T-SNE

Aurelio Negri, Virgilio Strozzi, Ahash Thamilmaran, Yuchen Chang

Department of Computer Science
ETH Zurich, Switzerland

1. ABSTRACT

With this work, we present different implementations of the original $\mathcal{O}(n^2)$ t-SNE algorithm [1]. Starting from a baseline baseline version (**Baseline**), we investigate different levels of optimizations aimed at single-core execution on Intel's x86 architectures, and finally use AVX2 instructions, blocking, and some algorithmic tricks to improve performance. We produce two final implementations: one for the scalar case (**Algorithmic**) and one for the vectorized case (**AVX2**). **Algorithmic** achieves an average speedup of 2, while **AVX2** achieves a speedup of 4 compared to **Baseline**. Moreover, we also compare our implementations to an existing scalar C++ implementation [2]. We can report that **Algorithmic** achieves an average speedup of 3.3, while the vectorized **AVX2** achieves a speedup of 7 compared to the existing implementation.

2. INTRODUCTION

In this section, we briefly introduce the motivation, setup, and related work of t-SNE [1].

Motivation. Over the past years, the volume of high dimensional data, such as audio and images, has surged dramatically. Interpreting this vast amount of data in a meaningful way is critical in various data analysis tasks, and effective visualization plays a key role in this process. t-Distributed Stochastic Neighbor Embedding (t-SNE) [1] has become one of the most widely used algorithms for visualizing high-dimensional data.

The algorithm reduces data to a lower-dimensional representation, typically 2D or 3D, making it easier to make sense of the data. This low-dimensional representation is obtained by first fitting a Gaussian distribution to all data points in the high-dimensional space and then representing this high-dimensional Gaussian distribution as a low-dimensional Student's t-distribution.

The algorithm minimizes the Kullback-Leibler divergence between the high-dimensional and low-dimensional representations using gradient descent. This ensures that the low-dimensional representation approximates the high-dimensional representation as closely as possible.

Despite its effectiveness, t-SNE is computationally intensive and slower than other dimensionality reduction methods like PCA, posing challenges for large datasets.

Setup. We decide to work with the MNIST dataset [3] as our input. We preprocess the data with PCA to a fixed dimension of 64. We then focus our efforts on two dimensions as the target dimension for t-SNE for optimal visualization. We use AVX2 instructions working with doubles. We test our algorithm with various sample sizes, applying various computational and algorithmic optimizations.

Related work. The t-SNE algorithm, as proposed by the authors [1], has a computational and memory complexity of O(n^2), where n is the number of data points.

Due to its quadratic runtime, t-SNE can become impractical for larger datasets. To address this, the authors developed Barnes-Hut t-SNE [4], which uses tree-based algorithms to reduce computational and memory complexity to O(n log n), making it more suitable for larger datasets. An implementation using FFT called FIt-SNE makes use of this variant [5].

Another variant, called parametric t-SNE [6], uses neural networks to learn a parametric mapping from a high-dimensional space to lower-dimensional space. This allows for faster computations and generalization to new data points.

The authors have created a webpage that offers several implementations of t-SNE in various programming languages. All the papers and algorithms can be accessed on their official webpage [7].

3. BACKGROUND ON T-SNE

This section dives deeper into the t-SNE algorithm. It elaborates on the key parts of the algorithm and defines the cost measure and shows the cost analysis used for the performance plots in section 6.

t-SNE. The t-SNE algorithm takes as input n high-dimensional data points $\mathcal{X} = \{x_1, \ldots, x_n\}$ and maps them to some low-dimensional data-points $\mathcal{Y} = \{y_1, \ldots, y_n\}$. The goal of this mapping is to represent the data visually to reveal global structures while trying to preserve most of the local structures.

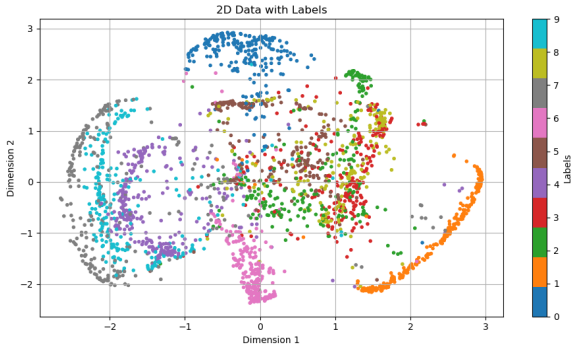

Fig. 1. Plot of the MNIST dataset with n=1024 images. The dataset is first reduced to dimension 64 using PCA and then to dimension 2 using our implementation of t-SNE

In Algorithm 2 below, we present the pseudocode for the algorithm. The algorithm is structured into different parts, each representing core components of our optimizations. This structured approach facilitates the optimization procedure and provides a clearer way to explain our optimizations. We describe each individual part in the following lines.

getP: First, this part computes the pairwise affinities $p_{i|j}$, which are conditional probabilities representing the similarity between points x_i and x_j. These probabilities indicate the likelihood that x_j is x_i's neighbor, based on a Gaussian probability density centered at x_i. Using these $p_{i|j}$, the algorithm then computes a joint probability p_{ij}. This joint probability serves as a similarity measure to ensure symmetry between the points.

getQ: Similarly, this part calculates the similarity measure q_{ij} for all low-dimensional representation points y_i and y_j. The key difference is the use of the Student-t distribution to compute q_{ij}. This distribution, characterized by heavy tails, is better suited for modeling moderate distances in the low-dimensional space.

getGrad: This part computes the gradient of the cost function, which is the Kullback-Leibler divergence minimized using gradient descent.

It is not explicitly highlighted in the pseudocode, but it is important to note that the Euclidean distances used in the various formulas are computed separately in a function called **getDistances**. This function is contained in both the **getP** and **getQ** part.

4. THEORETICAL ANALYSIS

In this section we describe the theoretical analysis of the **Baseline** implementation of t-SNE. In particular, we focus on the theoretical flops count W and the theoretical data movement count Q. We evaluate this count for each individual function of the Algorithm 2

Cost Function. The cost is defined as the sum of all flops in the algorithm and therefore dependent on the following parameters: number of images *n*, number of features *m* (=64), target dimension *target_dim* (=2), total number of iterations *n_iter*, number of binary search iterations *n_bin_iter* (=10) and size of the data type in byte*data_size_B* (=8). Formally we have:

$$
\begin{aligned}
&Cost(n, m, target\_dim, n\_iter) \\
&= Cost_{getP}(n, m) \\
&+ Cost_{getQ}(n, target\_dim) \\
&+ Cost_{getGrad}(n, n\_iter, target\_dim) \\
&+ Cost_{getDistances}(n, n\_iter, m, target\_dim)
\end{aligned}
$$

Baseline Theoretical Flops count (W). We describe the theoretical count of flops **W** in the baseline implementation (**Baseline**) for each individual part.

$$\text{getGrad} = 2n \cdot target\_dim + 5n^2 \cdot target\_dim + 1$$

$$\text{getQ} = n^2 + n^2$$

$$\text{getYDistances} = 3\left(\frac{n(n+1)}{2}\right) target\_dim + 3n^2$$

$$\text{getpij} = 5n + 1 + n$$

$$\text{getEntropy} = 5n + 1$$

$$\text{getP} = n \cdot n\_bin\_iter\,(4 + \text{getpij} + \text{getEntropy}) + 4\left(\frac{(n-1)n}{2}\right)$$

$$\text{initializeValuesY} = 1 + n \cdot target\_dim$$

$$\mathbf{W} = \text{getDistances}(n, m) + \text{getP} + \text{initializeValuesY}$$
$$+ niter\left(3\left(\frac{n(n+1)}{2}\right) m + \text{getQ} + n^2 + \text{getGrad} + 5n \cdot target\_dim\right) -$$

Baseline Theoretical Data Movement count (Q). We describe the theoretical lower bound of data movement **Q** in the baseline implementation (**Baseline**) for each individual part.

$$\text{distances} = n^2$$

$$\text{pij} = n^2$$

$$\text{Y} = 3n \cdot target\_dim$$

$$\text{grad} = n \cdot target\_dim$$

$$\text{y\_dist} = n^2$$

$$\text{qij} = n^2$$

$$\text{rij} = n^2$$

$$\mathbf{Q} = (\text{distances} + \text{pij} + \text{Y} + \text{grad} + \text{y\_dist} + \text{qij} + \text{rij}) \cdot data\_size\_B$$

Input: $\mathcal{X} = \{x_1, \dots x_N\}$ (High-Dimensional d_{in}),
perplexity $perp$, # iterations T, learning rate η, momentum $\alpha(t)$
Output: $\mathcal{Y} = \{y_1, \dots y_N\}$ (Low-Dimensional d_{out})

---

getP:

$\forall i, j$ compute pairwise affinities $p_{i|j}$ with $perp$: $p_{i|j} = \frac{\exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||^2/2\sigma_i^2)}$

$\forall i, j$ set $p_{ij} = \frac{p_{i|j} + p_{j|i}}{2N}$

---

sample initial solution $\mathcal{Y}^{(0)}$ from $\mathcal{N}(0, 10^{-4}I)$

for $t = 1$ to T **do**

---

getQ:

$\forall i, j$ compute low-dimensional affinities: $q_{ij} = \frac{(1+||y_i - y_j||^2)^{-1}}{\sum_{k \neq l}(1+||y_k - y_l||^2)^{-1}}$

---

getGrad:

$\forall i$ compute gradient: $\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1}$

$\forall i$ set $y_i^{(t)} = y_i^{(t-1)} + \eta \frac{\delta C}{\delta y_i} + \alpha(t)(y_i^{(t-1)} - y_i^{(t-2)})$

end for

Fig. 2. Pseudocode of the t-SNE algorithm is divided into different parts. For each part, we apply and describe optimizations throughout our work.

Practical Complexity. The asymptotic runtime of the t-SNE algorithm is dominated by the computation of the pairwise similarities of all points, since in general n is much larger than all the other parameters. Hence, the complexity is $\mathcal{O}(n^2)$, both from the computational and memory side.

5. OPTIMIZATIONS PERFORMED

In this section, we describe all the modifications we applied to the original implementation **Baseline**. As mentioned before, we split the algorithm into sub-functions in pseudocode 2, where we have applied all our optimizations. We focus on scalar optimizations in **Algorithmic** and on vectorized optimizations in **AVX2**.
We do not mention it explicitly, but we also apply common techniques such as blocking, common subexpression elimination, constant folding, and so on, to both **Algorithmic** and **AVX2** implementations.

GetDistances. In both steps **getP** and **getQ** of t-SNE algorithm, there's the need of computing the squared euclidean distance matrix of some data $A^{n \times v}$, for some dimension v. The entry of the distance matrix can be expressed in the following way by expanding the terms:

$$\begin{aligned} d_{ij} &= \sum_{k=0}^{v} (A_{ik} - A_{jk})^2 \\ &= (a_i - a_j)^T (a_i - a_j) \\ &= a_i^T a_i - 2 a_i^T a_j + a_j^T a_j \end{aligned}$$

This reformulation has some computational advantages, we first notice that we only need to compute half ($\frac{n(n-1)}{2}$) of the values of the distance matrix since it is symmetric. With the straightforward implementation, each entry requires v subtractions and v FMAs (we count them as 1 flop), bringing the total flop count to:

$$\frac{n(n-1)}{2} * 2v = n(n-1)v$$

When implementing the second variant, we first precompute the norms of the rows $a_i^T a_i$. This requires only v FMAs per row, resulting in a total of vn floating-point operations (flops). Next, we have $\frac{n(n-1)}{2}$ entries to compute, each costing v flops for the dot product, plus one FMA and one addition. Combining all the computations, we get:

$$\frac{n(n-1)}{2}(v+2) + vn$$

By plotting the fraction of these two functions with n fixed, we see that already for $n = 100$, the second implementation reduces flops by nearly 50% due to the efficient use of

FMAs and precomputations. Only for $v = 2$ are the number of flops the same for $\lim_{n\to\infty}$. Despite these improvements towards flop count, the performance doesn't improve. This suggests that the function is memory bound. Therefore, we need to resort to blocking to improve operational intensity. We implement second level dynamic blocking based on the number of features v such that at each outer iteration the whole data can fit in the L1 cache. This can speed up the execution time by up to 7 times compared to the naive vectorized implementation. The smaller blocks on the other hand are of size 4x4. This allows us to easily transpose them and store the upper triangular matrix if needed.

GetP. In **Baseline**, we first use a function called *getpij* to compute the pairwise affinities $p_{i|j}$ using a precomputed matrix of euclidean distances from **getDistances** and then, we use them to compute a matrix of the joint probabilities. As an optimization idea, we notice that the joint probability distribution p_{ij} is symmetric. Thus, we are able to use a one-dimensional array instead of a two-dimensional array. The results of those optimizations are that we have less computations (n^2 vs $\frac{n(n+1)}{2}$) and less memory usage (n^2 vs $\frac{n^2}{2}$). This is applied in both **Algorithmic** and **AVX2**. During the development of **AVX2**, we faced the fact that no vector instructions for exp and log functions are present in AVX2. Hence, we had to find a different solution for this problem. The solution was to write exp and log functions on our own that we were able to use on vectors. We achieved that by approximating those functions with a vectorized cubic Taylor approximation. In the whole optimization procedure, we also applied blocking using $4x4$ blocks to improve operational intensity.

GetQ. In **Baseline**, we separately implement the getDistances function in the low-dimensional space and the remaining operations of getQ. We notice some room for improvement when combining the two functions; this allows us to save a pass through the whole matrix, as we can compute the sum while we are computing the Y distances. This approach introduces a few challenges. Since we are only computing one half of the matrix, we need to multiply the sum by two. However, because we are working with 4x4 blocks, the diagonal blocks will include values from the upper triangular matrix. To correct this, we need to adjust the sum by these values, which is done using maskloads within the main getDistances loop. Another operation we need to perform is to set the values on the diagonal to 0. This is because when we look at the sum in the denominator, the diagonal is not included. There are various ways to achieve this, what we decided to do is to do a **_mm256_cmp_pd** followed by an **_mm256_and_pd** to set the entries that are equal to 1 to 0. This increases the number of cycles but these operations have a small latency so it's not a big issue. Finally, we iterate again through the matrix and divide it by the sum we previously computed. Another trick's used is to take advantage of the target dimension that we set to 2 when computing the Y distances. This allows us to fit two rows of Y inside of an intrinsic which we then process in the following way: As we can see, this allows us to compute the dot

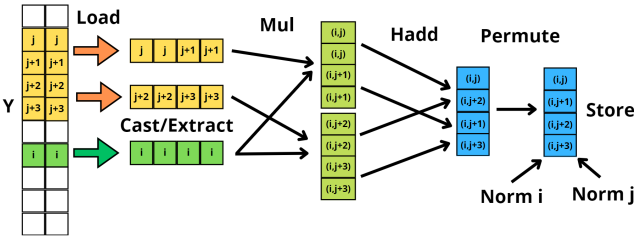

Fig. 3. Operations on matrix Y when target dimension is 2

product between the row i and the rows $j, j+1, j+2, j+3$ in an efficient way, and therefore efficiently get the distances. In the case the target dimension is larger than 2, the computation reverts back to the base vectorized implementation. In general, for **Algorithmic** and **AVX2** we exploit the symmetry of the matrix Q to halve the computations.

GetGrad. The main optimizations here are related to the symmetry of the data in P and Q, and in exploiting the anti-symmetric property of the gradient. In particular, when considering the gradient equation $\frac{\delta C}{\delta y_i}$, for a fixed y_i, we notice that the part inside the sum for a fixed j is anti-symmetric to $\frac{\delta C}{\delta y_j}$ when fixing i inside the sum:

$$
\begin{aligned}
&4\sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1} \\
&= 4\sum_j (p_{ji} - q_{ji})(y_i - y_j)(1 + ||y_j - y_i||^2)^{-1} \\
&= -4\sum_j (p_{ji} - q_{ji})(y_j - y_i)(1 + ||y_j - y_i||^2)^{-1}
\end{aligned}
$$

Hence, in the first step we exploit the fact that the matrices P and Q are symmetric on i, j, while the last step is a matter of taking the minus sign out of the computation. With this optimization, we only need to compute half of the terms. Implementation-wise, to reduce the number of memory accesses, we load and store elements that only depend on the outermost loop variable in that same loop body instead of loading and storing in the innermost loop. We reflect all of this optimization in **Algorithmic** and **AVX2**. For **AVX2**, we also included blocking. Similar to getQ, we used 4x4 blocks to achieve better operational intensity. It is also possible to use FMA instructions, which leads to improved performance. Even though it is recommended to use **_mm256_loadu_pd** instructions instead of **_mm256_set_pd** instructions to access the elements, we had to opt for the latter because the elements we access are not continuous in memory.

Flag optimization. On **Algorithmic**, we test different flag combinations for `g++`. For this, we write a python script that takes in the most common optimization flags used, creates random combinations of size 5 and stores them in a batch file that is then run. The full algorithm is then run 20 times with fixed input and the mean runtime of the last 15 runs is taken. We notice right away that taking away `-O3` results in a big performance loss. Therefore, we decided to include `-O3` with some additional flags. This allows some minor improvements of about 10% but due to the small amount of change we decided to just use `-O3` for the standard optimization flag.

6. EXPERIMENTAL RESULTS

In this section, we explain the system setup of our experiments. Moreover, we present, comment and compare the results of the different implementations of t-SNE. In particular, we make use of a baseline version (**Baseline**), the best algorithmic improved-version (**Algorithmic**) and the best final implementation using vectorization (**AVX2**). The following flags, **Autovec** and **Opt.**, attached to the names of the implementations, mean that autovectorization via the compiler and algorithmic optimizations are applied, respectively.

Experimental Setup.

We run all of our benchmarks on an Intel Core i7-1165G7 processor, utilizing the Tiger Lake architecture with a base frequency of 2.80 GHz. The cache sizes are as follows: 48 kB L1, 1.25 MB L2, and 12 MB L3 (shared across cores). We disable **Turbo Boost**, while also fixing the frequency to 2.80 GHz during the whole experiment and trying to reduce the number of running process in the system to the minimum. We vectorize the code using AVX2 instructions when needed.

Our system operates on Ubuntu 22.04 64-bit. We compile our codes using G++ 20.0 with the following flags: `-O3 -std=c++20 -march=tigerlake`, which showed to provide the best results. Notice that these flags autovectorize the code, which means that when we refer to not vectorized code, we also add the flag `-fno-tree-vectorize`.

We measure the runtime in CPU cycles using the RDTSC register via the `tsc_x86.h` header from the exercises. To assess performance, we use PAPI [8] for measuring data movements and the number of floating-point operations, hence adopting empirical measurements rather than theoretical in the plots. The theoretical calculations for the flops operation (W) and data movement (Q) of the baseline can be found in the Appendix at 4. Lastly, bandwidth measurements for the roofline plot are conducted using the STREAM [9] benchmark.

Our experiments involves varying the number n of input images in the following range: 64, 128, 256, 512, 960, 1440, 1920, 2480, 2960, 3440, 3920, 4880, 5904, 6928, 7952, 8976, 10000, while measuring the number of cycles, number of flops and memory movements across the various implementations. The input data to each run consists of n MNIST images with 64 features, gathered by pre-processing each image with PCA to allow faster computations, and the output data comprises n images with 2 projected features. We use double precision floating-point and we fix the number of iterations at 250, the perplexity at 30, and the number of binary search iterations at 10. To achieve a good visual result, we also made use of exaggeration as in the original paper. All the implementations are tested on correctness of their output with the baseline implementations.

The tests are performed using the functional library of c++. We compute the outputs for each function using a working python implementation first and store the results in csv files, then we pass the function we want to test as a parameter to the $test$ function together with the csv file path and we compare the outputs with the ones we stored beforehand. This allows us to write only one $test$ function for all the other functions.

Lastly, each implementation's benchmark is run with a cache warmed across 5 runs, and the results are then averaged over 10 measurements to ensure reliability.

Results. We decided to compare the best implementations under a single performance plot 4, a speedup plot 5 and a roofline plot 6. Clearly, different implementations can have different operation types and flops, hence the performance plot is an indication of how the different implementations perform. Nevertheless, when viewed alongside the speedup plot, these plots together can provide meaningful information into the behavior of the different implementations.

We first analyse the performance plot 4 and as expected, we can clearly notice how the performance decreases when the input data doesn't fit in the cache anymore. This slowdown is more apparent in the vectorized implementation.

We now consider the speedup plot 5 and overall, we notice the expected speedups since we are working with doubles of which only 4 fit into a vector register. Since the AVX2 version actually also includes the improvements we did in the algorithmic version, one would expect a larger speedup. The problem with this is that in general, for the algorithmic implementation the complexity of the code increased, which meant that a straightforward implementation is not possible anymore and therefore, we observe only a rough 2X speedup.

Finally, the roofline plot 6, which is the most informative one, confirms our suspicions from the performance plot, which is that in general, our implementation is memory bound. This is mostly due to the fact that, as seen in the graph, the operational intensity is quite low or in other words, we are not doing many operations on each byte.

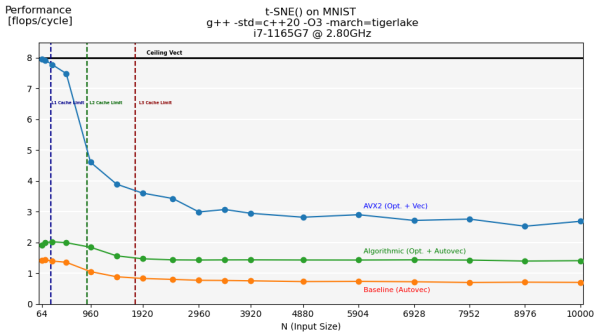

Fig. 4. Performance comparison of the 3 double-precision implementations of t-SNE. All the implementations outside **AVX2** tested here are autovectorized by the `-O3` flag.

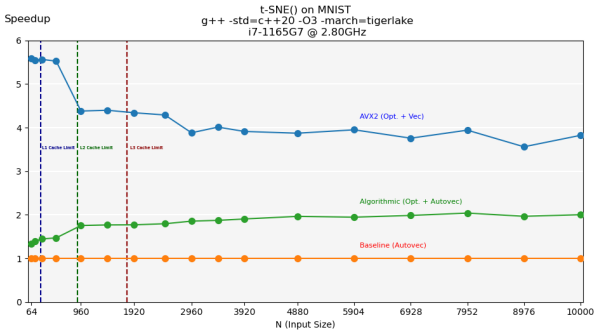

Fig. 5. Speedup of all the implementations compared to the autovectorized **Baseline** (Autovec), for different input sizes n.

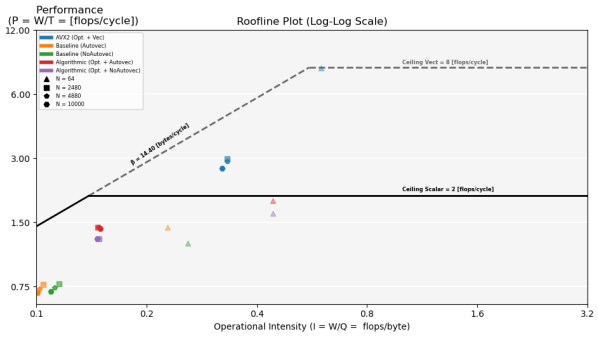

Fig. 6. Roofline plot for all the implementations on different input sizes n. We plot both the autovectorized and non-autovectorized versions of **Baseline** and **Algorithmic** versions.

7. CONCLUSION

To close off, we notice how mixing ideas from the lecture, mainly blocking and unrolling, with other neat performance and algorithmic tricks, one can take full advantage of intrinsics to improve the performance significantly. For example, using approximations of costly functions like log and exp, when precision is not that important since small errors in the low-dimensional affinities are acceptable, can have very large benefits for other applications as well. Moreover, for very skinny matrices for which the number of columns is smaller than the vector size, one can modify the functions similar to matrix multiplications in an intelligent way. Working with symmetric/anti-symmetric matrices poses additional interesting challenges as it allows to reduce the number of loads/stores at the cost of additional complexity. When working in the scalar implementation, it is easier to deal with a single array but when coupling it with intrinsics, it becomes challenging to use that same format. The performance plot showed how our vectorized implementation **AVX2** reached the desired results compared to the baselines, but the roofline plot leaves with some potential improvements on the memory management side. In future works, this is exactly where someone should work forward. Moreover, it would also be interesting to work with a much clearer cost function, since in our analysis, we counted each floating point operation as the same. Clearly, an exp operation can be much costlier than a simple sum, hence this should also be accounted in the analysis. Lastly, tiling (both in cache and registers) is another optimization worth to look at.

Contributions

- **Virgilio.**: Implemented the Algorithmic version, both theoretically and practically, while also working on fixing the scalar version and maintaining-merging all the code and versions. Also responsible for measurement of flops and memory movement, both theoretically and empirically, created the benchmark infrastructure and all the plots. Wrote the Experimental, Theoretical Analysis, Setup, getGrad, pseudocode and Figures part.

- **Aurelio.**: Implemented both the scalar and vectorized baselines from which we improved on, getQ, getDistances, getYDistances in the final implementation and written their respective sections, abstract section, parts of result and conclusion section. Also set up the testing functions and worked with Intel Advisor in order to improve the code.

- **Ahash.**: Implemented the unrolled baseline based on scaler baseline, getP in the final implementation and written its respective section, introduction section and proof-reading.

- **Yuchen.**: Worked on block and unrolling optimizations in the scalar implementations and getGrad function in the final implementation. Wrote the respective sections and the background section.

8. REFERENCES

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

[2] Qingzhou Yue, "t-sne c++ implementation," 2024.

[3] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges, "The mnist database of handwritten digits," 1998, Accessed: 21-06-2024.

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

[5] George C Linderman, Manas Rachh, Jeremy G Hoskins, Stefan Steinerberger, and Yuval Kluger, "Fast interpolation-based t-SNE for improved visualization of single-cell RNA-seq data," *Nat Methods*, vol. 16, no. 3, pp. 243–245, Feb. 2019.

[6] Francesco Crecchi, Cyril de Bodt, Michel Verleysen, John A. Lee, and Davide Bacciu, "Perplexity-free parametric t-sne," 2020.

[7] Laurens van der Maaten and Geoffrey Hinton, "Visualizing data using t-sne," 2008, Journal of Machine Learning Research 9:2579-2605, 2008.

[8] Innovative Computing Laboratory, "PAPI: Performance Application Programming Interface," `https://icl.utk.edu/papi/`, Accessed: 21-06-2024.

[9] John D. McCalpin, "STREAM Benchmark," `https://www.cs.virginia.edu/stream/`, Accessed: 21-06-2024.