dataset represents graphs of ≈40k–200k vertices with ≈2–20 million edges based on the MovieLens 25M dataset [14].

Function	**Best Rate [MB/s]**
Copy	14,273.6
Scale	13,646.1
Add	16,250.3
Triad	16,047.6

Table 2. Memory throughput of experimental platform for a single core measured using John D. McCalpin's STREAM memory benchmark [10, 11].

Measurement. For measuring the number of cycles, the `RDTSC` instruction has been used. In addition, the number of flops has been calculated directly in the measurement code. As mentioned in Section 3, we solely optimised and measured the belief propagation procedure of Algorithm 1. The number of iterations was set to 10 with the exception of optimisations ⟨1, 2⟩ measured on the big dataset. In those cases, only a single iteration was run due to the runtime growing prohibitively large otherwise.

4.2. Experimental results

The following paragraphs demonstrate the incremental performance and runtime improvements achieved by successively applying the optimisations described in Section 3. Additionally, interesting failed attempts are discussed. Finally, a roofline plot with the final optimisation step and the relative end-to-end speedup over the baseline ⟨1⟩ are shown.

Base implementation / optimisation. Since there are two base implementations, one implemented from scratch and the other using the libDAI library, it is interesting to see which one is faster. For this purpose, the relative speedup in runtime of the different base implementations over the from scratch implementation on the small dataset is displayed in Fig. 2. We see that libDAI library ⟨0⟩ uses the most cycles compared to the others. Already implementing the algorithm from scratch provided slight runtime benefits (baseline ⟨1⟩ in Fig. 2). Presumably, this came from the benefits of not using a library: less bound checks and function calls, and inlining of functions. Another reason is that baseline ⟨1⟩ was implemented in a way to maximise sequential array access and used a CSR inspired format for storing all the

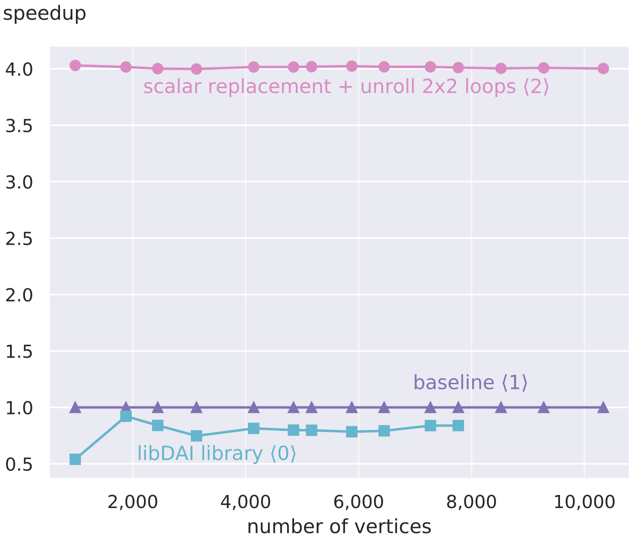

Fig. 2. Relative speedup of optimisation step ⟨2⟩ and the libDAI library ⟨0⟩ over the baseline ⟨1⟩ on the small dataset.

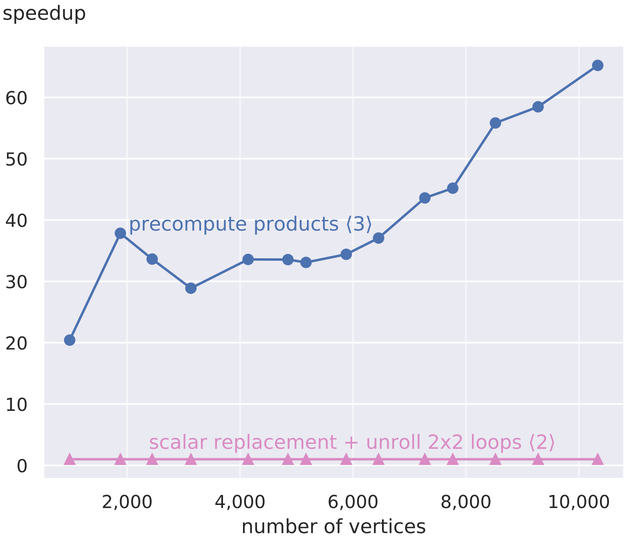

Fig. 3. Relative speedup of optimisation step ⟨3⟩ over optimisation ⟨2⟩ on the small dataset.

incoming messages. Applying optimisation ⟨2⟩ to the baseline ⟨1⟩ gives a 4× speedup. One reason for this is that the product calculation at line 8 of the algorithm can be reused by the outer loop iterating over states. Another important reason is scalar replacement, which eliminates many pointer dereferences.

Reducing computation. Optimisation step ⟨3⟩ reduced the number of flops, but also introduced divisions, which is bad because division has worse latency and throughput than addition/multiplication on the target machine. Fig. 3 shows that on the small dataset, optimisation step ⟨3⟩ achieves a relative speedup of at least 20× over optimisation ⟨2⟩. This is because the average degree is large enough that the cost of the introduced division is offset by improved time complexity.

Saving memory and vectorisation. For further improvement on top of optimisation ⟨3⟩, optimisation ⟨7⟩ tried to reduce the memory footprint and optimisation ⟨6⟩ added vectorisation. We see in Fig. 4 a relative speedup of around 1.1× for reducing the memory footprint while vectorisation yields a relative speedup of 1.1–1.6× with respect to optimisation ⟨3⟩ on the small dataset. The relative speedup of the combined version of both optimisations (optimisation ⟨8⟩ in Fig. 4) is 1.3–2× which is greater than each of them separately.

Graph padding. To examine the effect of SIMD alignment we tried to pad the graph so that the list of neighbours / incoming messages for each vertex starts on an address divisible by SIMD size. In Fig. 5, we see graph padding is slower compared to optimisation step ⟨8⟩. The latency and throughput on the platform do not differ between unaligned and aligned vectorised load. Therefore, using aligned loads did not benefit the relative speedup. Apart from that, graph padding increases the memory footprint and introduces excess computation. If padding is needed for most of the vertices, the memory overhead becomes significant. Since our test input data is sparse, we wanted test whether the negative effects on speedup are decreased on graphs of large average degrees. To this end, Fig. 6 displays the relative speedup over optimisation step ⟨8⟩ measured on complete bipartite graphs. Fig. 6 disproves the idea that padding could result in greater relative speedup on graphs of large average degree: The gap in relative speedup on bipartite graphs is not significantly decreased compared to Fig. 5.

Other unsuccessful optimisations. Optimisations ⟨9, 10⟩ tried to increase ILP of optimisation step ⟨3⟩ by unrolling the outer respectively the inner loop four times. Optimisation ⟨13⟩ tried to achieve the same thing by unrolling the inner loop sixteen times but based on ⟨8⟩. Fig. 7 shows that these optimisation steps could not increase the speedup relative to the optimisation steps they emerged from. Presumably, unrolling the outer loop resulted in worse locality whereas for the inner loop, the leftover loops have more iterations and possibly register spilling occurred. Optimisation step ⟨14⟩ using the restrict keyword is not shown in this plot as it almost coincides with the line representing optimisation step ⟨10⟩ which it was based on. This suggests the code already runs as fast as if no pointer aliasing was assumed without using the keyword.

Performance & Big inputs. The optimisations shown in Fig. 4 constitute the final successful steps taken. Therefore, it is interesting to finally see the actual performance

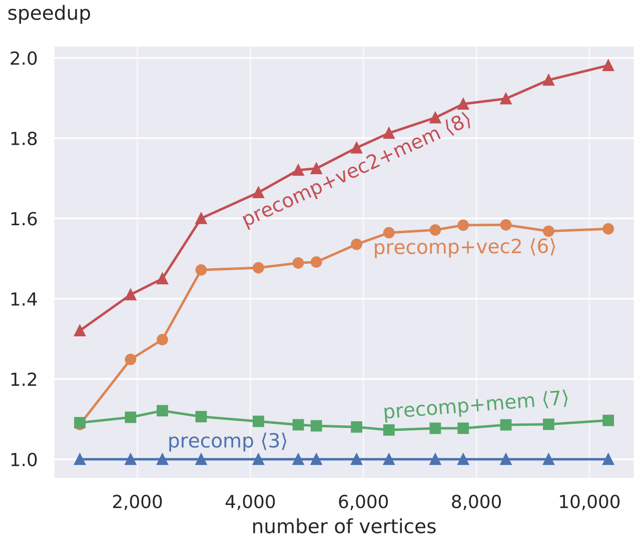

Fig. 4. Relative speedup of vectorisation and memory reduction optimisation steps over optimisation ⟨3⟩ on the small dataset.

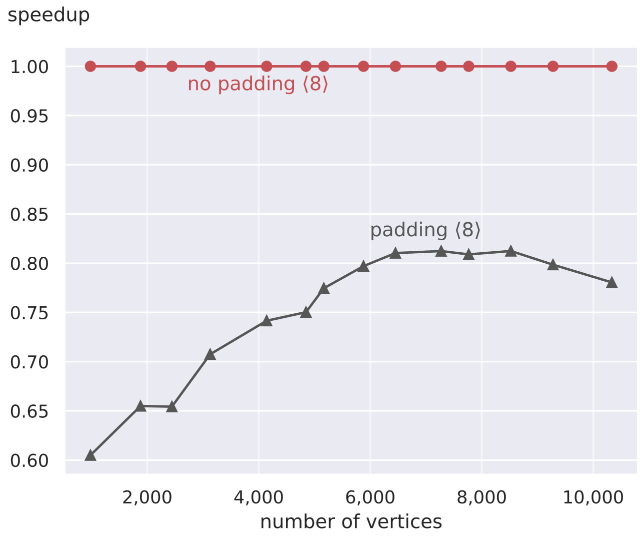

Fig. 5. Relative speedup of using graph padding on top of optimisation step ⟨8⟩ on the small dataset.

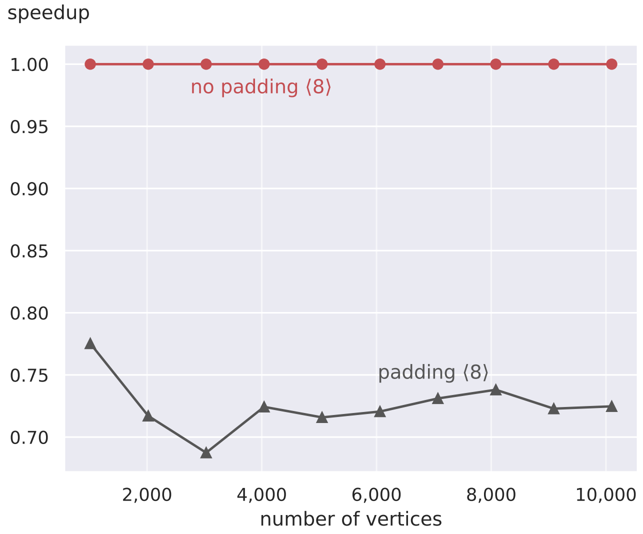

Fig. 6. Relative speedup of using graph padding on top of optimisation step ⟨8⟩ on complete bipartite graphs.

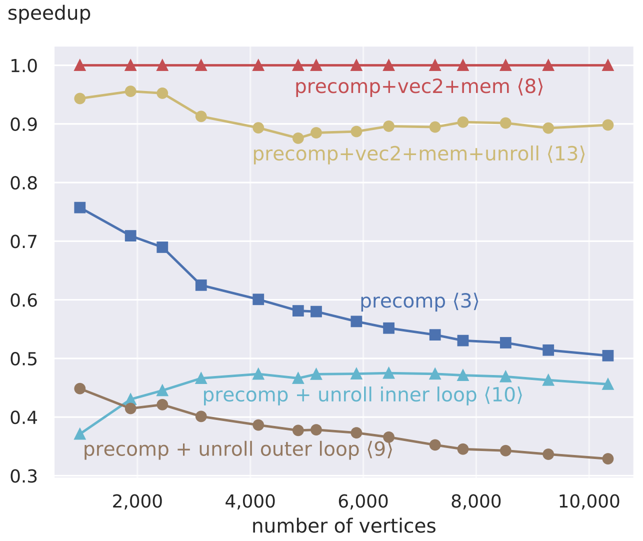

Fig. 7. Relative speedup of further unrolling added on top of optimisation step ⟨3⟩ by optimisations ⟨9, 10⟩, and on top of optimisation ⟨8⟩ by optimisation ⟨13⟩, all on the small dataset.

achieved. In addition, we only discussed what happens on the small dataset so far. To get a more complete view, Fig. 8 shows the performances of these optimisations on the small and big dataset combined. With the big dataset, the working set that doesn't fit into cache anymore. We see that for the bigger inputs the combined version is still the best but the memory saving has a much bigger impact on speedup. The vectorised versions drop faster than the non-vectorised versions as their advantages in more efficient computation get drowned out by slow memory accesses. All the optimisations displayed surpass the performances of the base implementations (mentioned in Fig. 2), whose performances were all below 0.05 flops/cycle (below 0.005 flops/cycle on the big dataset). For all versions, starting from the datapoint close to 50,000 vertices, the data structures didn't fit into L3 cache anymore.

Roofline. Optimisation step ⟨8⟩ is the final one. In order to see whether it is memory or compute bound and how close it is to the optimal performance, consider Fig. 10. It is obvious from Fig. 10 that all versions reside in the compute-

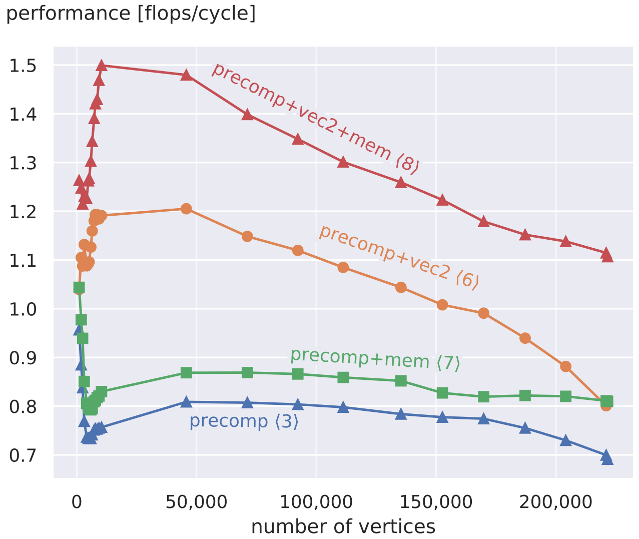

Fig. 8. Performance of optimisations ⟨3⟩, ⟨6⟩, ⟨7⟩ and ⟨8⟩ on the small and big dataset combined.

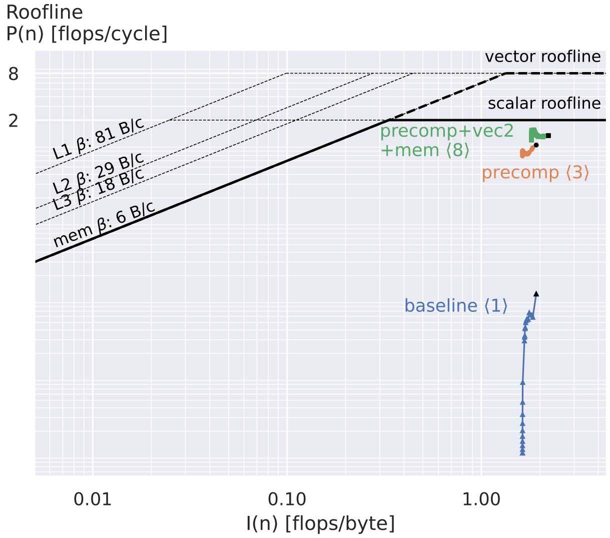

Fig. 10. Roofline plot of optimisation steps ⟨8⟩, ⟨3⟩ and the baseline ⟨1⟩ on the small and big dataset combined. Black dots represent the smallest input n. The rooflines are with respect to scalar / vectorised peak processor performance (2 resp. 8 flops/cycle) and the bandwidths of L1, L2, L3 and memory mentioned in Section 4.1. We assumed each random access has to load a cache line from main memory.

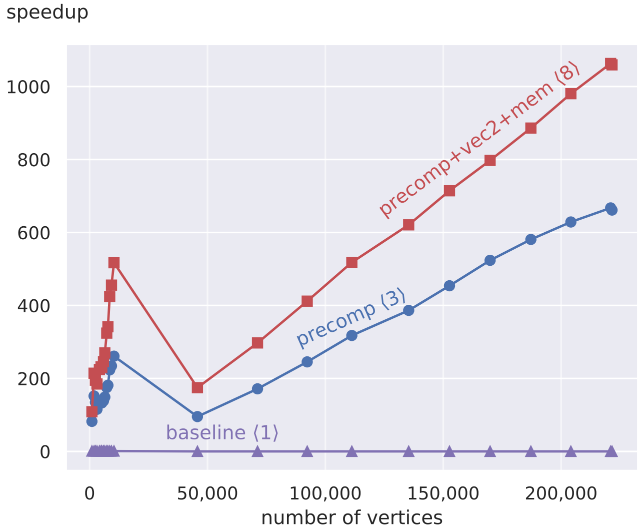

Fig. 9. Relative end-to-end speedup over baseline ⟨1⟩ on the small and big dataset combined

bound region. The last optimisation step achieves around 16.13% of the vectorised peak performance and 64.5% of the scalar peak performance on average on the platform.

End-to-end speedup. To see the runtime gain achieved overall, Fig. 9 displays the relative speedup over the baseline ⟨1⟩ on the small and big dataset merged. On the small dataset, the overall relative speedup lies between 100 to 500× for optimisation step ⟨8⟩. The big dataset appears after the first peak in Fig. 9, after which the relative speedup scales with input size. The peak is there because the different datasets have different average degree when starting from small inputs. Since optimisation step ⟨3⟩ was the last to have performed a major algorithmic optimisation, it is displayed for reference as well in Fig. 9.

5. CONCLUSION

In this paper, we focused on optimising the sequential performance of belief propagation for top-N recommendation. Top-N recommendation plays important role in online shopping, video and streaming platforms. We achieved a 100–1000× speedup on an Intel® Core™ i5-7200U @2.50GHz machine and on average 16.13% of the vectorised peak performance on dataset which does not fit entirely into cache. The speedup scales with input size such that for our greatest input containing more than 200,000 vertices, it exceeds 1000×. For bigger inputs advantages in efficient computation get gradually drowned out by slow memory access. Reducing the memory footprint of data structures at the expense of more computation had shown to have positive impact on speedup in this case. Despite the divisions being introduced by our algorithmic optimisation and the fact that they limit the amount of instruction level parallelism beneficial significantly, this tradeoff suited the type of our input data very well and achieved the greatest relative speedup out of all optimisation steps.

6. CONTRIBUTIONS

Henry. Implemented flop counting in code. Tried out unrolling outer loop over nodes further. Tried out unrolling inner loop over neighbours of a node further. Extended flop counting with e.g. byte counting. Measured graph padding on bipartite graph. Implemented roofline plot. Helped with preparing plots for the presentation, mainly the roofline.

Sarah. Implemented flop counting in code of both algorithms together with Henry. Improved library flop counting. Performed unrolling optimisation of inner loop over neighbours of a node. Mostly prepared the slides on optimisations for the presentation.

Richard. Did the base implementation from scratch in C. Performed unrolling optimisation of innermost 2×2 loops over states and scalar replacement. Implemented precomputation of product optimisation. Tried graph padding optimisation. Helped Henry implementing/improving the roofline plot. Mostly prepared the slides on cost measure, experimental setup. Performed measurements and prepared plots.

Stephanie. Did the base implementation using the libDAI library. Vectorised loop iterating over neighbours of a node. Implemented saving memory optimisation as discussed. Tried out the `__restrict__` keyword and compiler flags. Mostly prepared slides on algorithm and cost measure for the presentation.

7. REFERENCES

[1] E. Ayday, A. Einolghozati, and F. Fekri, "BPRS: Belief Propagation based iterative recommender system," in *2012 IEEE International Symposium on Information Theory Proceedings*, 2012, pp. 1992–1996.

[2] J. Ha, S. Kwon, S. Kim, C. Faloutsos, and S. Park, "Top-N recommendation through belief propagation," in *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, X. Chen, G. Lebanon, H. Wang, and M. J. Zaki, Eds. ACM, 2012, pp. 2343–2346. [Online]. Available: https://doi.org/10.1145/2396761.2398636

[3] J. M. Mooij, "libDAI: A Free and Open Source C++ Library for Discrete Approximate Inference in Graphical Models," *Journal of Machine Learning Research*, vol. 11, pp. 2169–2173, Aug. 2010. [Online]. Available: http://www.jmlr.org/papers/volume11/mooij10a/mooij10a.pdf

[4] Y. Ren, C. Zhang, X. Liu, and X. You, "Efficient early termination schemes for belief-propagation decoding of polar codes," in *2015 IEEE 11th International Conference on ASIC (ASICON)*, 2015, pp. 1–4.

[5] A. Mendiburu, R. Santana, J. A. Lozano, and E. Bengoetxea, "A Parallel Framework for Loopy Belief Propagation," in *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 2843–2850. [Online]. Available: https://doi.org/10.1145/1274000.1274084

[6] L. Zheng, O. Mengshoel, and J. Chong, "Belief Propagation by Message Passing in Junction Trees: Computing Each Message Faster Using GPU Parallelization," 2012. [Online]. Available: https://arxiv.org/abs/1202.3777

[7] "Intel® Core™ i5-7200U Processor Specification," Accessed: 2022-06-13. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/95443/intel-core-i57200u-processor-3m-cache-up-to-3-10-ghz/specifications.html?wapkw=Intel(R)%20Core(TM)%20i5-7200U

[8] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corp., February 2022, Order number: 248966-045. Pages: 67, 761-780. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[9] "Intel Core i5-7200U specifications," Accessed: 2022-06-13. [Online]. Available: https://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5%20i5-7200U.html

[10] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[11] ——, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. [Online]. Available: http://www.cs.virginia.edu/stream/

[12] F. M. Harper and J. A. Konstan, "The MovieLens Datasets: History and Context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, 2016. [Online]. Available: https://doi.org/10.1145/2827872

[13] "MovieLens Latest Datasets," https://grouplens.org/datasets/movielens/latest/, Dataset version: 9/2018. Accessed: 2022-06-13.

[14] "MovieLens 25M Dataset," Released: 12/2019. Accessed: 2022-06-13. [Online]. Available: https://grouplens.org/datasets/movielens/25m/