OPTIMIZING T-SNE FOR THE ZEN3 ARCHITECTURE

A. Abdrado, N. Baumann, D. Koutsoukos, B. Wu

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

The abundance of data produced everyday has lead to a proliferation of techniques to analyze and visualize them. One of the most prominent techniques for discovering patterns and visualizing data in low dimensional spaces is t-distributed Stochastic Neighbor Embedding (t-SNE). However, as usually the input data dimension is large and also the algorithm requires heavy computation to converge to a meaningful solution, optimizing t-SNE at the implementation level is crucial. In this work, we start by analyzing the performance of a vanilla t-SNE implementation and we split the algorithm into 4 main computation kernels. We then identify and resolve most of the bottlenecks of the original implementation by optimizing memory access patterns, approximating computation-heavy functions and also incorporating a large class of known optimization tricks (scalar replacement, vectorization). Our final implementation achieves a 3-5x speedup compared to the original implementation on the latest AMD Zen3 architecture.

1. INTRODUCTION

Data analysis has become one of the de-facto domains in every university and company nowadays. This was motivated both from the increasing amount of data that are produced everyday, especially in the web, and also from the advances in the field of Artificial Intelligence (AI), which made extracting insights and making predictions on a large scale easy and applicable without any specialized knowledge and/or tools. The first step in every data analysis pipeline is to visualize the data in order to get insights before applying preprocessing and/or Machine Learning (ML) algorithms for making predictions. This is usually done indirectly either with embedding the data into a lower dimensional space using Principal Component Analysis [1] and subsequently visualizing them, or by applying directly an algorithm that was designed for visualizing data into low dimensional spaces like t-SNE [2].

However, as the input dimensionality of most datasets today is very large [3], reducing it for visualization purposes is often a non-trivial task computationally. Especially in the case of t-SNE, where the algorithm has to perform a binary search to fit a Gaussian to the input points, the implementation of the algorithm can make a significant difference to the total runtime, depending on the optimizations performed.

To this end, we are focusing on performing hand-tuned optimizations on a vanilla t-SNE implementation [2] for 2 dimensions, in order to achieve the best performance possible. We explicitly choose this output dimensionality as it is the one that leads to easier interpretation of the final results. To do so, we first analyze the original implementation to identify the main bottlenecks.

We then apply standard optimizations that are known to increase performance (e.g. scalar replacement, vectorization, arithmetic transformations). Based on an initial evaluation, we subsequently optimize the memory layout access patterns and we also approximate computational-heavy functions to further decrease the runtime. We finally adjust the codebase to the processor characteristics of the latest AMD Zen3 architecture, which is the first in a decade that is faster than Intel in single- & multicore performance [4]. Our final implementation achieves a 3-5x speedup, depending on the algorithmic part. We validate our results by applying a pen-and-paper performance analysis and we observe that we reach in most of the cases what the theoretical maximum should be. Our final results confirm that our t-SNE implementation presents meaningful dimensionality reduction visualizations that come at a improved total runtime compared to the original implementation, making the algorithm suitable to real world datasets.

Related work t-SNE [2] has been proposed to visualize high dimensional data in scatter plots in 2 or 3 dimensions. However, as the original proposal is computation-heavy, the authors came up with an improved version [5] that accelerates the algorithm using tree-based methods. Although, there have been attempts to improve t-SNE at the algorithm level [5] and also how to use it effectively [6], to our knowledge no efforts have been made to improve the implementation for performance reasons, especially on the AMD Zen3 architecture [7].

2. BACKGROUND

In this section we explain how the t-SNE algorithm works and we analyze its basic steps. Afterwards, we split the algorithm into 4 main computation functions (which we call *kernels*) and finally we perform a cost analysis per kernel.

The algorithm starts by computing the similarity p_{ij} between two D-dimensional points x_i, x_j according to a probability density under a Gaussian centered at x_i using the following formula: $p_{ij} = \frac{p_{j|i}+p_{i|j}}{2N}$, where:

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-\|\mathbf{x}_i - \mathbf{x}_k\|^2/2\sigma_i^2)} (1)$$

, $p_{i|j}$ is defined in an analogous manner and N is the number of input points. Note that $p_{i|i} = 0$ and $\sum_j p_{j|i} = 1$ for all i. The variance σ^2 depends on the per-point Gaussian and the number of points surrounding the center of the distribution. The algorithm binary searches the variance that gives the user-defined perplexity. The formula of the perplexity is:

$$Perp(P_i) = 2^{-\sum p_{j|i}\log_2 p_{j|i}}$$

where the exponent is the Shannon entropy.

After t-SNE has computed the similarities between the input points it creates a low-dimensional space K (where K is usually 2 or 3) where it spreads the points randomly. At this step, the algorithm tries to preserve the similaries p_{ij} as much as possible. To do that, it measures again the similarities of two output points y_i, y_j using the following formula:

$$q_{ij} = \frac{(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2)^{-1}}{\sum_k \sum_{l\neq k}(1 + \|\mathbf{y}_k - \mathbf{y}_l\|^2)^{-1}}$$

where $q_{ii} = 0$. This is similar to the first step but instead of a Gaussian, the algorithm uses a Student t-distribution with a single degree of freedom.

In the final step of the algorithm, t-SNE embedds the points y_i to the low dimensional space by minimizing the KL-divergence of the input distribution P and the output distribution Q using Stochastic Gradient Descent. The formula of the KL-divergence is:

$$\text{KL}(P\|Q) = \sum_{i\neq j} p_{ij}\log\frac{p_{ij}}{q_{ij}}$$

The computational and memory complexity of the algorithm is $\mathcal{O}(n^2)$. Finally, the points should have a zero mean per dimension, for convergence of the algorithm.

Based on the above description and as we have to calculate the Euclidean distance between each pair of points, we identify 4 kernels which together constitute the basis of the t-SNE algorithm: 1) `zeroMean`, which centers the data such that average of each feature is zero. 2) `computeGP`, which computes the probability density of the points by fitting a Gaussian and also performs the binary search for the user-defined perplexity. 3) `updateGradient`, which performs SGD to minimize the KL-divergence and updates the output $\mathbf{Y}$. 4) `computeSED` computes the Squared Euclidean Distance of any given set of points. In the rest of the section, we perform a cost analysis per kernel for the original implementation done by the authors [8].

ZeroMean. This kernel centers the data such that the average of each feature, or column in $\mathbf{X}$, is zero. It does one pass to compute the means at cost ND additions + D divisions, and then a second pass to subtract the means from the columns, cost ND. Flop count in total is $2ND + D$, where D is input dimension.

ComputeGP. This kernel contains both simple mathematical operations and complex functions(e.g. $\log, \exp$). We count simple mathematical operations as 1 flop, and we execute microbenchmarks to get the equivalence for the complex functions. The total flop count is $200N(21N + 27) + 3N^3$.

ComputeSED. As this kernel only contains additions and multiplications, the flops count is already a good measure of cost. Since each Euclidean distance calculation costs $3D - 1$ flops, where D is the input dimension, and there are "N chooses 2" unique pairs of data, the total flop count is hence $5N(N-1)/2$.

UpdateGradient. This kernel contains `computeSED` inside of it, so the cost of it is dependent. The total flop count is $12N^2$.

3. METHOD

In this section, we delve into implementation details for each one of the 4 kernels and we explain in detail the optimizations performed.

3.1. ZeroMean

In its basic form, `zeroMean` has 3 phases. It first accumulates the sum of each input feature, where inputs are stored in a $N \times D$ matrix $\mathbf{X}$, which is stored in row-major order. At the end of this operation, $\mathbf{mean}_d = \sum_{i=1}^{N} \mathbf{X}_{id}$. It then divides the sums calculated by the number of observations: $\mathbf{mean}_d = \mathbf{mean}_d/N$. Finally, it subtracts the averages calculated in the previous step from the features of each observation.

Blocking Optimization. Recall that the first phase of `zeroMean` consists of summing all columns of $\mathbf{X}$, which is done using two for-loops. We optimize this block of code by unrolling the outermost loop which iterates over the observations, and the innermost loop which iterates over the elements of array $\mathbf{mean}$. On the Zen3 micro-architecture, the inner most loop should have $\lceil \text{add TP} \cdot \text{add latency} \rceil = 6$

Fig. 1: ACC0, ACC1, ..., ACC5 act as one large vector that accumulates values in X in jumps of 24 doubles

accumulators. For the outermost for-loop, there is a memory trade-off: if we use a too large an unrolling factor, we risk having capacity misses for the instructions in the inner most loop body. On the other hand, if we use a small unrolling factor then many array accesses for **mean** are incurred as there is less scalar replacement. Empirically, using an unrolling factor of 4 for the outer loop offers good speedup. For the third phase, we use the optimization of accessing 6 observations per outer iteration. A possible setback to the blocked approach is that the innermost loop, which passes over the array **mean**, requires an unrolling factor of 6. Thus, for values of D smaller than 6, instead of unrolling the innermost loop that passes over **mean**, we should unroll the outer loop such that 6 addition/subtraction are done inside the inner most loop body. For real data though, the input has high dimension and thus the problem is avoided.

Vectorization $(D = 2)$**.** For the first phase we also use 6 accumulators; however, now the accumulators are AVX2 vectors. As we see in Fig. 1, ACCi aggregates all observations x_i such that i mod 12 $\in \{2i, 2i+1\}$. We then add all 6 accumulators (plus residuals) into a vector, called S whose elements, from low to high address, are S0, S1, etc. We ensure those values are divided by N. Observe that S0 and S2 must be subtracted from the first feature column, and S1 and S3 must be subtracted from the second column. A simple solution would be as in Fig. 2 where we subtract in jumps of 2 doubles. However, that would result in unaligned loads and hence a slow runtime. Of course, the first 2 elements are handled separately. To align accesses we observe that the operation can be done as in Fig. 3. Thus, we create a vector that stores the values (S0+S2, S1+S3, S0+S2, S1+S3) and subtract this vector from each interval of X of length 4, or from each interval of length 24 since we improve ILP by using 6 accumulators.

The idea is easily extended for 3 dimensions; however, to avoid unaligned accesses we must read a multiple of 12 values in **X** per iteration as 12 = GCD(4, 3). Instead of having one vector, we have three vectors where the first vector

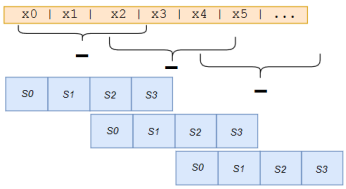

Fig. 2: Simple zeroMean vectorization

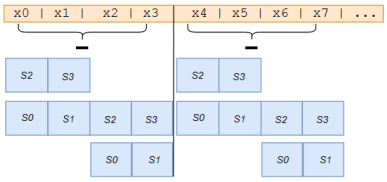

Fig. 3: Aligned zeroMean vectorization

stores (from low to high address) (S0, S1, S2, S0), the second vector stores (S1, S2, S0, S1), and the third vector stores (S2, S0, S1, S2) where Si = sum of i-th column/N. Likewise, we process 24 values of **X** per iteration and subtract these three vectors from each half segment.

3.2. ComputeGP

Kernel analysis. The computeGP kernel starts by using the computeSED to compute the Euclidean distances. It then has a double for-loop. The outer loop iterates through all the points. The inner one performs a binary search to find the best variance candidate for a specific point, by optimizing the Shannon entropy and it consists of a number of smaller loops that: 1) compute the Gaussian kernel per row 2) compute the entropy per row. If the entropy is within a user-defined tolerance, the inner loop , otherwise the binary search continues. After the inner loop finishes, the outer loop continues. It row-normalizes the similarity between the points, which is stored in matrix P. Finally, it iterates again through all the N points and dimensions to symmetrize the input similarities, according to the t-SNE algorithm.

As follows by the description above, computeGP is the largest and the most computation-heavy kernel in the t-SNE implementation. Thus, in the rest of the section we explain what optimizations we performed and their potential impact on the final performance. Note that since all the matrices are stored in major-row order, no memory optimizations are available in this kernel. Finally, since the binary search does not always converge in the same number of steps, we fixed the number of iterations to 200. That helps in evaluating the impact of the various optimizations and it also provides a meaningful visualization in the embedded space. Regarding the operational intensity, we perform microbenchmarks to count the complex functions. For memory accesses, we read only two N^2 matrices and we count only compulsory misses. Therefore $I(N) = \frac{200N(21N+27)+3N^3}{16N^2}$ The N^3 term comes in contrast to the complexity of the algorith, but it is present in the original implementation of the authors.

Basic optimizations. As the outer loop is already quite large, and its code fills the processor pipeline, we did not observe any additional performance benefits by unrolling it. We instead focused on unrolling the inner loop. For each one of the small loops, mentioned in the previous para-

graph, we increased the ILP by first unrolling them 4 times. Depending on the calculation after unrolling we either put intermediate results into separate accumulators or we put `#pragma ivdep` instructions such that the compiler knows that no aliasing is present. We chose to unroll each loop 4 times empirically, as unrolling the loops 8 or more times did not increase performance further. Afterwards, we converted all the divisions that were present in the loop to multiplications. This was done either by simple mathematical transformations (e.g. converting $\frac{1}{2}$ to 0.5) or by assigning intermediate variables that contained loop constants (e.g. instead of dividing with `sum_P`, we precalculated `1/sum_P` and we multiplied with it). Lastly, we precalculated all other constants that were inside loops.

Vectorization. Besides the above optimizations, the performance was marginally increased. To understand the cause, we profiled the code and we saw that the exponential function, which is necessary to fit the Gaussian into the individual points was very computational intensive. There was also a logarithm function which was necessary to compute the entropy. However, the exponential was computed many times per binary search iteration and therefore it was the main bottleneck of the computation. To evaluate the complexity of these two functions, we run microbenchmarks that showed that $\log$ is 10x slower and $\exp$ is 15x slower than a multiplication in the Z3 architecture. Since we already planned to vectorize the code, we also decided to approximate the exponential. For that we used the library developed by Agner Fog [9], which calculates the $\exp$ using a Chebyshev approximation. Together with vectorization, this made a huge performance difference, leading to a 5x speedup from the original implementation. However, this comes at a price. Although the algorithm still converges, it does to a slightly different embedded space than the original code. That happens because the similarities are in the order of 10^{-18} and therefore even small calculation differences can have a large impact on the result. Lastly, the vectorized version includes all the optimizations of the previous version that were possible (e.g. increase the ILP, loop unrolling, mathematical transformations).

Compiler flags. Finally, because the kernel has many mathematical operations, we decided to experiment with compiler flags. The one that was most relevant to this kernel was `-ffast-math`, which also approximates functions. The performance benefit from using this flag was minimal ($\sim 5\%$) but we have included it, both in the most optimized scalar version and also in the vectorized version.

3.3. UpdateGradient

`updateGradient` is calculated after every iteration, which means any improvements in the performance will strongly impact the overall runtime. The kernel consists of 4 smaller steps. It first computes the `computeSED` of the output data point $\mathbf{Y}$ and saves them into $N \times N$ matrix $\mathbf{DD}$. Afterwards, it transforms the data of $\mathbf{DD}$ to the desired inverse square root. Then, it takes in the precomputed similarity of the data points $\mathbf{X}$ from `computeGP` and computes the gradient. Lastly, it updates the output data points $\mathbf{Y}$.

For performance measurements we do not consider the runtime of the first step. Section 3.4 covers `computeSED` in detail. The matrix $\mathbf{DD}$ will be precomputed and loaded in like matrix $\mathbf{P}$ from `computeGP`.

As mentioned, the flop count is $12N^2$. In our vectorized version the flop count has an additional N term, as the diagonal entries of the matrices will be ignored when calculating the gradient. Additionally, the operation intensity can change depending on if matrices/vectors are preloaded from the previous steps. Finally this kernel reads matrices once and in row-major order. Therefore, the operational intensity is:

$$I(N) = \begin{cases} \frac{12N^2}{8*(3N^2+8N)} \leq \frac{1}{2} = 0.5 & \text{, if } N \lessapprox 1414 \\ \frac{12N^2}{8*(4N^2+8N)} \leq \frac{3}{8} = 0.375 & \text{, if } N \gtrapprox 1414 \\ \frac{12N^2}{8*(6N^2+6N)} \leq \frac{1}{4} = 0.25 & \text{, if } N \gtrapprox 2'000'000 \end{cases}. \tag{1}$$

Scalar Optimizations. In this version, we first save the total sum of matrix $\mathbf{DD}$ as an inverse constant. We also unroll the inner for-loop that calculates the gradient 4 times.

Vectorization. We vectorize the code using AVX/AVX2 instructions. We fuse operations FMA/FMSUB instructions. To use the FMSUB we make a simple tranformation as shown in Equation 2.

$$\begin{aligned}(A - B * C) * (D - E) \equiv (B * C - A) * (E - D) \\ \Rightarrow fmsub(B, C, A) * sub(E, D)\end{aligned} \tag{2}$$

Upper Triangular Matrix. We also only considered the upper triangular part of all the matrices, because all matrices in t-SNE are symmetric. This means that we can improve the access pattern in `computeSED`, `computeGP` and `updateGradient`. We first applied this optimization to the `updateGradient` kernel. When computing the gradient, we need to read the whole matrix in row-wise order. However, as we only have the upper triangular part of the matrix $\mathbf{DD}$, the algorithm reads every row in an $\mathbf{L}$-shape (see Figure 4). We additionally vectorize this implementation.

3.4. ComputeSED

`computeSED` takes a $N \times D$ matrix $\mathbf{X}$ as input and outputs a $N \times N$ matrix $\mathbf{DD}$, where $\mathbf{DD}_{i,j} = \sum_{d=0}^{D-1}(\mathbf{X}_{i,d} - \mathbf{X}_{j,d})^2$. Based on this definition, $\mathbf{DD}$ is a symmetric matrix. In the above, N is the number of data points and D is the dimension.

There are at least two ways to compute $\mathbf{DD}$. 1) Compute the Euclidean distance for each pair of data points.

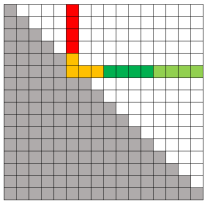

Fig. 4: Sketch of the matrix $\mathbf{DD}$ with only the upper triangle being saved and read. We show in color how row 6 would be read in this format.

Then, the total flops count is $C_1 = \frac{(3D-1)N(N-1)}{2}$. 2) Precompute the 2-norm for each data point. Then, for any two data points x and y, we simply compute $x^T x + y^T y - 2x^T y$. The total flop count is $C_2 = N(N-1)(D+1) + N(2D-1)$.

In our case, since $D = 2$, we always have $C_2 > C_1$. Therefore, for optimization purposes, since the second method exhibits the same memory access pattern as the first one, we chose the first method for a lower number of flops.

Baseline. The baseline implementation calculates $\mathbf{DD}_{ij}$ with $\mathbf{X}_i$ and $\mathbf{X}_j$ and copies it to $\mathbf{DD}_{ji}$ for every $i < j$. The operational intensity is $I_b = \frac{5N-5}{88N+216} \leq \frac{5}{88} = 0.0568$. We ignore data movement from writing $\mathbf{DD}$ to memory because in reality the write-back only happens when the cache line needs to be evicted. In general, we cannot know how much data is actually flushed to memory. We are interested in an upper bound of the operational intensity for roofline model analysis.

Mini-blocking. The baseline implementation has two major drawbacks. First, accessing every $\mathbf{DD}_{ji}$ will cause a cache miss since there is no spatial locality. Second, accessing data point $\mathbf{X}_k$ does not benefit from temporal locality most of the time. To fill in the row we load $\mathbf{DD}_{i,i:N}$, $\mathbf{X}_k$ $(k > i+1)$ into L1 cache. Since L1 is small, $\mathbf{X}_k$ is likely to be evicted after use and re-loaded into L1 cache for row $\mathbf{DD}_{i+1,i+1:N}$. This leads to substantial L1 read misses. To remedy the above, we adopted the blocking technique, which transforms the computation to achieve better cache utilization. The goal is to reduce the L1 cache read and write misses.

Let's take a block size b. For any two blocks $\mathbf{X}_i^b$ and $\mathbf{X}_j^b$ $(i \leq j)$, we compute all pairs of Euclidean distances and write to $\mathbf{DD}_{i,j}^b$ and its symmetric block $\mathbf{DD}_{j,i}^b$. This pattern accesses $\mathbf{X}$ with temporal and spatial locality and $\mathbf{DD}$ with spatial locality. To determine the optimal b, we employ a working-set analysis. A working set at any time involves $\mathbf{X}_i^b$, $\mathbf{X}_j^b$, $\mathbf{DD}_{i,j}^b$ and $\mathbf{DD}_{j,i}^b$, in total $2Db + 2b^2$ doubles. We fit the working set in the L1 cache, since it has significantly lower access latency L2/L3. Thus, we want a b such that: $(2Db + 2b^2) \times 8$ Bi ≤ 32 KBi. When $D = 2$, $b_{opt} = 32$ (rounded to the smaller lower power of 2).

Since this new scheme significantly changes the data movement pattern, we develop a new operational intensity analysis. With a write-back/write-allocate cache, the total input size is far larger than the L3 cache. To prepare a $b \times b$ block $\mathbf{DD}_{i,j}^b$ in $\mathbf{DD}$, we have two cases. 1) $i = j$. We only need to write into cache $\mathbf{DD}_{i,j}^b$ and load $\mathbf{X}_i^b$ from memory to cache. This will cause $8b^2$-bytes writes and $16b$-bytes reads. 2) $i < j$. We need to write into cache $\mathbf{DD}_{i,j}^b$ and $\mathbf{DD}_{j,i}^b$ and load $\mathbf{X}_j^b$ from memory into cache. Notice that according to the LRU replacement policy, $\mathbf{X}_i^b$ will be kept in cache until its associated computation is finished. This will cause $16b^2$-bytes writes and $16b$-bytes reads.

Therefore, we have $\frac{N}{b}$ case-1 data movement and $(\frac{N}{2b} - \frac{1}{2})\frac{N}{b}$ case-2 data movement. Thus, we write $\frac{N}{b} \times 8b^2 + (\frac{N}{2b} - \frac{1}{2})\frac{N}{b} \times 16b^2 = 8N^2$ bytes into memory and read $\frac{N}{b} \times 16b + (\frac{N}{2b} - \frac{1}{2})\frac{N}{b} \times 16b = \frac{8N^2}{b} + 8N$ bytes from memory. Again, if we ignore writing back to memory, the total data movement is $8N^2 + \frac{8N^2}{b} + 8N$ bytes. With the above information, we can derive the operational intensity as:

$$I_{l1} = \frac{5\binom{N}{2}}{8N^2 + \frac{8N^2}{b} + 8N} \overset{\text{b=32}}{=} \frac{10N - 10}{33N + 32} \leq \frac{10}{33} \approx 0.303$$

Ideally, this mini-blocking implementation should reduce L1 read and write misses. However, it turns out that this implementation does not improve L1 write misses when $N \geq 256$. The reason for this is cache organization. Our L1 cache is 32 KB and 8-way associative, which implies a 4 KB set. Therefore every 512 doubles will be mapped to the same set. Together with 8 way-associativity, one conflict miss will happen for accessing every 4096 doubles. Take $N = 512$ as an example. $\mathbf{DD}_{j,i}$,...,$\mathbf{DD}_{j+7,i}$ will be mapped to the same set but different ways. Afterwards, accessing $\mathbf{DD}_{j+8,i}$ will evict $\mathbf{DD}_{j,i}$ and when we access $\mathbf{DD}_{j,i+1}$ (same cache line as $\mathbf{DD}_{j,i}$), we need to reload it into the L1 cache. From Table. 1, it is clear to see that miniblock and baseline have almost the same number of L1 write misses. As a result, this cache organization will eradicate all the efforts for accessing $\mathbf{DD}$ in a cache-friendly manner if we do not deal with it carefully.

Mini-blocking+Buffering. To make blocking useful for reducing L1 write cache misses, we introduce the buffering technique. Instead of directly writing into block $\mathbf{DD}_{j,i}^b$, we define and write into a reusable 32×32 buffer array. Later, we flush the buffer to $\mathbf{DD}_{j,i}^b$ *row by row* for efficient cache accesses. With buffering, the L1 write misses drops by a factor of 4 as shown in Table. 1.

Micro-blocking. Recall that the lack of temporal locality for reading the data matrix $\mathbf{X}$ accounts for the inefficiency of the baseline. Therefore in this section, we aim to reduce the load instructions for $\mathbf{X}$ in addition to cache misses. The micro-blocking implementation builds on top of mini-blocking but transforms the computation inside the 32×32 block such that each data point stays in the regis-

ters as long as possible and hence reducing the load instructions. Inside the mini-block, we carry out the computation in a 4×1 micro-block manner. As a concrete example, we pre-load $\mathbf{X}_i, ..., \mathbf{X}_{i+3}$ into the registers (by scalar replacement), and calculate the Euclidean distances with $\mathbf{X}_j$ in a 4-way manner. To see how this scheme reduces the number of load instructions, for each $\mathbf{X}_j$, without micro-blocking, we have to load it for each i, which in total is 32 times. With micro-blocking, each $\mathbf{X}_j$, is loaded only $32/4 = 8$ times. We verify this by the results in Table. 1 as micro-block only executes one third of load instructions as mini-block. Moreover, this re-arrangement facilitates the vectorization because vectorizing one micro-block of computation is very straightforward. Finally, it is worth mentioning that we can combine micro-blocking with the buffering technique to gain additional performance in some cases.

Vectorization. As mentioned before, vector code can be easily developed for micro-blocks. First, we load the first dimension of $\mathbf{X}_i, ..., \mathbf{X}_{i+3}$ into vector x_0 and the second dimension into vector x_1. Then we compose two vectors y_0 and y_1, each holding 4 duplicates of $\mathbf{X}_{j,0}$ and $\mathbf{X}_{j,1}$ respectively. Lastly, we calculate the Euclidean distances in a 4-way manner by $(x_0 - y_0)^2 + (x_1 - y_1)^2$. and we store the result into $\mathbf{DD}$ directly or into the buffer.

3.5. Combining Kernels

After optimizing each kernel separately, we explored if fusing kernels offers additional performance. To verify our hypothesis, we combined the `computeSED`, `updateGradient` and `zeroMean` kernels, because they are called consecutively after every iteration. We call this kernel `updateGradient_zeroMean`. The first optimization that we performed was to combine step 1. and 2. from section 3.3. Instead of first calculating `computeSED` and then writing into a new matrix the product $x \leftarrow 1.0/(1.0 + x)$, we can directly save it into the matrix $\mathbf{DD}$, together with calculating its total sum. This both saves N^2 of storage and a N^2 for-loop. Another optimization we applied was to combine step 1. of section 3.1 with step 4. of section 3.3. More specifically we are using the same for-loop, both to update the output data $\mathbf{Y}$ and to calculate the `mean`. This eliminates another $2N$ for-loop. We present the performance cost in the next equation: $W(N) = W_{sed}(N) + W_{upGr}(N) + W_{zM}(N) = \frac{5}{2}N(N-1) + 12N^2 + 4N + 2 \approx 14.5N^2 + 1.5N$. Additionally, the number of bytes transferred changes to: $Q(N) = Q_{sed}(N) + Q_{upGr}(N) + Q_{zM}(N) = (8N^2 + \frac{8}{32}N^2 + 8N) + (2N^2 + 8N)8 + 0 \approx 24.25N^2 + 72N$. $Q_{zM}(N) = 0$ because the output data $\mathbf{Y}$ is already loaded from the last step of `updateGradient`. Hence we have an operation intensity of

$$I(N) = \frac{14.5N^2 + 1.5N}{24.25N^2 + 72N} \leq \frac{14.5}{24.25} \approx 0.598. \quad (3)$$

4. EXPERIMENTAL RESULTS

In this section, we evaluate how well our optimizations perform compared to the original codebase. We also perform a roofline analysis. We study how input size affects performance, verify our previous theoretical analysis and explain the empirical results.

Experimental setup. For the experiments we use an AMD Ryzen 5950X CPU. It uses the Zen3 micro-architecture and has a main frequency of 3.4 GHz. For the memory hierarchy, each core has a 8-way 32 KB L1 data cache, a 8-way 512 KB L2 cache and 16-way 32 MB L3 cache. All caches uses a write-back/write-allocate policy. As each core has 2 FMA units and 2 FP add/sub units, the peak performance of one core is 6 flops/cycle. With AVX2 vectorization, the peak performance can reach 24 flops/cycle. The tool `bandwidth` [10] gave a measurement of memory bandwidth equal to 6.474 bytes/cycle. This implies the ridge points $(0.927$ flops/byte$, 6$ flops/cycle$)$ for scalar code and $(3.707$ flops/byte$, 24$ flops/cycle$)$ for vector code. Additionally we use g++ 10.2.0 with flags `O3`, `std=c++17`, `ffast-math`, `march=native`, `mavx2`, `mfma` and `mtune=native`. For the scalar implementations, we also use `fno-tree-vectorize` to disable compiler-generated vectorized code. To ensure we are measuring performance for a single core and avoid context switching, we pin the process to a core by using `taskset` and we make sure no other processes are running on the same core.

Results. First, we show the results for each kernel separately and we finally demonstrate the performance effect on combining kernels.

	L1 load miss	Total load	L1 write miss	L3 write miss
Baseline	33,311,730	536,838,149	150,994,922	150,765,568
Mini-block	1,879,373	282,708,228	150,766,066	33,555,469
Mini-block+buf	2,562,331	316,854,791	35,177,462	33,556,819
Micro-blocking	1,470,574	81,520,385	50,304,832	33,554,577
Micro-block+buf	2,682,798	118,409,479	35,385,494	33,556,711

Table 1: Cache behavior statistics for `computeSED` kernel when $N = 16384$.

ZeroMean. We show the performance results for `zeroMean` in Figure 5. For small input sizes ($N < 2^7$), there is a considerable overhead for the vectorized implementation; hence, the speedup in comparison to the other two methods is small. The graph confirms the speedup decay we observed in Table 3, where the memory bandwidth of the L3 cache and the memory limit the attainable performance. This suggests that the operation is memory bound and the operational intensity is below the ridge of the processor. Indeed, one finds $I(n) = \frac{2ND+D}{8(ND+D)} \leq \frac{1}{4}$ flop/byte, which is less than the ridge 0.93 in the roofline plot of figure 10. For this simple kernel, we also studied how compiler flags affect the performance. The blocked version was tested with $D = 12$ to increase the ILP. The results on a Ryzen 5800H (Zen3)

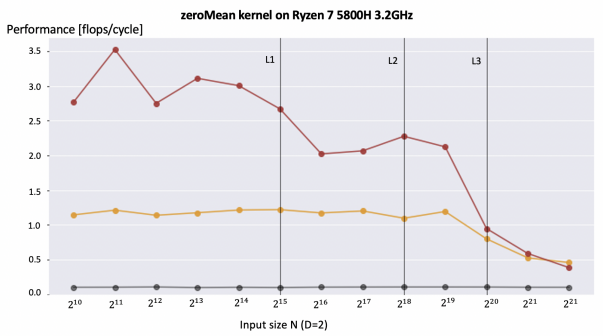

Fig. 5: Performance plot of the kernel `zeroMean`. Base implementation with `-fno-tree-vectorize` (black), blocking optimization (all flags) (orange) , AVX2 with `-O3` flag (red).

Length	Basic	Blocked (no flags)	Blocked (O3)	Blocked (all flags)
1000 (L1)	595,418	241,217 (2.4)	60,115 (9.9)	20,707 (28.75)
10,000 (L2)	6,277,946	3,228,522 (1.9)	562,374 (11.2)	205,976 (30.5)
100,000 (L3)	62,459,792	38,057,592 (1.6)	4,820,212 (12.9)	2,157,520 (28.9)
175,762 (RAM)	104,008,128	44,878,272 (2.3)	12,409,692 (8.4)	5,293,266 (19.6)

Table 2: Runtime (cycles) and speedups w.r.t basic implementation (`-O0`) for $D = 12$. Scalar version.

are shown in Table 2. Compared to the baseline with no compiler flags (`fno-tree-vectorize`), using all compiler flags(`mtune=native,O3, mfma, ffast-math, fno-tree-vectorize`) is 20-30 times faster. For the vectorized version, we get the results shown in Table 3. Despite computations being faster for the AVX2 version, the higher latency memory transfer for larger input sizes results in a decreasing speedup, and a performance convergence below that of the blocking implementation.

ComputeGP. We show the results of the optimization of the `computeGP` kernel in Figure 6. As we can observe in the diagram, the kernel is clearly compute-bound. We run only up to input size of 1024, because for larger input size the kernel does more than 15 hours to converge. The baseline version achieves a performance of around 0.8 flops/cycle for small input sizes and 1.3 flops/cycle for larger input sizes. The scalar optimized version marginally improves that by having a performance of around 1 flop/cycle for small input sizes and 1.6 flops/cycle for larger input sizes. However, the optimized version that approximates the exponential starts with a performance of 2.2 flops/cycle and goes up to 5.6 flops/cycle for input size of 1024.

UpdateGradient. We conduct experiments with data input sizes from 16 up to 2048. L1 cache is full with $N = 35$, L2 cache with $N = 144$ and for L3 $N = 1153$. The baseline implementation (blue) only reaches about ~ 0.4 flops/cycle, where the non-SIMD implementation (orange) reaches ~ 0.8 flops/cycle. The SIMD implementation (red) reaches a max performance of ~ 2.4 flops/cycle. The experimental implementation that only considers upper triangular matrices (green) performs well for small input sizes but it is

Length	Basic	Vectorized (no flags)	Vectorized (O3)	Vectorized (all flags)
10,000 (L1)	491,016	109,563 (4.5)	15,530 (31.6)	15,045 (32.6)
100,000 (L2)	4,929,208	1,095,362 (4.5)	178,539 (27.6)	181,585 (27.1)
1,000,000 (L3)	52,554,041	12,931,664 (4.1)	4,659,946 (11.3)	5,227,015 (10.1)
1,049,576 (RAM)	51,832,819	13,726,845 (3.7)	5,768,474 (8.9)	5,914,188 (8.8)

Table 3: Same as Table. 2 but vector version for D=2

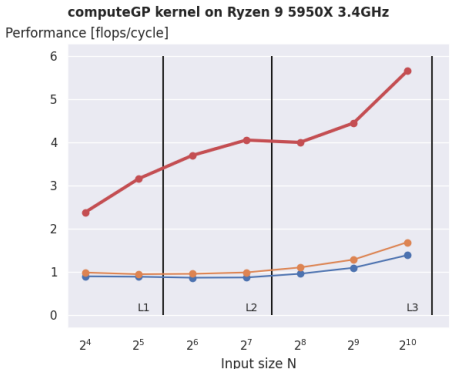

Fig. 6: Performance plot of the kernel `computeGP`. (blue) Base implementation, (orange) non-SIMD optimization, (red) SIMD.

slower than the base implementation for larger input sizes. This is because reading data in column wise direction is very costly, and one of the weakness of Zen3 are load/stores. This becomes more eminent when we read data in column direction. We have validated the latter with VTune [11].

ComputeSED. Fig. 8 shows the performance plot for `computeSED` kernels. It contains results for 7 different implementations over various input sizes. We would like to highlight some interesting information from the plot. First, all optimized scalar implementations achieve better performance than the baseline except when the input fits in L1 cache entirely ($N = 64$). This is because optimized versions, at this size, only add overhead for more instructions, loops structures and buffers. Moreover, memory hierarchy manifests through the step-wise trends of curves. Interestingly, a more sheer plummet of performance is observed when the input size exceeds L2 than L3. This is due to the fact that Zen3 has a much larger access latency to L3 (~ 46 cycles) than L2 (~ 12 cycles). Second, implementations with buffers perform better than those without buffers when N grows larger than or equal to 256. This can be attributed to the cache thrashing issue which starts to happen at $N = 256$ and peaks at $N = 512$ and beyond. This explains the out-performance as well as why the performance of implementations without buffers becomes almost flat beyond $N = 512$. Third, vectorization without buffering is even worse than mini-blocking and micro-blocking when the input size goes beyond the L2 cache. The performance of implementations without buffering is seriously capped by excessive write misses that constantly fall back to L3 cache when the input size goes beyond L2 cache. In addition, vector code introduces additional overhead, e.g., permuta-

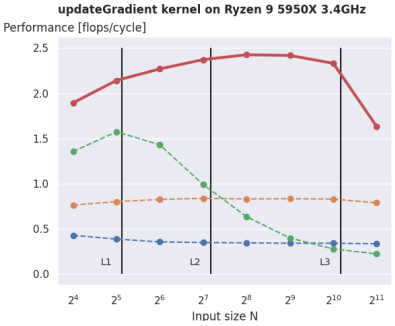

Fig. 7: Performance plot of the kernel `updateGradient`. (blue) Base implementation, (orange) non-SIMD optimization, (red) SIMD, (green) upper triangular matrices.

tion and unpacking, which makes it less performant than the scalar code. Finally, we put the result into the context of roofline model. With the operational intensity given in 3.4, we derive the peak performance to be $0.303 \times 6.464 = 1.96$ flops/cycle. Note that this analysis ignores the writes to the memory, which accounts for almost half of the data movement. If we include the writes, the operational intensity becomes 0.1538 flops/byte, which implies a peak performance of 0.996 flops/cycle. Therefore, we conclude that our best optimization achieves around $45.3 - 89.3\%$ of the theoretical peak performance.

UpdateGradient_zeroMean. We present the performance of the kernel `updateGradient_zeroMean` in Figure 9. Because we use the same matrices/vectors as in `updateGradient` kernel, we still have the same L1, L2, and L3 input size limits. However, the performance peaks are slightly different from Figure 7. The non-SIMD implementation (orange) now has a peak performance of $\sim$ 1.8 flops/cycle and the SIMD a peak performance of $\sim$ 2.8 flops/cycle. When the input size is very small ($N = 16$) then the upper triangular matrix implementation outperforms the SIMD implementation. This is because all the matrices are all load L1 cache and we additionally access a very small number of elements, which is reduced even more by accessing only half the matrix. However, when N increases the upper triangular matrix implementation is worse the base implementation, due to the column-wise accesses and the additional memory transfers.

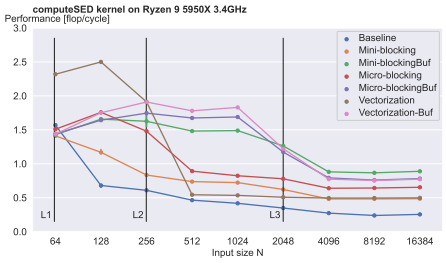

Fig. 8: Performance plot of the kernel `computeSED`.

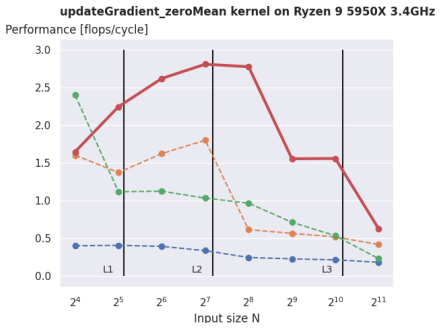

Fig. 9: Performance plot of the kernel `updateGradient_zeroMean`. (blue) Base implementation, (orange) non-SIMD optimization, (red) SIMD, (green) upper triangular matrices.

Roofline. Finally, we present the roofline model in Figure 10. The Figure contains all the kernels (except `computeGP`). We do not plot `computeGP` because its intensity is $I_{GP}(N = 2048) = 646.66$ after $N = 1024$ it needs more than 15 hours to converge. We predict the peak performance to be at least ≥ 5.8 flops/cycle. The ridge points got measured to be (0.927flops/byte, 6flops/cycle) for scalar code and (3.707flops/byte, 24flops/cycle) for vector code.

5. CONCLUSION

In this paper, we optimized the implementation of one of the most basic dimensionality reduction and visualization methods, t-SNE. After analyzing the original implementation done by the authors of the paper, we split the codebase into 4 computation functions, we analyze their performance and behaviour and we perform suitable optimizations to each one of them. Our final implementation achieves a 3-5x speedup, depending on the kernel and the input size, making t-SNE's runtime acceptable for large datasets used by data scientists today.

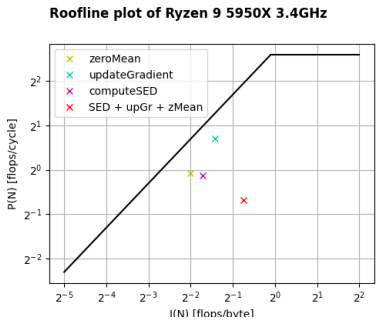

Fig. 10: Roofline model of the kernels `zeroMean`, `updateGradient`, `computeSED`.

6. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Amro. Focused on the optimization of the `zeroMean` kernel. Implemented basic optimizations such as strength reduction: changing division to multiplication, and did more involved optimizations such as blocking and vectorization. To optimize blocking he identified ideal unrolling factors and performed scalar replacement. For the vectorized version, he eliminated unaligned loads and improved ILP by using multiple accumulators. Did the performance plot for `zeroMean` and tested different compiler flags.

Noah. Focused on the optimization of the kernels `updateGradient` and `updateGradient_zeroMean`. Both non-SIMD & SIMD. As well as analytically derive cost and operation intensity of those two kernels. Experimenting in upper triangular matrices usage. Using the profiling tool VTune [11] to look for bottlenecks in all the kernels. Conducted the plots for `updateGradient`, `updateGradient_zeroMean` & roofline.

Dimitris. Focused on the optimization of the kernels that fits the Gaussian. Performed basic optimizations (e.g. scalar replacement, inlining). Experimented with compiler flags as the kernel involved heavy mathematical operations. Vectorized the code and approximated mathematical functions that were the bottleneck of the algorithm (this was identified using a profiler). Finally, developed a scratchpad memory to avoid stalls called by `malloc/calloc`.

Bowen. Focused on the optimization of computing pairwise Euclidean distance. This includes developing mini-blocking and micro-blocking, introducing buffering and vectorizing the code. Besides implementation work, theory work was conducted around the kernel for deriving the operational intensity, analyzing sub-optimal cache accesses and roofline analysis. Moreover, used valgrind to profile the cache behavior in order to substantiate the performance improvement. Researched the Zen3 micro-architecture for peak performance and cache hierarchy. Measured the machine's memory bandwidth.

7. REFERENCES

[1] Christopher M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer-Verlag, Berlin, Heidelberg, 2006.

[2] Laurens Van der Maaten and Geoffrey Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[4] Linus Tech Tips, "Remember this day... - amd ryzen 5000 series," .

[5] Laurens Van Der Maaten, "Accelerating t-sne using tree-based algorithms," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3221–3245, 2014.

[6] Martin Wattenberg, Fernanda Viégas, and Ian Johnson, "How to use t-sne effectively," *Distill*, vol. 1, no. 10, pp. e2, 2016.

[7] AMD, "Amd zen 3 core architecture," online: https://www.amd.com/en/technologies/zen-core-3, 2020.

[8] Laurens van der Maaten, "bhtsne," online: `https://github.com/lvdmaaten/bhtsne/`, 2021.

[9] Agner Fog, "C++ vector class library," online: `https://www.agner.org/optimize/vectorclass.pdf`, 2021.

[10] Zack Smith, "Bandwidth: a memory bandwidth benchmark," online: `https://zsmith.co/bandwidth.php`.

[11] James Reinders, "Vtune performance analyzer essentials," *Intel Press*, 2005.