OPTIMIZED T-SNE

Shengze Jin, Jiale Chen, Muyu Li, Levin Moser

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

In this paper, we present an optimized version of t-distributed Stochastic Neighbor Embedding (t-SNE) [1] for data visualization. We implemented the exact $\mathcal{O}\left(N^2\right)$ version for arbitrary output dimensions. We applied multiple single-core optimization techniques such as reducing the operation numbers, improving cache locality and vectorization. We achieved a speedup of 20-26x compared to the state-of-the-art C/C++ implementation[1].

1. INTRODUCTION

Motivation. During the last decade, many data that are collected lie in very high-dimensional spaces, for example images or word embeddings. Making sense out of the vast amount of high-dimensional data has become a crucial part of any data analysis task. Data visualization is an easy approach to this problem.

t-SNE is one of the mostly used algorithms for visualizing data. It reduces the data to a low-dimensional representation (usually 2D or 3D space) such that the data can be interpreted more easily. This low-dimensional representation is found by first fitting a Gaussian distribution over all data points and then representing this high dimensional Gaussian as a low-dimensional Student's t-distribution. To find a good low-dimensional distribution, the Kullback-Leibler divergence between the two representations is minimized using gradient descent.

However, t-SNE is very slow compared to other dimension reduction methods such as PCA, making it hard to use in large datasets. And fast implementations are very hard and expensive to get because the data locality is bad when the dataset is large (memory bound).

Contribution. The algorithm performs many independent computations and is thus well suited for parallelization and the usage of GPUs. However, in this paper we focus on single-core optimizations for Intel's x86 architectures.

Our implementation does not limit the input or output dimension, or the number of data points in any way, but we focus on the common use case with 1000 input dimensions, 2 or 3 output dimensions, and around 10K sample points for our optimization. We only focus on computational optimization and we do not apply any algorithmic optimizations such as adaptive learning rate or early stopping.

Related work. The algorithm proposed in the authors original paper [1] has a computational complexity of $\mathcal{O}\left(N^2\right)$ where N is the input dimension. There are multiple implementations available in various programming languages listed and maintained by the authors [2].

Especially for large datasets the quadratic runtime of the original t-SNE algorithm leads to a problematic runtime. The same authors therefore proposed a tree-based approximation of t-SNE [2] and has a complexity of $\mathcal{O}\left(N \log N\right)$.

Additionally, David Chan et. al [3] implemented t-SNE for multi-core and used the GPU to accelerate the algorithm.

It is noteworthy that the GPU accelerated implementations of the exact algorithm outperform the approximated algorithm. While the approximated version of t-SNE might have a better computational complexity, the tree-based data-structure that it uses make it harder to implement optimizations. In this paper we thus focus on the original exact algorithm.

2. BACKGROUND ON THE T-SNE ALGORITHM

In this section we will discuss the t-SNE algorithm and our implementation. At the end of this section we will also discuss the cost functions and the complexities of the algorithm.

Dimension reduction. Dimension reduction is to find a low-dimensional representation $\mathcal{Y}$ of the high-dimensional input data $\mathcal{X}$. In other words, it maps the input points $\mathbf{x}_i$ to a lower-dimensional representation $\mathbf{y}_i$. The goal is to find such a representation that preserves most of the local structure in $\mathcal{X}$ as well as revealing global structures such as clusters.

t-SNE algorithm. The first step in the t-SNE algorithm is to compute the squared Euclidean distances between all input data points. In the next step we compute the so-called pairwise affinities $p_{i|j}$. These values express the similarity

[1] https://github.com/lvdmaaten/bhtsne/

[2] https://lvdmaaten.github.io/tsne/

Algorithm 1: t-SNE Algorithm

Data: high dimensional data $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\} \subseteq \mathbb{R}^{D_{\text{in}}}$, target perplexity P, gradient descent iterations T
Result: low dimensional data $\mathcal{Y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_N\} \subseteq \mathbb{R}^{D_{\text{out}}}$

Sample initial $\mathcal{Y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$ from Gaussian distribution $\mathcal{N}\left(0, 10^{-4}\right)$; Initialize $\mathcal{U} = \{\mathbf{u}_1, \ldots, \mathbf{u}_N\}$ to 0;

Part 1 (Squared Euclidean Distances): $\forall i, j = 1..N, \quad d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|^2$;

Part 2 (High-Dimensional Affinities):
for $i = 1..N$ **do**
  initialize $\beta_i = -\frac{1}{\max_j d_{ij}}$;
  repeat
    $\forall j = 1..N,\ j \neq i, \quad p_{j|i} = \frac{\exp(\beta_i d_{ij})}{\sum_{k \neq i} \exp(\beta_i d_{ik})}$; $H_i = -\sum_j p_{j|i} \log_2 p_{j|i}$; Update β_i using binary search;
  until $H_i \approx \log_2 P$;
$\forall i, j = 1..N, \quad p_{ij} = \left(p_{j|i} + p_{i|j}\right) \cdot (2N)^{-1}$;

for $t = 1..T$ **do**
  Part 3 (Low-Dimensional Affinities): $\forall i, j = 1..N,\ i \neq j, \quad t_{ij} = \left(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2\right)^{-1}$; $S = \sum_{i \neq j} t_{ij}$;
  Part 4 (Gradient Descent):
  $\forall i = 1..N, \quad \mathbf{g}_i = 4 \sum_j \left(p_{ij} - t_{ij} S^{-1}\right) (\mathbf{y}_i - \mathbf{y}_j)\, t_{ij}$; $\forall i = 1..N, \quad \mathbf{u}_i = -\eta \mathbf{g}_i + \alpha \mathbf{u}_i, \quad \mathbf{y}_i = \mathbf{y}_i + \mathbf{u}_i$;

between data points $\mathbf{x}_i$ and $\mathbf{x}_j$ by converting their Euclidean distance into conditional Gaussian probabilities that represent their similarity. Once we obtain these conditional probabilities (i.e. the pairwise affinities) we want to compute a joint probability distribution p_{ij}.

The next step is to find a corresponding joint distribution q_{ij} over the mapped data points $\mathbf{y}_i$ in the low-dimensional space $\mathbb{R}^{D_{\text{out}}}$. To do so we also first compute the squared Euclidean distances. On the contrary to the high-dimensional space, we do not convert these pairwise distances into a Gaussian distribution, but we use a Student's t-distribution. This distribution has a much heavier tail which allows us to better model moderate distances in the mapped space.

The goal of the algorithm now is to minimize the difference between the two distribution p_{ij} and q_{ij}. This is done by running gradient descent with the Kullback-Leibler divergence as the objective function to be minimized.

Modifications. Algorithm 1 shows the algorithm as we implemented it in our baseline version. Our version of the algorithm applied a few minor modifications that helped us with optimization later. Any further optimization techniques that we applied did not change this algorithm and it very closely follows the algorithm that was proposed and implemented by the authors. To make it easier to distribute the work, we divided the algorithm into 4 parts.

Cost Analysis. The time complexity of the algorithm is $\mathcal{O}(N^2)$ which comes from calculating the pairwise affinities for each pair of data points.

The cost function for the baseline version is defined as $Cost\,(N, D_{\text{in}}, D_{\text{out}}, T) = (\#add, \#mul, \#div, \#exp, \#log)$. Table 1 in the Appendix shows the leading terms of the cost of baseline. Many optimizations that we are applying aim to reduce the number of flops. The reduced numbers will be discussed in the next section.

3. OPTIMIZATION TECHNIQUES

In this section we discuss the different optimization techniques that we applied to the algorithm. We will present our optimizations step-by-step for the 4 parts of the algorithm.

Part 1 (Squared Euclidean Distances).

The first part computes the Euclidean distance for each pair of high-dimensional data points and store the results in a $N \times N$ matrix. We first reduce the number of flops and then apply blocking.

Reducing flops. The formulas below show 2 equivalent computations of the squared Euclidean distances.

$$d_{ij} = (\mathbf{x}_i - \mathbf{x}_j)^{\mathrm{T}} (\mathbf{x}_i - \mathbf{x}_j) \tag{1}$$

$$d_{ij} = \mathbf{x}_i^{\mathrm{T}} \mathbf{x}_i - 2\mathbf{x}_i^{\mathrm{T}} \mathbf{x}_j + \mathbf{x}_j^{\mathrm{T}} \mathbf{x}_j \tag{2}$$

The first one is more straightforward and is used in baseline. The second one is derived by applying linear algebra to the first one. In our second approach, we can pre-compute $\|\mathbf{x}_i\|^2 = \mathbf{x}_i^{\mathrm{T}} \mathbf{x}_i,\ \forall i = 1..N$. This allows us to reuse the

squared norms of $\mathbf{x}_i$ and $\mathbf{x}_j$ for any pair of Euclidean distance d_{ij}. The reuse of these norms reduces the amount of flops from $(3D_{\text{in}} - 1)\frac{N(N-1)}{2}$ for the first approach to $(2D_{\text{in}} + 2)\frac{N(N-1)}{2} + (2D_{\text{in}} - 1)N$, where D_{in} is the dimension of $\mathbf{x}_i$. This reduction of flops is especially significant if D_{in} is large. Another benefit is it has balanced additions and multiplications, so that they can be fully fused to FMA instructions and has the potential to achieve peak hardware performance. Therefore, we use this approach in Part 1. However, for $D_{\text{in}} \leq 3$ the first approach produces less flops and thus is used in Part 3 to compute the t matrix because the output dimension D_{out} is typically 2 or 3.

Blocking. Despite the flops is decreased by nearly 33% compared to the baseline, we got less than 1.2x speedup only with this numerical optimization. We concluded that this part is memory bound. The bottle neck is clearly the dot product $\mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j$, which has total complexity of $\mathcal{O}\left(D_{\text{in}}N^2\right)$. Because D_{in} is large, $\mathcal{X}$ cannot fit into cache, and for each pair of sample points, at least one of them needs to be loaded from memory. Reducing the flops itself has little help because it is bounded by memory bandwidth.

We further did blocking to improve data locality and it was very effective. We used 2 levels of blocking, one 8×2 blocking for L1D cache, and another 128×128 blocking for L2 cache. In each L1D level block, we load 8 samples $\mathbf{x}_i$ to $\mathbf{x}_{i+7}$, and 2 samples $\mathbf{x}_j$ and $\mathbf{x}_{j+1}$. we can compute $2 \times 8 = 16$ dot products with these $2 + 8 = 10$ samples. When we compute for next block, we can load the 8 samples $\mathbf{x}_i$ to $\mathbf{x}_{i+7}$ from L1D cache and only need additional $\mathbf{x}_j$ and $\mathbf{x}_{j+1}$ from memory or next level cache. The L2 level blocking works in a similar way.

Hand-vectorization. As for the AVX optimization, we unrolled the inner most loop and used the AVX vectors to load and compute on multiple dimensions at once, which is very straight-forward.

Operational Intensity. We only need to compute for half of the pairs because the result matrix is symmetric, but we need to copy the results to the other half of the matrix because we need a full matrix in Part 2. In the optimized version, the number of flops is $W_{\text{opt}} = (D_{\text{in}} + 1)\, N^2 + (D_{\text{in}} - 2)\, N$. If we use $B_i \times B_j$ blocking, the amount of data transfer is

$$Q_{\text{opt}} \geq 4\left(\frac{1}{2} \cdot \frac{N^2}{B_i B_j} \cdot B_j D_{\text{in}} + 2N^2\right) = \left(\frac{2D_{\text{in}}}{B_i} + 8\right) N^2.$$

The operational internsity is $I_{\text{opt}} \leq \frac{D_{\text{in}}+1}{\frac{2D_{\text{in}}}{B_i}+8} \approx 0.5B_i$ for large D_{in}.

In our computer, L1D cache can store about 12 samples points and L2 cache can store about 128 sample points, and the CPU has 32 floating point registers which can be used to compute 16 dot products simultaneously. We chose the blocking sizes mentioned above such that B_i is maximized under these constraints.

Part 2 (High-Dimensional Affinities).

This part computes the high-dimensional affinities using the results of Part 1. This is the most complex part of the t-SNE algorithm and we performed many optimizations here.

Beta tricks. The Gaussian distribution of the pairwise distances is found by doing binary search on the variance σ_i until the newly calculated perplexity is close enough to the given hyper-parameter P. However, to make this search more efficient we replace $-1/2\sigma_i^2$ by β_i and perform binary search on β_i. This could already be considered an optimization, however, we included this in our baseline because it was also used by the reference implementation of the authors.

Some other things worth mentioning here: (a) At the beginning of binary search (bootstrap stage), $\beta_i^{\max}$ and $\beta_i^{\min}$ are not available. We search for the upper or lower bound by doubling or halving β_i, which will hopefully be found soon. (b) On the contrary to the version of original authors, we do not use any sort of normalization of the input data $\mathcal{X}$ or squared distances d_{ij}. To avoid rounding issues for very large distances, i.e., the $\exp$ values would be too close to 0 which causes problems, β_i is initialized to $-1/\max_j d_{ij}$ as a way of normalization, instead of -1 (or 1) in the reference implementation by the authors. These are already applied in our baseline code.

Removing exponential functions. We have got three nested for-loops: the outer loop for i (N times), the intermediate one for binary search (unpredictable, suppose average number of iterations to be M), and the inner one for j (N times). The complexity of this algorithm should be $\mathcal{O}\left(MN^2\right)$. Moreover, the MN^2 times of $\exp$ function seems extremely costly in this piece of code, serving as the bottleneck. Thus, eliminating the $\exp$ function within the inner loop is the core part of our solution.

The binary search for β_i is stated as follows: we've got initial value β_i and upper/lower bounds $\beta_i^{\max}$ and $\beta_i^{\min}$. For each binary search iteration, we calculate $\exp\left(\beta_i d_{ij}\right)$ and accordingly the new perplexity. We judge the new perplexity with the target value P, update $\beta_i^{\max}$ or $\beta_i^{\min}$ to current β_i, and update the β_i in either direction to $\left(\beta_i + \beta_i^{\max}\right)/2$ or $\left(\beta_i + \beta_i^{\min}\right)/2$.

The interesting fact is that, when calculating the current perplexity, all we need is the $\exp(\beta_i d_{ij})$, where d_{ij} is fixed across each round. While updating β_i, we thought about the possibility of avoid calling the $\exp$ function over and over again. And the answer lays in a simple mathematical transformation: $\exp\left((a+b)/2\right) = \sqrt{\exp\left(a\right) \cdot \exp\left(b\right)}$.

This means we can deduce the new exponential values from the original ones without involving extra $\exp$ function calls. More specifically, we built an array of size N storing the $\exp\left(\beta_i d_{ij}\right)$ for each position j at the beginning of the outer i loop, and another two arrays for $\exp\left(\beta_i^{\max} d_{ij}\right)$ and $\exp\left(\beta_i^{\min} d_{ij}\right)$ as well. Then all we need to do is to up-

date these values along with the binary search and without calling the exp in the inner loop.

The exp function consists of more than 10 primitive flops, much more costly than the square root. The replacement of the exp function to a multiplication and a square root also enables vectorization. The SIMD instruction of exp does not exist on x86 but that of multiplication and square root are already supported by hardware.

Other scalar optimizations. Up to here we already have a 2x speedup for scalar version. Other things we did include:

We used a cubic polynomial to approximate the exp function during the first iteration of binary search, so that the exp function is fully removed from our code. The exp implementation of glibc math library uses a high-degree polynomial together with some range reduction techniques. This isn't quite effective, as the input of the first iteration is $-\frac{d_{ij}}{\max_j d_{ij}} \in [0, 1]$. A cubic polynomial is sufficient for this small input domain.

We also need to symmetrize the affinity $p_{i|j}$ and $p_{j|i}$ to p_{ij}. We did a simple 16×16 blocking to make sure each cache block is load and stored at most once during the symmetrization.

Hand-vectorization. AVX optimization is obvious in this way. We mentioned before that we have introduced 3 arrays of size N on which we do multiplication as well as square root. These operations can naturally be vectorized. For the affinity symmetrization, some shufflings are required.

Part 3 (Low-Dimensional Affinities).

Part 3 computes the low-dimensional affinities. We designed a special way to use the transposed y matrix with some blocking techniques in this part.

Part 3 looks very similar to Part 1 (computing the squared Euclidean distances). However, the dimension of data points in Part 3 is very small, which creates additional challenges for optimization, and therefore different optimizations are needed.

Transposed matrix. We first analyzed the unique problems of small dimensions. A typical way to store N data points of D dimensions is to create a $N \times D$ matrix, as what we did in Part 1 and the baseline version of Part 3 and 4. Several dimensions from the same data point can be loaded in one SIMD vector as they are stored together. Figure 1(a) shows the shape of this kind of matrix. Note that when the number of dimensions is not divisible by vector length, it is safe to pad 0 as it has no influence on the pairwise distances or the gradient. Figure 1(b) shows that padding enables using SIMD instructions for dimensions that are smaller than vector length. However, for small dimensions, the waste of computation resources is relatively big. Figure 1(c) is another solution. It packs multiple samples (2 in this figure) in one vector, but this is not possible for any dimension and

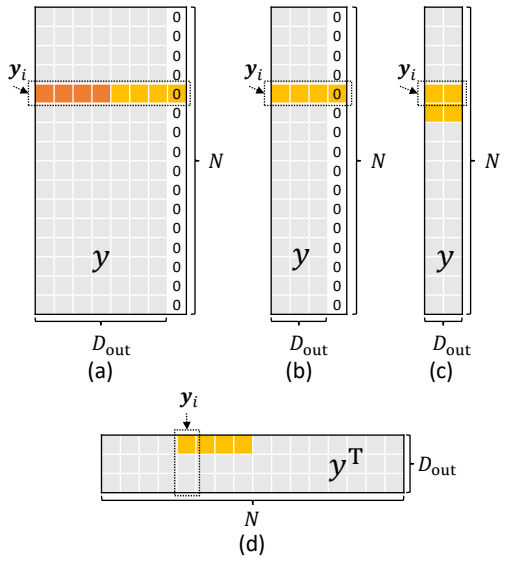

Fig. 1. Possible ways to deal with low-dimensional data points. This figure assumes using SIMD vectors of length 4. The orange or yellow cells can be directly loaded into one vector. We use (d) transposed matrix to store low-dimensional data points in Part 3 and 4.

requires handcraft for each possible dimension.

An ultimate solution is to use transposed matrix where data points are stored in columns, as shown in figure 1(d). There is no waste, and it is generalizable to any small dimensions. Each SIMD vector can load one dimension from multiple samples at once, which is also a natural way of register level blocking.

Blocking. In Part 1, we showed 2 ways to compute the distances. We used the straight-forward method because it has less flops for our typical use case (2 or 3 dimensions). Figure 2 illustrates our computation method. With the transposed y matrix, we can do $B \times B$ register level blocking. We compute the (I, J)-th block of t matrix using the I-th and J-th block of y matrix. All the intermediate $B \times B$ square blocks are in register files, and the final results can be directly stored into the corresponding position of t matrix. We also compute the summation $S = t_{ij}, \forall i \neq j$, during computation, which will be used in Part 4.

We only need to calculate half of the matrix since the t matrix is symmetric. The baseline also has this feature. But in our optimized version, we only store half of the matrix, while the baseline also copies the data to another half of the matrix. This optimization is related to Part 4's optimization.

AVX version. We use AVX-512. We chose block size $B = 16$ because each 512-bit vector can store 16 floats and so this is the natural blocking size, and the 32 zmm registers can provide enough space for all the intermediate values. Our experiment shows this blocking size is good

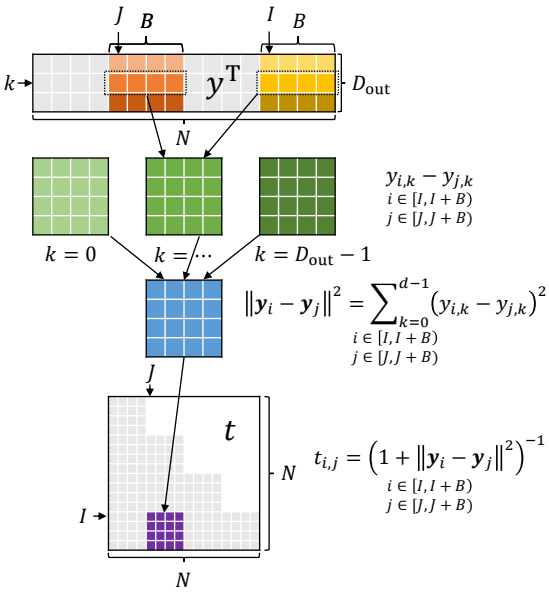

Fig. 2. Computation in Part 3 (using blocking).

enough, a second level of blocking have no benefit. We also implemented AVX2 version for this part and part 4. However, we use AVX-512 to explain our findings in the experiment part since the points are similar.

Scalar version. We chose block size $B = 4$ because we need to make sure all intermediate values can be stored in registers. We further did a 16×16 second level of blocking to match the AVX version. Compared to AVX version, one advantage might be that it does not need shuffling to compute the "outer product" style square blocks. But our experiment showed the AVX version is still much faster. The shuffling overhead does not shadow the benefit of vectorization.

Operational Intensity. Because D_{out} is very small, the y matrix can fit into L2 cache even if N is very large. Also, because Part 3 is inside the gradient descent loop, y matrix may be able to be preserved in cache. We only need to load and store half of the t matrix. The memory traffic is $Q_{\text{opt}} \geq 4N^2T$. The number of flops is $W = (1.5D_{\text{out}} + 1)\, N\, (N-1)\, T$. The operational intensity is $I_{\text{opt}} \leq \frac{3D_{\text{out}}+2}{8}$.

Part 4 (Gradient descent).

Part 4 computes gradient and update the low-dimensional data points. Because we need $S = \sum_{i \neq j} t_{ij}$ in this part, Part 4 cannot be merged with Part 3. Part 4 also uses transposed matrices, following a similar principal to Part 3.

Eliminating g matrix. The u matrix is the update matrix and g matrix is the gradient matrix, which have the same shape as y matrix. We made some small changes on computation to eliminate the g matrix.

$$\forall i = 1..N, \quad \mathbf{u}_i = \mathbf{u}_i - \textstyle\sum_j \left(p_{ij} - t_{ij} S^{-1}\right)(\mathbf{y}_i - \mathbf{y}_j)\, t_{ij}$$
$$\forall i = 1..N, \quad \mathbf{y}_i = \mathbf{y}_i + 4\eta \mathbf{u}_i, \quad \mathbf{u}_i = \alpha \mathbf{u}_i$$

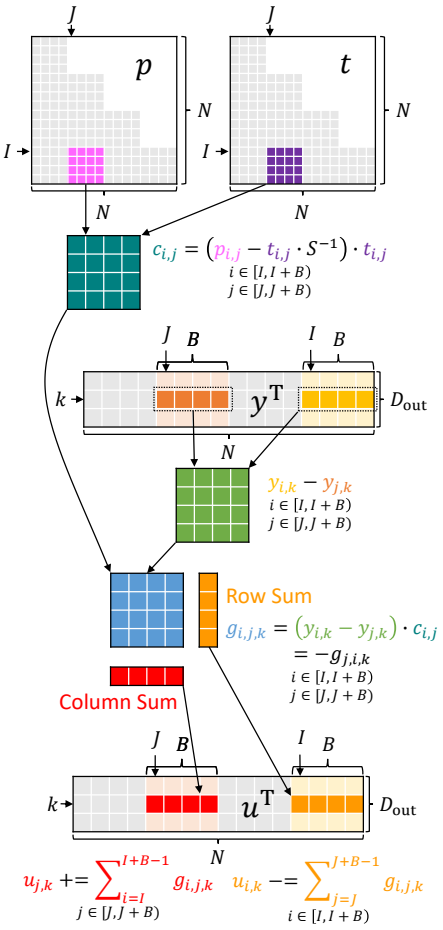

Fig. 3. Computation of transposed u matrix in Part 4 (using blocking).

Transposed matrices. We were particularly interested in computing u matrix since this is the $\mathcal{O}\left(N^2\right)$ computation in this part. The optimized method to compute transposed u matrix is shown in figure 3.

It is worth to note that in Part 4, all the computations are dimension independent. The values in one dimension have no computation with the values in another dimension. This confirms the usefulness of the transposed matrix for data points.

Using only half matrices. An important optimization is to utilize the antisymmetric property of $g_{i,j,k}$ (colored in blue in figure 3): $g_{i,j,k} = -g_{j,i,k}$. Thus, we only need to do half of the calculations, and get the gradient for both I-th block and J-th block by taking row sum and column sum of the square block. This not only reduces flops, but also allows us to only read half of the p and t matrix, and therefore only half of p and t matrices are needed to be filled in Part 2 and Part 3.

Blocking. Same as in Part 3, the block size B is 16 in

AVX version and 4 in scalar version. In scalar version, there is also a second level 16×16 blocking.

Operational Intensity. Both y and u matrix can fit into L2 cache even if N is very large. Because Part 4 is also inside the gradient descent loop, y and u matrix may be able to be preserved in cache. We only need to load half of the p and t matrix. The memory traffic is $Q_{\text{opt}} \geq 4N^2T$. The number of flops for the optimized version is $W_{\text{opt}} = N\left((2D_{\text{out}} + 1.5)\,N + D_{\text{out}} - 1.5\right)T$. The operational intensity is $I_{\text{opt}} \leq \frac{4D_{\text{out}}+3}{8}$.

4. EXPERIMENTAL RESULTS

In this section we discuss the benchmark results of our implementations. We evaluated the baseline, the best scalar version, and the AVX (AVX-512) version.

Experimental setup.

Our benchmarking computer has an Intel Core i7-1065G7 processor with Ice Lake Client architecture and a base frequency of 1.30 GHz. The cache sizes for a single core are: 32 kB L1I, 48 kB L1D, 512 kB L2, and 8 MB L3 (shared). We disabled hyper-threading and turbo boost in BIOS. This CPU supports AVX-512, but it has the same peak performance as AVX2, because it can only issue either 1 512-bit floating point instruction or 2 256-bit instructions per cycle. However, we still benefit from the larger register files (zmm0-31 and ymm/xmm16-31), the larger L1 load/store bandwidth, and the larger integer performance.

Our system runs on Ubuntu 20.04.2 LTS 64-bit. We used GCC/G++ 11.1.0 to compile our code with flags `-O3 -march=icelake-client -ffast-math`. When evaluating without auto-vectorization for the baseline and scalar versions, we also added `-fno-tree-vectorize` flag. We also tried Intel C++ Compiler but there was no improvement and thus we did not include this compiler in our plots. We used the `tsc_x86.h` from Assignment 2 to measure the runtime in CPU cycles using the RDTSC register. To draw the performance plot we used Intel SDE to measure the exact number of flops, although we already calculated this numbers from a theoretical analysis.

As we mentioned previously our code works with any input. For the experiments we wanted to use a typical use case. We thus used the MNIST dataset[3] as it is used in the t-SNE paper with an input dimension of $D_{\text{in}} = 784$. The number of gradient descent iterations T was fixed to 1000 and the target perplexity P to 50. This values among the other parameter were chosen according to scikit-learn library's default value [4]. We mainly tested with output dimension $D_{\text{out}} = 3$ but also explored other output dimensions for part 3.

Part 1 (Squared Euclidean Distances).

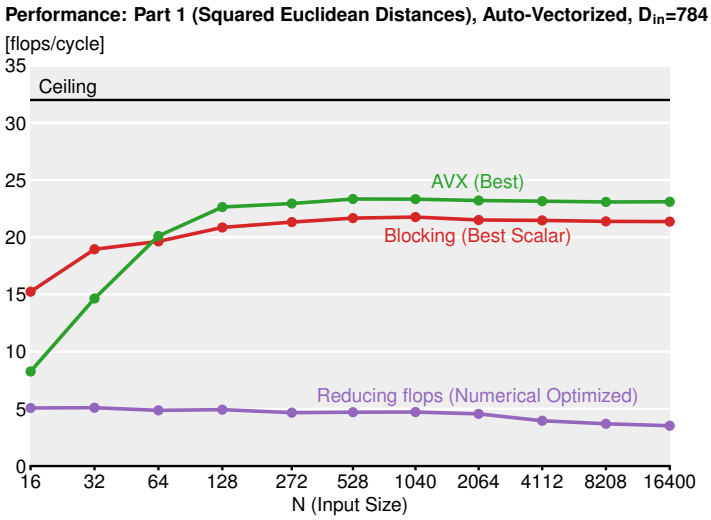

Fig. 4. Performance of Part 1 with $D_{\text{in}} = 784$.

The performance of compiler auto-vectorized codes are shown in Figure 4. The best scalar version can achieve almost the same performance as the hand-vectorized AVX version. The AVX version has some small overhead so it does not beat the scalar version when N is very small. Because this part has balanced addition and multiplication, the performance ceiling is 32 flops/cycle. For larger N the performance is about 23 flops/cycle (72% of the peak performance) and the speedup compared to baseline is about 5-8x.

Part 2 (High-Dimensional Affinities).

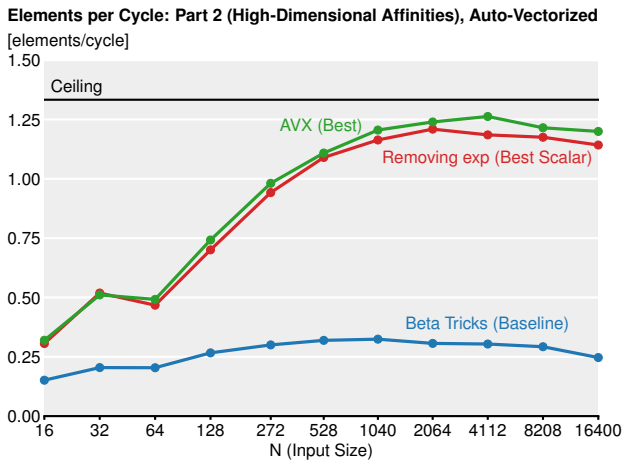

Fig. 5. Performance of Part 2 measured in elements/cycle. We do not compare flops/cycle because the operation types and counts are greatly different.

In this part, we do not compare flops/cycle because the operation types and counts greatly varies in different versions. For example, we have exp functions only in baseline and square root only in optimized versions. Even though we

[3] http://yann.lecun.com/exdb/mnist/index.html

[4] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html

basically used the same computation in best scalar and AVX codes, the compiler can almost double the flops while doing auto-optimizations in scalar version. Therefore, we choose to use element/cycle as the performance measure. The number of elements is the total number of binary search iterations: $\sum_i m_i N$, where m_i is the number of binary search iterations of the i-th row. The performance of compiler auto-vectorized codes are shown in Figure 5. The best scalar version can achieve almost the same performance as the hand-vectorized AVX version.

This part is bounded by the square root computation. Each element roughly corresponds to a square root computation. We use AVX-512 instructions (16 floats per vector) which have a gap of 12 cycles. Compiler auto-vectorization uses AVX2 instructions (8 floats per vector) which have a gap of 6 cycles. The throughput is both 1.33 elements/cycle. For large N the performance (i.e. elements/cycle) is very close (>90%) to the upper bound. The speedup is about 3-5x.

We also measured the performance in flops/cycle, taking into account all kinds of floating point operations. The performance of the AVX version is about 6 flops/cycle and the best scalar version (not using any exp function calls) is 10 flops/cycle.

Part 3 (Low-Dimensional Affinities).

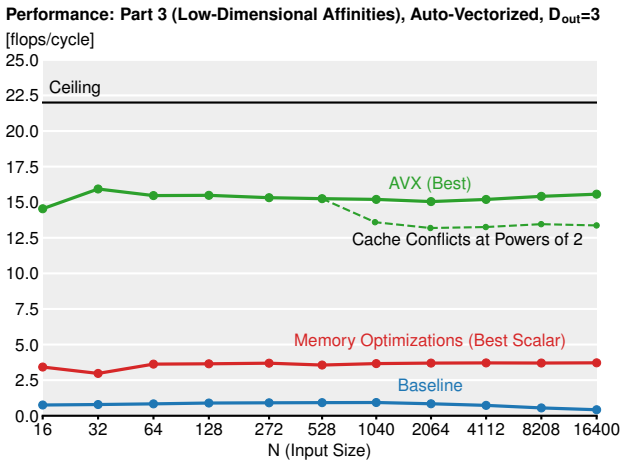

Fig. 6. Performance of Part 3 with $D_{\text{out}} = 3$.

The performance of the compiler auto-vectorized codes are shown in Figure 6. The performance of baseline version drops when the input size is larger than 4000 (out of L3 cache), which means it becomes memory-bounded as we increase the input size.

After applying memory optimizations such as blocking, the performance (around 3.7 flops/cycle) is very stable when increasing the input size. This part involves many "outer product" style computation, so it is very hard for the compiler to auto-vectorize. We checked the assembly code and indeed there is little vector instructions. We achieved 2-5x speedup in this version.

The best AVX version (green line in the Figure 6) achieved 15 flops/cycle on performance, which is 68% of the peak performance based on instruction mix and dependency (22 flops/cycle). Although the shuffling operation uses port 5 which does not conflict with port 0/1 for floating point arithmetic, we believe the shuffling still has some overhead such as long latency and data dependency. Also the long latency (7 cycles) and low throughput (0.5 instruction/cycle) of calculating the reciprocity may play a role. However, we achieved an impressive speedup of 15-40x because the baseline has an extremely poor performance for larger N.

The green dash line shows the cache conflict effect when the input size is the powers of 2. For the actual AVX plot line we used slightly uneven input sizes to avoid this corner cases. For future work we could use padding in the input size dimension to avoid the power of 2 sizes.

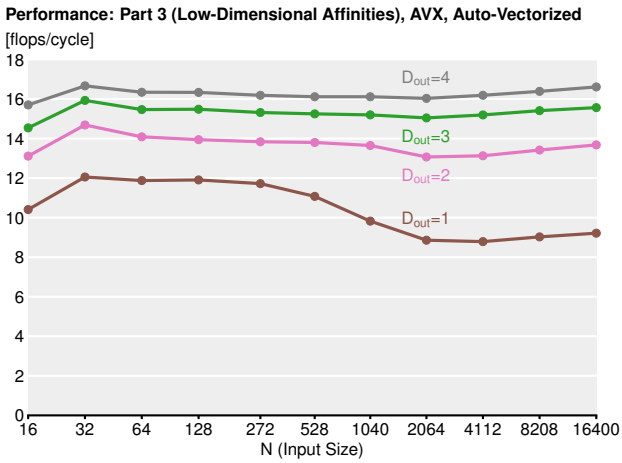

Fig. 7. Performance of AVX (best) version of Part 3. Same binary executable file with different D_{out} parameters.

In the Figure 7, we investigated the performance of our best AVX version for different output dimensions. As we can see, the performance increases with the output dimension. We believe this is caused by the latency of calculating the reciprocity. The portion of add/mul increases with the dimensions, so the reciprocity will become less an issue in higher dimensions. Also, the intensity $\frac{3D_{\text{out}}+2}{8}$ is related to the dimension. For $D_{\text{out}} = 1$, this is clearly memory bound, as the performance starts to drop when it is out of L2 cache and reaches the low platform when it is out of L3 cache. For actual use cases of t-SNE only $D_{\text{out}} = 2$ and $D_{\text{out}} = 3$ are relevant because only in this dimension we can plot a meaningful visualization of data.

Part 4 (Gradient Descent).

The performance of compiler auto-vectorized codes are shown in Figure 8. It has many similar characters as Part

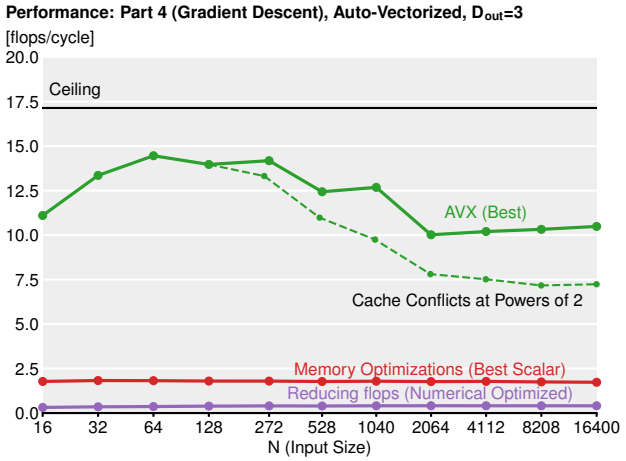

Fig. 8. Performance of Part 4 with $D_{\text{out}} = 3$.

3. In the best AVX version, we achieved 10 flops/cycle for large N, which is 59% of the peak performance based on instruction mix and dependency (17 flops/cycle). The speedup is about 20-25x. Although this part has a higher intensity than Part 3, it looks more memory bound. We believe this part has more strided loading which causes too many cache conflicts. And it is also more sensitive to the powers of 2 inputs.

Overall analysis. Part 3 and 4 which are inside the gradient descent loop and therefore play an important role in the overall runtime. Compared to the baseline the runtime composition of the different parts changed but the overall magnitude stayed the same. Table 2 in the appendix shows the runtime composition of the baseline and AVX versions.

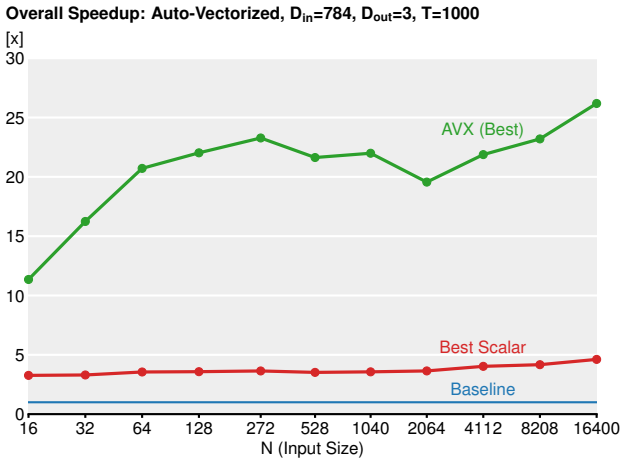

Fig. 9. Overall speedup with $D_{\text{in}} = 784$, $D_{\text{out}} = 3$, $T = 1000$.

Figure 9 shows the overall speedup of the best scalar and AVX versions compare to baseline. With $D_{\text{in}} = 784$, $D_{\text{out}} = 3$, $T = 1000$, we can achieve a speedup of 20-26x. This speedup is dominated by the improvement of Part 3 and 4.

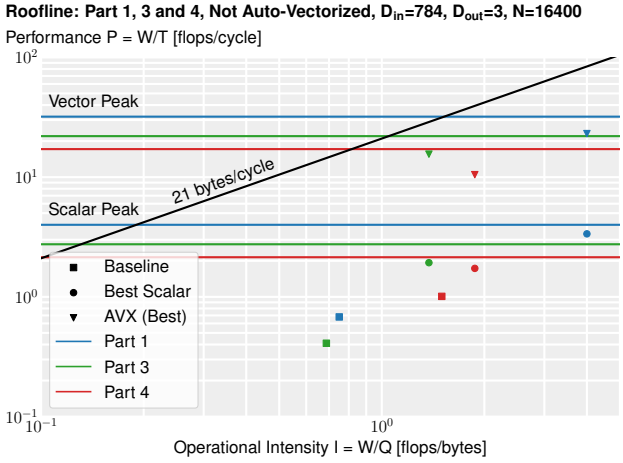

Fig. 10. Roofline plot of Part 1, 3, and 4. $D_{\text{in}} = 784$, $D_{\text{out}} = 3$, $N = 16400$. We do not include Part 2 because the performance ceiling (instruction mix) depends on the data.

Figure 10 is the roofline plot. The AVX performance is close to the peak, and we are within 2x of the potentially highest performance. Our optimizations have greatly increased the intensity. Note that although our optimized versions do not seem to be memory bound, they might still be bounded by L3 or L2 caches (like the what is shown in Figure 8).

5. CONCLUSIONS

In this paper we showed multiple optimization techniques to improve the overall runtime of t-SNE. One aspect of our work was to reduce the number of flops. Using multiple steps of optimizations we were able to fully omit the use of the expensive exponential function in the second part.

The gradient descent part marks the bottleneck of the algorithm and is memory bound. For part 3 and 4 we thus focused on improving the spacial locality by using blocking.

Many of these optimizations we applied are not limited to the t-SNE algorithm but can also be used in many other computation. The optimized squared Euclidean distances or the gradient descent part are relevant beyond this algorithms and are used in many other machine learning algorithms.

We were able to speedup the total runtime by 20-26x. We believe in this paper we showed a significant improvement in the runtime of the algorithm without changing the correctness of the algorithm and without limiting it to certain dimensions. The runtime could be further improved by applying algoritmic optimization in the gradient descent such as early stopping. Additionally our optimized code could also be parallelized to improve the runtime even more.

6. CONTRIBUTIONS

In this section we briefly describe the individual contributions of each team member.

Shengze. Focused on the baseline, scalar and AVX2 version of part 3 and 4. Applied multiple optimization techniques, including cache optimization, block optimizations, vectorization and using symmetric property to reduce memory access and computation. Worked with Jiale, Muyu and Levin to combine different parts of the algorithm.

Jiale. Worked on every part and proposed many important optimization suggestions, such as reducing the flop count in Part 1, 2, and 4, and use transposed matrix in Part 3 and 4 to unlock small dimensions. Helped Shengze, Muyu, and Levin to fix the bugs and solve the performance issues in the baseline and scalar versions. Implemented the whole AVX-512 version.

Muyu. Mainly worked on part 2 with Levin, implementing scalar and avx2 version for binary search.

Levin. Focused on non-AVX optimization for part one and two. Worked with Muyu on the scalar version of these two parts. Implemented the optimized computation of the squared euclidean distances as well as blocking of different sizes to reduce cache misses.
Implementation of the baseline code of part one and two and combining the different parts together.

7. APPENDIX

Equations.

Part 1

$$W_{\text{opt}} = (D_{\text{in}} + 1) N^2 + (D_{\text{in}} - 2) N$$

$$W_{\text{base}} = (1.5D_{\text{in}} - 0.5) N (N - 1)$$

$$Q_{\text{opt}} \geq \left(\frac{2D_{\text{in}}}{B} + 8 \right) N^2$$

$$Q_{\text{base}} \geq (2D_{\text{in}} + 8) N^2$$

$$I_{\text{opt}} \leq \frac{D_{\text{in}} + 1}{\frac{2D_{\text{in}}}{B_i} + 8} \approx 0.5B_i$$

$$I_{\text{base}} \leq \frac{3D_{\text{in}} - 1}{4D_{\text{in}} + 16} \approx 0.75 \text{ flops/byte}$$

$$\max P_{\text{opt}}^{\text{vec}} = 32 \text{ flops/cycle}$$

$$\max P_{\text{base}}^{\text{vec}} = 24 \text{ flops/cycle}$$

Part 2

$$W = O\left(MN^2\right) + N(N-1)$$

$$W_{\text{opt}} \neq W_{\text{base}}$$

$$Q_{\text{opt}} = Q_{\text{base}} \geq 14n^2$$

Part 3

$$W_{\text{opt}} = W_{\text{base}} = (1.5D_{\text{out}} + 1) N (N - 1) T$$

$$Q_{\text{opt}} \geq 4N^2T$$

$$Q_{\text{base}} \geq 8N^2T$$

$$I_{\text{opt}} \leq \frac{3D_{\text{out}} + 2}{8}$$

$$I_{\text{base}} \leq \frac{3D_{\text{out}} + 2}{16}$$

$$\max P_{\text{opt}}^{\text{vec}} = \max P_{\text{base}}^{\text{vec}} = \frac{24D_{\text{out}} + 16}{D_{\text{out}} + 1}$$

Part 4

$$W_{\text{opt}} = N((2D_{\text{out}} + 1.5) N + D_{\text{out}} - 1.5) T$$

$$W_{\text{base}} = 3N((D_{\text{out}} + 1) N - 1) T$$

$$Q_{\text{opt}} \geq 4N^2T$$

$$Q_{\text{base}} \geq 8N^2T$$

$$I_{\text{opt}} \leq \frac{4D_{\text{out}} + 3}{8}$$

$$I_{\text{base}} \leq \frac{3D_{\text{out}} + 3}{8}$$

$$\max P_{\text{opt}}^{\text{vec}} = 16 \cdot \frac{4D_{\text{out}} + 3}{4D_{\text{out}} + 2}$$

$$\max P_{\text{base}}^{\text{vec}} = 24 \text{ flops/cycle}$$

$$\max P^{\text{scalar}} = \frac{1}{8} \max P^{\text{vec}}$$

Table 1. Cost of Baseline

	Part 1	**Part 2**	**Part 3**	**Part 4**
add	$D_{\text{in}}N^2$	$2N^2M$	$D_{\text{out}}N^2T$	$2D_{\text{out}}N^2T$
mul	$D_{\text{in}}N^2/2$	$2N^2M$	$D_{\text{out}}N^2T/2$	$D_{\text{out}}N^2T$
div	0	N^2	N^2T	0
exp	0	MN^2	0	0
log	0	MN	0	0

Table 2. Comparison of time composition of baseline and best AVX version.

	Part 1	**Part 2**	**Part 3**	**Part 4**
Baseline	0.90 %	0.32 %	44.23 %	54.55 %
AVX	3.05 %	1.73 %	31.62 %	63.60 %

8. REFERENCES

[1] Geoffrey Hinton Laurens van der Maaten, "Visualizing data using t-sne," *Journal of Machine Learning Research 9*, November 2008.

[2] Laurens van der Maaten, "Accelerating t-sne using tree-based algorithms," *Journal of Machine Learning Research 14*, August 2014.

[3] David M Chan, Roshan Rao, Forrest Huang, and John F Canny, "Gpu accelerated t-distributed stochastic neighbor embedding," *Journal of Parallel and Distributed Computing*, vol. 131, pp. 1–13, 2019.