SPEEDING UP T-SNE FOR 2D VISUALIZATION

G. Fringeli, K. Zhang, M. Nowack, M. Rettenbacher

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

t-Distributed Stochastic Neighbor Embedding (t-SNE) is an embedding technique used to visualize high-dimensional data. We implement a version that is highly optimized for embedding the high-dimensional data into two dimensions, which is the most common use case, and targets common single-threaded processors. Benchmarks show that our optimized version performs up to 28x better than the original implementation supplied by the authors of t-SNE.

1. INTRODUCTION

Motivation. We as human beings are accustomed to the three-dimensional environment we live in. As a result, we are able to produce and perceive visualizations of data with up to three dimensions. For more dimensions, finding intuitive visual representations becomes increasingly hard and even infeasible for most high-dimensional datasets of practical interest.

One solution to alleviate this issue is to map the original high-dimensional datapoints to a two- or three-dimensional space while retaining as much of the structure of the original data as possible. Popular methods for this task include Locally Linear Embedding (LLE) [1], Isomap [2], and t-distributed stochastic neighbor embedding (t-SNE) [3].

Contribution. In a world where the amounts of data we work with are constantly increasing, so is the demand for fast data processing algorithms such as t-SNE. Thus, we develop a performance-optimized version of t-SNE in C that takes advantage of the AVX2 vector instruction extension that is present in state-of-the-art x86-64 processors.

Related work. Besides optimizing the exact t-SNE algorithm with asymptotic runtime $\mathcal{O}(n^2)$, van der Maaten developed an approximate version [4] with time complexity in $\mathcal{O}(n \log n)$. By contrast, our work does not rely on approximations and optimizes the exact formulation.

Another way for speeding-up t-SNE is to take advantage of GPUs [5]. However, this approach relies on a suitable computational hardware, while our work only requires an x86-64 processor with AVX2 vector instruction extension, which are widespread at the time of writing.

2. T-DISTRIBUTED STOCHASTIC NEIGHBOR EMBEDDING

We give a short overview and the definition of the exact t-SNE algorithm, along with the cost measure we used to determine the amount of floating point operations (flops).

Probability model. Assume we are given n points $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^D$ in a high-dimensional space. The aim of t-SNE is to determine corresponding low-dimensional points $\mathbf{y}_1, \dots, \mathbf{y}_n \in \mathbb{R}^d$ that retain as much of the structure of the original data as possible. Usually, one chooses $d = 2$ or $d = 3$ for visualization purposes.

t-SNE defines probability models on pairs of points for both the high- and the low-dimensional spaces, which are based on pairwise Euclidean distances. For any two high-dimensional datapoints $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^D$ one defines a conditional probability by

$$p_{j|i} = \begin{cases} \frac{\exp\left(-\|\mathbf{x}_i - \mathbf{x}_j\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\|\mathbf{x}_i - \mathbf{x}_k\|^2 / 2\sigma_i^2\right)}, & i \neq j \\ 0, & \text{otherwise} \end{cases}$$

and subsequently a pairwise probability by

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}. \tag{1}$$

For a datapoint $\mathbf{x}_i$ the variance σ_i^2 is chosen such that the perplexity of its conditional probability distribution is equal to the value specified by the user. The perplexity is defined as

$$\text{Perp}(p_{1|i}, \dots, p_{n|i}) = 2^{H(p_{1|i}, \dots, p_{n|i})}, \tag{2}$$

where $H(\cdot)$ is the Shannon entropy

$$H(p_{1|i}, \dots, p_{n|i}) = -\sum_j p_{j|i} \log_2 p_{j|i}.$$

In the low-dimensional space the probability of a pair of points $\mathbf{y}_i, \mathbf{y}_j$ is given by

$$q_{ij} = \begin{cases} \frac{\left(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2\right)^{-1}}{\sum_{k \neq \ell} (1 + \|\mathbf{y}_k - \mathbf{y}_\ell\|^2)^{-1}}, & i \neq j \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Loss function. The embeddings $\mathbf{y}_1, \ldots, \mathbf{y}_n$ are determined such that they minimize the Kullback-Leibler divergence between the high- and the low-dimensional probability distributions given by

$$\mathcal{L} = KL(P\|Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}. \tag{4}$$

Algorithm. The full t-SNE algorithm is shown in Algorithm 1. It starts by determining the variances σ_i^2 such that the perplexity in Equation (2) is equal to the perplexity Perp chosen by the user. This is achieved using the bisection method. Subsequently the high-dimensional probabilities p_{ij} are calculated. Next the embeddings are initialized by sampling points from $\mathcal{N}(0, 10^{-4}I)$.

The loss function in Equation (4) is minimised using gradient descent. In each of the T gradient descent iterations the low-dimensional probabilities q_{ij} are calculated based on the current embeddings and the embeddings are updated via a gradient descent step with user-defined learning rate and momentum schedule. The gradient of $\mathcal{L}$ with respect to a given embedding $\mathbf{y}_i$ is given by

$$\frac{\partial \mathcal{L}}{\partial \mathbf{y}_i} = 4 \sum_{j=1}^{n} (p_{ij} - q_{ij})(\mathbf{y}_i - \mathbf{y}_j). \tag{5}$$

We use the adaptive learning rate heuristic proposed in the Python implementation [6] by the authors of [3] which introduces a multiplicative factor for the learning rate for each point as follows

$$\beta_i^{(t)} = \begin{cases} \beta_i^{(t-1)} + 0.2, & \left(\mathbf{y}_i^{(t-1)} - \mathbf{y}_i^{(t-2)}\right) \frac{\partial \mathcal{L}}{\partial \mathbf{y}_i^{(t-1)}} < 0 \\ \beta_i^{(t-1)} \cdot 0.8, & \text{otherwise} \end{cases}$$

with $\beta_i^{(0)} = 1$.

Cost Analysis. Due to the usage of exponentials and logarithms we count basic flops (add, sub, mult, div, min, max), exponentials, and logarithms separately. This results in the following cost function:

$$C(n) = (\text{basic\_flops}(n), \text{exps}(n), \text{logs}(n)).$$

The exact operation count depends on the parameters D, d, T, and k. To get to a single cost measure we consider the situation of embedding hand-written digits from the MNIST dataset [7] to two-dimensional embeddings. In this setting we have $D = 784$ and $d = 2$, and choose $k = 50$ and $T = \{100, 1000\}$. $k = 50$ and $T = 1000$ are the values chosen by the Python implementation. We mainly used $T = 100$ for more practical execution times, but it produces good results too. Under this assumption, calculating the high-dimensional probabilities has cost function

$$C(n) = (1379n^2 - 928n, 50n(n-1), 50n)$$

Algorithm 1: t-SNE

Input: high-dimensional datapoints $\mathbf{x}_1, \ldots, \mathbf{x}_n$, perplexity Perp, number of iterations T, learning rate η, momentum schedule $\alpha(t)$, number of bisection steps k for determining variances σ_i^2

Output: low-dimensional embeddings $\mathbf{y}_1, \ldots, \mathbf{y}_n$

begin

- determine σ_i^2 and calculate p_{ij};
- sample $\mathbf{y}_1^{(0)}, \ldots, \mathbf{y}_n^{(0)} \sim \mathcal{N}(0, 10^{-4}I)$;
- **for** *t = 1* **to** *T* **do**
  - compute q_{ij};
  - update learning rate factors $\beta_i^{(t)}$;
  - update embeddings $\mathbf{y}_i^{(t)} = \mathbf{y}_i^{(t-1)} - \eta\beta_i^{(t)} \frac{\partial \mathcal{L}}{\partial \mathbf{y}_i^{(t-1)}} + \alpha(t)\left(\mathbf{y}_i^{(t-1)} - \mathbf{y}_i^{(t-2)}\right)$
- **end**
- $\mathbf{y}_i = \mathbf{y}_i^{(T)}$;

end

and calculating one iteration of gradient descent has cost function

$$C(n) = (13n^2 + 13n + 2, 0, 0).$$

To further simplify the analysis we reduce the cost measure to a single number by assuming that an exponential takes 19 flops and a logarithm takes 23 flops. These are obtained by manually counting the basic flops in the GNU C Library implementation. Thus, executing t-SNE under this setting requires approximately

$$2329n^2 - 728n + T(13n^2 + 13n + 2)$$

flops. Instantiated for $T = 100$

$$3629n^2 + 572n + 200$$

and $T = 1000$

$$15329n^2 + 12272n + 2000.$$

3. OPTIMIZATION APPROACH

In this section we describe our optimizations. We break down the description into two parts. In the first part we will go through optimizations applied to individual subprocedures required by the t-SNE algorithm. In the second part we concentrate on global optimizations that affect several parts of the algorithm at once. To simplify vectorization using AVX-2, we assume that n is divisible by 4 throughout and data is 32-byte aligned.

Euclidean distances. Calculating the high- and low-dimensional probabilities p_{ij} and q_{ij} requires the calculation of the squared Euclidean distance between each pair of

points in the respective space. For two points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ we have two possibilities for calculating the squared Euclidean distance, either by calculating element-wise differences (variant I)

$$\|\mathbf{x} - \mathbf{y}\|^2 = \sum_{j=1}^{m} (\mathbf{x}_j - \mathbf{y}_j)^2 \tag{6}$$

or by computing the norms and taking the inner product (variant II)

$$\|\mathbf{x} - \mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\mathbf{x}^T\mathbf{y}. \tag{7}$$

If one computes the distance for every pair of a set of n points, which is the case for t-SNE, the second variant has the advantage that the norm of a given point only has to be calculated once and can be re-used in pairs consisting of that point. Moreover, its operation structure is more suitable for applying fused multiply-add (FMA) instructions.

The flop counts of variants I and II are

$$\frac{3}{2}n(n-1)m$$

and

$$2nm + n(n-1)\left(m + \frac{3}{2}\right)$$

respectively. The choice of the most appropriate variant depends on the dimensionality of the points m and the number of points n. Assuming $n \geq 32$ points, we get the following results. For $m \leq 3$, variant I requires less flops than variant II. For $m > 3$ it is the other way around. Thus, we use variant I for the low-dimensional embeddings ($m = 2$) and variant II for the high-dimensional points.

Besides the choice of algorithm for calculating the squared Euclidean distance we apply optimizations as follows. The baseline implementation consists of a triple loop where the two outer loops are over points and the inner loop is over the point dimensions. The resulting distances are stored in a $n \times n$ matrix, which is later used to calculated the low- and high-dimensional probabilities.

We start with the low-dimensional case. As the resulting distance matrix is symmetric and the calling function only requires the upper triangular elements to calculate the low-dimensional probabilities we do not fill the lower-triangular elements. This eliminates unused data accesses in the order of $\mathcal{O}(n^2)$. As the point dimensions are fixed to $m = 2$ for the low-dimensional case we remove the innermost loop and calculate the squared norm directly.

Finally, we vectorize our function. In order to increase spatial locality and instruction level parallelism (ILP) we loop over blocks of 4×8 points and completely unroll the calculations of the pairwise distances within such a block.

For the high-dimensional case, we first modify the baseline to use variant II for calculating the distances. The resulting function initially calculates the squared norm of each point and stores them in a temporary array. Subsequently, the distances are calculated according to Equation (7). Both parts consist of nested loops where the innermost loop calculates a sum by iterating over the point dimensions. We unroll both innermost loops and add additional accumulators to increase ILP.

Next we take advantage of vector instructions. For this, we move from unrolling over the point dimensions to unrolling over the points. That is, we unroll the outermost loop for both the calculation of the norms and the distances. For the norms we unroll by 8, i.e. we calculate the norms of 8 points in parallel. For the loop calculating the final distances we work with blocks of 4×4 points and use 16 accumulators sum up the inner product for each pair of points. Both loops primarily consist of multiplications followed by additions which we implement with FMA instructions.

High-dimensional joint probabilities. As per Algorithm 1, before calculating the low-dimensional embeddings, we need to obtain the high-dimensional joint probabilities p_{ij}. This procedure takes the input X to first obtain the pairwise euclidean distance in the high-dimensional space. The distance is then used to determine the conditional probabilities. While calculating the conditional probabilities, we need to adjust the bandwidth σ_i of the Gaussian kernel for each row of X. This is done by using bisection method to match the log perplexity to the target one.

Here, we need to access the entire distance matrix. However, since the joint probabilities matrix is symmetric, we only need to calculate half of the matrix. The baseline implementation already leverages this symmetry and implements the aforementioned steps.

Before jumping into optimizations, we profiled the entire function and noticed that almost the entire CPU time is spent on calculating the log perplexity in each bisection step. Thus, we focus our attention on this particular subprocedure. First, we unroll the inner loop by a factor of 2, 4, 8 respectively, using the corresponding number of accumulators. Next we noticed that the memory access pattern of the log perplexity is row-wise, so we already have good spatial locality and no blocking is employed here. Finally, we performed vectorization with 1, 2, 4, 8 accumulators and make use of FMA instructions where possible.

One peculiarity during the optimization is that t-SNE requires the diagonal entries of the joint probabilities matrix to be 0 instead of 1 (e^0). Initially, we want to avoid the additional if-statement (and the branch prediction penalties) by leaving the 1s as is, and subtract the affected terms at the end. However, this proves to be problematic for the normalizer of the conditional probability (the denominator) because most of the numerators of the conditional probability is small, and adding the 1s results in the fractional parts being ignored. Thus, subtracting the 1s in the end just yields 0 for the denominator. We mitigate this issue by re-

introducing the if conditions, but using ternary operators to utilize conditional moves [8] in the scalar versions and conditional masks in vectorized ones.

Zooming out to the entire joint probabilities function: After vectorization, only a fraction of the time is still spent on calculating log perplexity, and the baseline pairwise distance becomes the bottleneck. But thanks to Equation (7) and vectorization of the Euclidean distances, now more than half of the time is spent on log perplexity. Due to the Read-After-Write dependency of the bisection method, no optimization can be made between each step. However, we tried to unroll the outer loop and adding accumulators by reordering the for loops and inlining the log perplexity function, but this produces even worse performance, most likely due to register spilling.

As we are using a different variant of calculating the distances, the flop count changes to

$$C(n) = (988.5n^2 + 1030.5n, 50n(n-1), 50n).$$

Furthermore, we make use of Agner's vector class library [9] for a vectorized $\exp$ function, which takes 36 basic flops. Thus, the flop count of calculating the high-dimensional joint probabilities is now approximately

$$2788.5n^2 + 380.5n$$

instead of

$$2329n^2 - 728n.$$

Low-dimensional probabilities. This function takes the squared Euclidean distances of the embeddings as input and outputs both the low-dimensional probabilities as in Equation (3) as well as only the numerators of the probabilities.

The baseline implementation first calculates each numerator and writes them to a temporary matrix. Since the matrix is symmetric, only numerators of the upper triangular half are calculated and then written to both locations. During this process the sum of all entries is accumulated which then yields the denominator. Note that the denominator is identical for all entries. Next, the baseline implementation divides all numerators by the denominator and writes the resulting probability to the output matrix. Since the matrix is again symmetric, only probabilities of the upper triangular half are calculated and then written to both locations.

As both resulting matrices are symmetric and the calling function only requires the upper triangular elements to calculate the gradient descent, the first optimization is not to write the lower triangular half. This results in a speedup much higher than just a factor of two because the lower triangular accesses are in column-order which has no spatial locality. However, this also turned out to be a somewhat futile optimization because optimized versions of the calling function do in fact rely on the lower triangular half being present.

We additionally applied loop unrolling, accumulator variables, and vectorization. However, all these optimizations result only in minor performance improvements. After careful profiling it turns out that, while the implementation is not memory bound because the number of transferred bytes exceeds the memory bandwidth, the raw latency of memory accesses is nevertheless a crippling bottleneck. This bottleneck cannot be resolved in this function alone.

Gradient descent. As shown in Algorithm 1, the gradient descent function calculates the probabilities q_{ij}, updates the learning rate factors $\beta_i^{(t)}$ and calculates the updated embeddings $\mathbf{y}_i^{(t)}$. We omit the computation of q_{ij}, as it is optimized separately.

The baseline implementation first fills a temporary matrix to exploit its symmetry and calculate only half of the matrix. Using that matrix it then calculates the gradient with respect to the embeddings $\mathbf{y}_i^{(t)}$, followed by updating the learning rate factors. We then update the values of $\mathbf{y}_i^{(t)}$ and center each dimension around 0.

Given the fixed output-dimension, we first unroll and use two separate accumulators for each calculation that loops over the two-dimensional embeddings $\mathbf{y}_i^{(t)}$.

Profiling the code shows that calculating our temporary symmetric matrix is the current bottleneck due to having to write and read n^2 values and also causing many cache-misses due to accessing half the matrix in column-order. We merge the temporary matrix with the calculations for the y-gradient. We are now calculating double the amount of values than before with respect to the matrix, but do not store or read those values from memory anymore.

To make further use of temporal locality, we merge our loops and now have two major loops: One for calculating all the new values and updating $\beta_i^{(t)}$, and one for updating $\mathbf{y}_i^{(t)}$ with the centered values.

To improve ILP we unroll the loops again, use separate accumulators and remove the gradient-matrix storage. The bottleneck is now the double for-loop where we calculate the gradient of y. Further unrolling the inner loop did not have a positive effect on performance.

Lastly, we vectorize the above version of gradient descent. When handling our embeddings $\mathbf{y}_i^{(t)}$, we decided to load the values such that we have a vector for each of the two columns. This makes computations simpler, but requires shuffles when loading and storing the embeddings.

Combined algorithm. Starting from the previously described subprocedures of the t-SNE algorithms that have been optimized in isolation, we now consider the full algorithm.

In the baseline, we determine and store the full low-dimensional Euclidean distance matrix before moving to

calculating the low-dimensional probabilities. We are able to combine both subprocedures into a single one, which eliminates temporary storage of the Euclidean distances in memory.

Next, note that calculating the low-dimensional probabilities in Equation (3) is a two-part process, consisting of calculating the numerators $\left(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2\right)^{-1}$ for all pairs (i, j) and a subsequent normalization. Directly normalizing after calculating the numerators results in accessing all n^2 unnormalized probabilities again. To prevent these additional memory accesses we move the normalization to the gradient descent step in Equation (5), where the same data has to be accessed anyways. We denote this implementation by *opt1*.

Encompassing gradient descent. Profiling the previous work shows that the bottleneck of the whole gradient descent part is the latency of memory accesses. This is not a major surprise considering the baseline implementation uses no less than eight matrices as temporary storage. Hence, we reevaluate the gradient descent not from the perspective of individual functions but as an encompassing function that should be optimized to avoid writing temporary results to main memory. We begin this work from the completely unoptimized baseline implementation.

As shown in Algorithm 1, the gradient descent function calculates the probabilities q_{ij}, updates the learning rate factors $\beta_i^{(t)}$ and calculates the updated embeddings $\mathbf{y}_i^{(t)}$.

In order to calculate the low-dimensional probabilities q_{ij} as in Equation (3) the baseline implementation first calculates the squared Euclidean distances between all points in the previous embedding and writes them to a temporary matrix. Then, the baseline implementation calculates q_{ij} which is split into two steps. First, the numerators of q_{ij} are calculated and written to another temporary matrix, and second, the final q_{ij} are calculated by dividing each numerator by the denominator which is the sum over all numerators. Note that this denominator is identical for all entries. Finally, q_{ij} is written to yet another temporary matrix.

However, since the squared distances are only used to calculate the numerators, it is entirely pointless to ever write them into a matrix. Thus, the first optimization is to merge calculating the squared distances and calculating the numerators into a single loop. This avoids writing and reading a full n^2 matrix.

A similar problem arises with the matrix that holds q_{ij} as these values are only used to calculate the gradient. Thus, the second optimization is to merge calculating q_{ij} from the numerators and calculating the gradient into a single loop. This avoids writing and reading a full n^2 matrix.

Writing the numerators to a temporary matrix appears to make sense at a first glance because they are used twice in a manner that does not allow merging them into a single loop as all numerators have to be divided by a denominator that can only be calculated after having calculated all numerators. Calculating the numerators from scratch costs $3d + 2$ flops each, including a division which is not cheap. Nevertheless, we decide to avoid writing the numerators to a temporary matrix because as evinced from the previous work memory accesses are a bigger problem than numerical computations. Initially, this change increases runtime, but when combined with further optimizations it performs much better. Two details work in our favor here. First, the numerators are symmetric which allows calculating only half of them to get the denominator. Second, the gradient anyway requires the distances between individual coordinates and therefore we get this operation for free while calculating the Euclidean distances, although we do not apply this optimization until later when the index of the reused coordinates can be easily hardcoded.

When calculating the gradient, the baseline implementation fills a temporary matrix to exploit symmetry. However, due to writing and reading n^2 values and causing many cache misses in the process due to accessing the lower triangular half of the matrix in column-order this temporary matrix actually increases runtime instead of decreasing it. Thus, the next optimization is to ignore the symmetry, calculate all n^2 temporary values and immediately use them instead of first writing them to a temporary matrix.

The previous optimization introduces an if-statement inside the inner loop to ensure the temporary value corresponding to a diagonal matrix entry is zero. These values can be handled implicitly because their calculation involves a multiplication by the distance, which is naturally zero in case of diagonal entries because the two points are one and the same. Thus, we can remove the if-statement without replacement.

By now we have removed four out of eight temporary matrices. Can we do better? Storing the joint probabilities p_{ij} is non-negotiable because they are very expensive to compute and used in every iteration of the gradient descent. Similarly, storing the delta of the previous embeddings $\mathbf{y}_i^{(t-1)} - \mathbf{y}_i^{(t-2)}$ and the previous learning rate factors $\beta_i^{(t-1)}$ are non-negotiable because they are used in every iteration of the gradient descent and calculating them from scratch requires repeating all previous iterations. The one remaining matrix temporarily stores the gradient. In principle, this matrix can be avoided too, but this does not turn out to be beneficial because this matrix has only dimension $n \times d$ unlike the other removed matrices which have dimension $n \times n$. In fact, anything past the calculation of the gradient is negligible due to its linear runtime which is completely dominated by the quadratic runtime of the prior calculations.

Until this point, the implementation took the dimension d of the embeddings $\mathbf{y}_i$ as an input parameter. However, for practical purposes only $d = 2$ or possibly $d = 3$ make

sense. By assuming d to be a fixed parameter additional optimizations can be performed. Previously, distances between points were computed like this:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < d; k++)
      double dist = Y[i*d+k] - Y[j*d+k];
```

However, with fixed $d = 2$ these loops can be rewritten to remove aliasing so that the first memory accesses are reduced by a factor of n.

```
for (int i = 0; i < n; i++)
  double Y_i1 = Y[i * d];
  double Y_i2 = Y[i * d + 1];
  for (int j = 0; j < n; j++)
    double dist_1 = Y_i1 - Y[j*d];
    double dist_2 = Y_i2 - Y[j*d+1];
```

The innermost loop during the calculation of the gradient is executed n^2d times. The baseline implementation only executes three flops in one iteration, but after the previous optimizations much more calculations are performed here. However, the new calculations are unnecessarily repeated because they calculate only n^2 unique results. With d being fixed the innermost loop can be unrolled by d which reduces the flop count and therefore the runtime by almost a factor of d.

As profiling reveals no further distinct bottlenecks, vectorization is applied next. Vectorizing the calculation of the gradient is straightforward. All scalar operations can be replaced by the corresponding vector intrinsic without any special handling at all. Vectorizing the calculation of the denominator is slightly more complex because only the upper triangular half is processed which requires handling some leftover entries by scalar operations. Furthermore, as the denominator is a sum over all entries the usage of accumulator variables becomes necessary.

Finally, we unroll the vectorized loops to increase ILP. This did not increase performance at all because apparently the compiler was already unrolling loops on its own accord.

The flop count changes several times as the various optimizations are applied. The flop count for one iteration of the most optimized gradient descent is

$$20.5n^2 + 15.5n + 6$$

compared to the baseline of

$$13n^2 + 13n + 2.$$

Despite the significantly higher flop count the function is much faster than the baseline.

4. EXPERIMENTAL RESULTS

In this section we give an overview of the setup we used, then briefly show the optimization results of each of the individual procedures, followed by the optimizations performed on the whole algorithm and the final performance improvement results.

Experimental setup. We use version 10.3.0 of the gcc compiler with the relevant flags `-O3, -march=native, -m64, -fno-tree-vectorize -std=c++17`. We have also tried the icc compiler, but it yielded no noteworthy speedup.

The platform is an Intel® Core™ i5-6200U CPU at frequency 2.3 GHz. The cache sizes are L1: 32KB, L2: 256KB, and L3: 3MB

We used the MNIST dataset [7] as input. The number of datapoints used ranges from 2^2 up to 2^{12}. To ensure deterministic results, we generated an initial random embedding once and used it as input too.

Euclidean distances. The baseline implementation for the low-dimensional distances reaches a performance of up to 0.5 flops / cycle for small inputs and decays to 0.15 flops / cycle for more than $n = 64$ points. This decay is prevented by not filling the lower triangular elements. Eliminating the innermost loop gives a speedup of 3x. The final vectorization with 4×8 blocks reaches peak performance of 5 flops / cycle for $n = 128$ and decays towards 3 flops / cycle for larger input sizes.

For the high-dimensional distances switching from variant I to variant II does not considerably affect the runtime. Unrolling the innermost loop and adding accumulators gives a 3x speedup. With vectorization and the 4×4 blocking we get a further speedup of 6x with a peak performance of 8.5 flops / cycle. We finally validate our choice for variant II by comparing both variants in the fully optimized vectorized version. For 128 points and more variant II, i.e. computing norms and inner product, is 23% faster than variant I. Below that margin the overhead for calculating and storing the norms is not outweighed by the reduction of computation. However, such small sample sizes are not of importance in practice.

High-dimensional joint probabilities. Unrolling by 8 with 8 accumulators gives the best speedup of 1.7x. This likely due to most operations being additions and multiplications (including the exp library function), and using a Skylake processor which has two ports for addition and multiplication, where each operation takes 4 cycles. The actual performance is around 1.2 flops / cycle, whereas the baseline is around 0.8 flops / cycle for all input sizes. Vectorizing with 4 accumulators gives the best speedup of 7.1x comparing to baseline. As seen in Figure 1, the actual performance varies with input size: There is little improvement when $n < 2^3$, but with larger n, the code benefits from the

instruction level parallelism and increases the performance to around 6 flops / cycle. Adding more accumulators decreases the performance due to register spill.

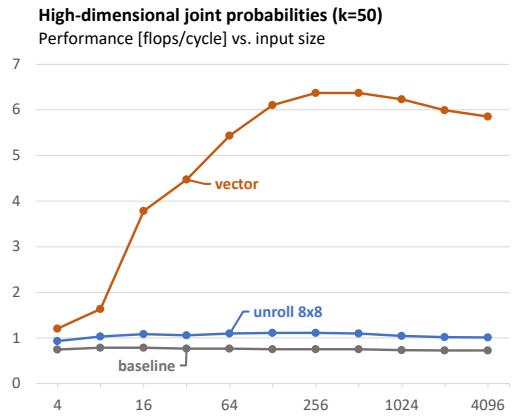

Fig. 1. Performance of three implementations of high-dimensional joint probabilities, with $k = 50$ bisection steps.

Low-dimensional probabilities. The baseline has a performance of 0.1 flops / cycle. Not writing the lower triangular half increases performance to 0.6 flops / cycle. Beyond that, further optimizations are futile because memory accesses are dominating. A fully vectorized version yields an additional speedup of barely 1.05x.

Gradient descent. Initial unrolling gives only a minor speedup. Removing the temporary matrix gives the largest speedup of 5.7x. Merging the loops together does not give any speedup by itself, but further loop unrolling gives an additional speedup of 1.2x. Vectorization adds a speedup of 1.3x, which results in a total speedup of 9x for the gradient descent procedure.

Encompassing gradient descent. Since this function optimizes memory accesses, its speedup increases as the input size grows. At $n = 2^{12}$, merely eliminating the four temporary matrices without any other optimizations already results in a speedup of 4.6x. The most optimized version evens out at a speedup of over 30x for $n \geq 2^{11}$ compared to the baseline. The peak performance is 5.3 flops / cycle at $n = 2^{12}$.

Complete algorithm. We start with a performance analysis on the full t-SNE algorithm for $n = 2^2$ up to $n = 2^{12}$ points. The analysis includes the baseline implementation, a scalar and a vectorized naive combination of the subprocedures optimized in isolation, and two the implementations that are optimized as a whole. Naive in this case means that the separately optimized functions are only used to replace the corresponding functions in the baseline without considering optimizations across them. The results of this analysis are shown in Figure 2. In the following analysis the implementations previously described as combined algorithm and encompassing gradient descent are now referred to as *opt1* and *opt2* respectively.

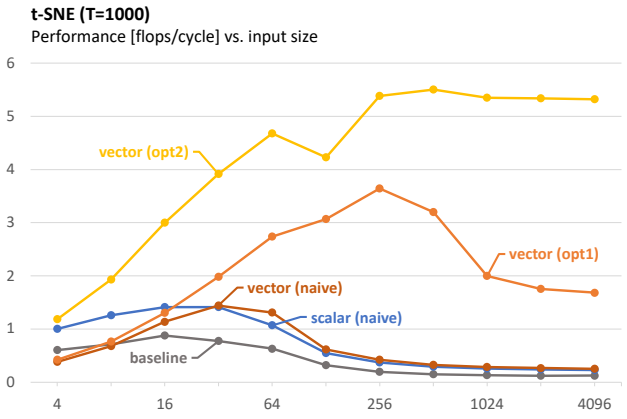

Fig. 2. Performance of t-SNE implementations for $T = 1000$ iterations.

The baseline implementation reaches a peak performance of 0.85 flops / cycle for $n = 16$ points. However, for larger inputs sizes the performance quickly decays towards 0.25 flops / cycle. The naive combinations of optimized scalar and vectorized functions both achieve peak performances of 1.5 flops / cycle. The almost identical behaviour nicely illustrates how much these versions are dominated by the latency of memory accesses. Compared to the baseline, the strong performance decay for larger input sizes is slightly delayed.

The two globally optimized implementations *opt1* and *opt2*, which primarily deal with reducing memory accesses, are able to alleviate the aforementioned performance decay. With *opt1* the peak performance moves towards 4 flops / cycle and then falls below 2 flops / cycle for large input sizes. For *opt2* the decay is completely eliminated and we observe a performance around 5.5 flops / cycle for $n \geq 256$ datapoints. A general performance decay can be observed at $n = 128$, which is most striking for *opt2*, because the $n \times n$ matrix no longer fits into the L1 cache. Similarly, *opt1* has a striking performance decay after $n = 512$ when the $n \times n$ matrix no longer fits into the L3 cache.

As explained when describing our optimizations, the flop count varies between the five implementations included in the performance analysis. This implies that the performance improvements do not directly translate to runtime improvements which is of primary importance in practice. Thus, we separately analyze the speedup of the optimized implementations with respect to the baseline. The results are shown in Figure 3.

The results from the performance analysis mostly translate to the speedups. A notable difference, however, can be observed with *opt1* and *opt2*. While the performance of

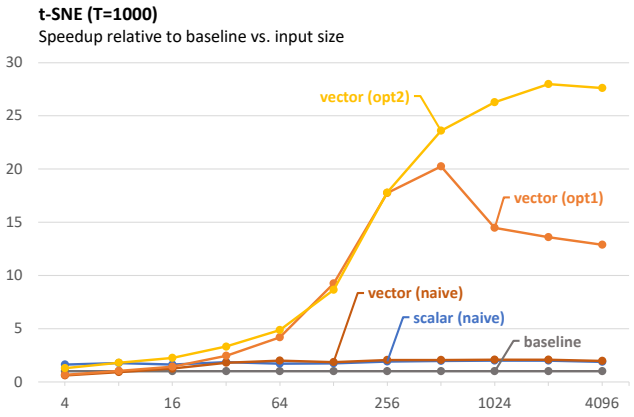

Fig. 3. Speedup of t-SNE implementations relative to baseline for $T = 1000$ iterations.

opt2 is 1-2 flops / cycle higher than the one of *opt1* for up to $n = 256$ datapoints, the runtime of both implementations are nearly identical in this range of inputs. Still, *opt2* is able to maintain its speedup relative to the baseline beyond this point while the speedup decreases for *opt1*. To conclude we are able to reach a speedup of around 28x for input sizes $n \geq 1024$.

Roofline analysis. We proceed to create the roofline plot for the complete algorithm. The memory transfer (in bytes) is estimated by multiplying the last-level cache misses with the cache line size of 64B. The cache is warm with the input data (if it fits inside the cache). Here we use a smaller $T = 100$ (instead of 1000) on a smaller set of data (up until 2^{11} instead of 2^{12}), such that the cache simulator can finish within reasonable time. The result is shown in Figure 4.

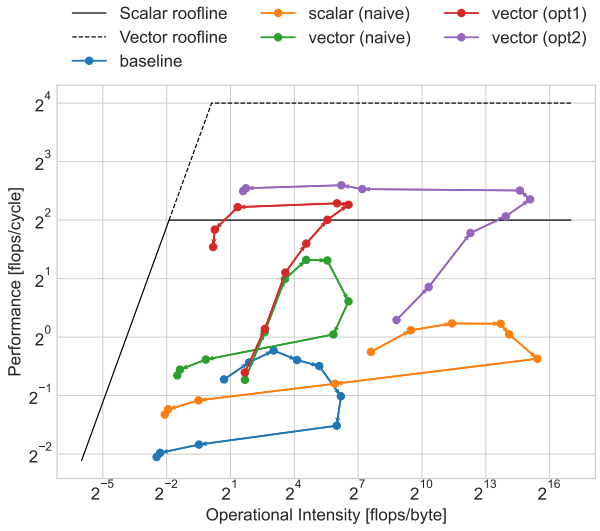

Fig. 4. Roofline plot of various t-SNE implementations. The arrows indicate data sizes from $n = 2^2$ to 2^{11}.

It can be seen that all implementations have the largest operational intensity when n is around 2^6 and 2^7, which is roughly the size of L3 cache (3MB) considering all the intermediate variables (a little more than four $n \times n$ matrices). After this point, the operational intensity drops dramatically due to cache misses and hence increased memory transfers. As the data size increases, the performance increase for the more tightly integrated implementations (*opt1* and *opt2*) is sustained longer before it starts to drop, with *opt2* still giving minor performance increase until the last two data sizes. It can be seen that the computations are mostly compute-bound. Due to the Read-After-Write dependencies of the intermediate variables, as well as the number of such variables (before register spill happens due to multiple accumulators), there is still some gap to the maximum achievable performance.

5. CONCLUSIONS

In this project, we implemented highly-optimized versions of t-SNE for embedding the high-dimensional data into two dimensions. We applied a multitude of optimization techniques: mathematical transformations to reduce the flop count, loop unrolling with multiple accumulators, moving if-statements outside of inner loops, vectorization, and various cache and memory optimizations including blocking, removing unnecessary intermediate variables and removing memory aliasing. Naively combining the individually optimized subprocedures merely gives speedup at most 2.1x compared to the baseline. Only after more tightly combining the subprocedures a more tangible speedup of up to 20x was achieved. Since the latency of memory accesses remained the dominating bottleneck in the gradient descent despite good locality, we resorted to an unusual optimization technique and repeated the same calculations in different parts of the algorithm instead of caching the results to avoid writing and reading from memory whenever possible. While this significantly increased the flop count, it was nevertheless only after this optimization that the implementation could maintain a high speedup of 28x for large input sizes. Although having several independent functions is considered clean coding style, we see that the performance can suffer significantly as a consequence.

6. CONTRIBUTIONS OF TEAM MEMBERS

Gabriel. Optimized the calculation of Euclidean distances for both low- and high-dimensional spaces, including loop unrolling, multiple accumulators, and vectorization. Worked on combining separately optimized subprocedures.

Kaishuo. Optimized the calculation of high-dimensional joint probabilities with loop unrolling, multiple accumulators and vectorization. Worked on roofline plots with Manuel.

Manuel. Optimized low-dimensional probabilities. Optimized encompassing gradient descent. Derived flop count formulas.
Marc. Focused on optimizing the gradient descent function, except the low-dimensional probabilities subprocedure. Unrolled, used multiple accumulators, merged loops, removed superfluous matrices and vectorized the gradient descent. Gradient descent performance plot. Compiler flags.

7. REFERENCES

[1] Sam T. Roweis and Lawrence K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[2] JB Tenenbaum, V de Silva, and JC Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science (New York, N.Y.)*, vol. 290, no. 5500, pp. 2319—2323, December 2000.

[3] Laurens Van Der Maaten and Geoffrey Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, pp. 2579–2625, 11 2008.

[4] Laurens Van Der Maaten, "Accelerating t-sne using tree-based algorithms," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3221–3245, Jan. 2014.

[5] David M. Chan, Roshan Rao, Forrest Huang, and John F. Canny, "T-sne-cuda: Gpu-accelerated t-sne and its applications to modern data," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 330–338.

[6] Laurens Van Der Maaten, "t-sne python implementation," `https://lvdmaaten.github.io/tsne/`, 2017.

[7] Yann LeCun and Corinna Cortes, "MNIST handwritten digit database," 2010.

[8] Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, chapter 5, Addison-Wesley Publishing Company, USA, 3rd edition, 2016.

[9] Agner Fog, "Vector class library," `https://github.com/vectorclass/version2`, 2021.