A FAST IMPLEMENTATION OF SPECTRAL CLUSTERING

Leo Horne, Julien Tinguely, Zuowen Wang, Pouya Pourjafar

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Working with non-euclidean data is a key challenge in many fields of science. One of the most established algorithms for unsupervised learning in this domain is *spectral clustering*. In this work we develop a fast implementation of spectral clustering by breaking the algorithm down to its individual components and discovering the main bottlenecks. Furthermore, we discover optimizations reaching across different phases of the algorithm. To achieve this, we make use of fundamental optimization techniques such as scalar replacement, unrolling, cache blocking, and vectorization. Our implementation achieves up to 68% of theoretical peak performance for crucial phases of the algorithm. Our approach also beats popular libraries such as scikit-learn [1, 2] and achieves high performance as well as low runtime.

1. INTRODUCTION

Motivation. Clustering has always been one of the most important domains in unsupervised learning. Its applications include market research, data analysis and many others. Many of these applications, such as gene categorization [3, 4], involve operations on high-dimensional datasets. Spectral clustering first conduct eigenvalue decomposition on the Laplacian of similarity matrix, which serves as a dimension reduction step. It then performs *k*-means algorithm on the resulted space. Since it is suitable for high-dimensional and large datasets, runtime performance becomes crucial, especially for users without GPU hardware.

We implemented a single threaded fast spectral clustering algorithm. When the dimension of data becomes large, it is non-trivial for efficient cache memory data transportation. Moreover, it is expensive to calculate the similarity between data points in Laplacian construction (see Section 2), as well as the distance computations in *k*-means. We believe a fast single threaded implementation is crucial for users with limited computational resources to overcome these issues.

Contribution. We present a fast single threaded spectral clustering implementation which is vectorized for high dimensional data and reasonable large number of clusters. We conduct cache optimization and vectorization with Advanced Vector Extensions (AVX) from Intel, in order to reduce the time spent on data movement and boost the floating point computation performance. We also make use of Hamerly's algorithm, which is a *k*-means algorithm equivalent to Lloyd's algorithm, but requiring less flops. We show that our implementation is substantially faster than the popular scikit-learn library. We believe with comparable hardware settings, namely with similar CPU type and cache sizes, our implementation is one of the fastest.

Related work. In [5] Li *et al.* purposed a GPU-based *k*-means algorithm which optimize high-dimension and low-dimension cases separately. In [6] Alfke *et al.* purposed a method which exploits fast summation based on the non-equispaced fast Fourier transform to embed Lanczos-based eigenvalue decomposition. This matrix-vector multiplication approximation based method avoids to ever construct the graph Laplacian. A fast multi-core/many-core *k*-means implementation was proposed in [7], which utilizes a task mapping strategy for load-balanced job distribution, register blocking technique to improve data locality and a data sharing method to enable efficient updates of computing processing elements. Elkan proposed a fast exact *k*-means in [8], which uses triangle inequalities to skip expensive distance updates at the cost of maintaining extra upper bounds and lower bounds for each data point.

2. BACKGROUND ON THE ALGORITHM

In this section we describe the spectral clustering method and give a formal definition of the variant we implemented and optimized in this work.

Spectral clustering [9, 10]. Spectral clustering is a type of clustering algorithm consisting of mainly four components: (1) similarity graph construction, (2) graph Laplacian construction, (3) spectral decomposition, and (4) *k*-means.

Similarity graph construction. We define $G = (V, E)$ as undirected weighted graph with vertices $V = \{v_1, ..., v_n\}$. w_{ij} stands for the weight of the edge connecting vertices v_i and v_j and $\forall i, j, w_{ij} \geq 0$. We define the weighted adjacency matrix of a graph G as matrix $\mathbf{W}$ with w_{ij} on the

entry (i, j). The degree matrix $\mathbf{D}$ is defined as the diagonal matrix with degrees $d_1, ..., d_n$ on the diagonal, where $d_i = \sum_{i=1}^{n} w_{ij}$.

Given a dataset $\mathcal{X} = \{x_1, ..., x_n\}$, we define s_{ij} as the pairwise similarity between point x_i and x_j. We construct the *similarity graph* as a *fully connected graph* with edge weight s_{ij} between vertices x_i and x_j. In our implementation we choose the Gaussian similarity metric $s(x_i, x_j) = \exp(-\|x_i - x_j\|_2^2 / (2\sigma^2))$, where σ controls the width of the neighborhoods. There are other ways of constructing similarity graphs such as ϵ-neighborhood graph and k-nearest neighbor graph; however, we focus on optimizing fully connected graph with Gaussian similarity function in our work. The graph is useful for modeling the similarity between the data points with non-euclidean geometry.

Graph Laplacian. Graph Laplacian matrices are the main tools for spectral clustering. There exist various types of them. We mainly focus on one type of graph Laplacian used in our project and we describe its properties in this section.

The *unnormalized graph Laplacian matrix* is defined as $\mathbf{L} = \mathbf{D} - \mathbf{W}$, and it has the following properties: (1) $\mathbf{L}$ is symmetric positive semi-definite; (2) $\mathbf{L}$ has n non-negative, real-valued eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. A useful property of the eigenvalues of $\mathbf{L}$ is that the multiplicity k of the eigenvalue 0 of $\mathbf{L}$ equals to the number of connected components in the graph.

Spectral decomposition. We use a third party library called 'Spectra' [11] for solving the spectral decomposition problem. Spectra is a C++ reimplementation of the popular ARPACK framework [12] for efficiently finding extremal eigenvalues of large, possibly sparse matrices. This is useful because spectral clustering only requires the eigenvectors corresponding to the k smallest eigenvalues of the graph Laplacian. It relies on the implicitly restarted Arnoldi method [13] and uses a matrix-free approach, where the user must only specify a matrix-vector product for the computation instead of supplying the matrix itself.

***k*-means.** The last step of spectral clustering is performing the k-means algorithm on points $(y_i)_{i=1,...,n} \in \mathbb{R}^k$ which are the row vectors of the first k columns of $\mathbf{U}$. k-means is one of the 'traditional' clustering algorithms. To initialize the algorithm, the user starts by defining k for the number of clusters. In spectral clustering, the number of clusters is the same as number of eigenvectors we computed in the spectral decomposition (both annotated as k). Then user needs to assign k initial cluster centers as an initialization step, which is vital for the final clustering output. In our work we use the following initialization procedure. Denote the dataset as $\mathcal{X}$, $c_1, ..., c_k$ as centers for the first to k-th clusters, $\text{dist}(a, b)$ as the distance between point a and b, $\forall x \in \mathcal{X}, D(x) = \min_i \text{dist}(x, c_i)$, initialize the centers as the following 1:

Procedure 1: INITIALIZE-CENTERS

Input: A dataset $\mathcal{X}$, desired number of cluster k

1 pick $x \in \mathcal{X}$ $u.a.r.$ and assign $c_1 = x$.

2 pick $x' \in \mathcal{X}$ with probability $\frac{D(x')}{\sum_{x \in \mathcal{X}} D(x)}$.

3 repeat step 2 until k centers sampled.

Output: k points as initial centers

In each iteration, the point which has the largest distance to its nearest cluster center has the biggest probability of being picked as the next center. We can now proceed with the actual clustering. Here we introduce Lloyd's algorithm [14], which is the most broadly used *k*-means algorithm. It performs the following steps:

1. *assign step*: assign each x_i to its nearest cluster C_j
2. *update step*: update each center c_j as the mean of its assigned points
3. repeat steps 1 and 2 until convergence.

At this point we give an overview of the base version of spectral clustering as follows:

Algorithm 1: Spectral Clustering

Input: A dataset $\mathcal{X}$, desired number of cluster k

1 Construct similarity graph G with $\mathcal{X}$.

2 Compute the unnormalized Laplacian $\mathbf{L}$.

3 Compute the first k eigenvectors $u_1, ..., u_k$ of $\mathbf{L}$.

4 Let $\mathbf{U} \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, ..., u_k$ as columns.

5 For $i \in \{1, ..., n\}$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of U.

6 Cluster the points $(y_i)_{i=1,...,n}$ in $\mathbb{R}^k$ with the k-means algorithm into clusters $C_1, ..., C_k$.

Output: k clusters of points

Cost analysis. In this paragraph we first discuss the exact number of floating-point operations. This includes additions (ADD), multiplications (MUL), divisions (DIV), exponential function evaluations (EXP) and square root (SQRT). Notice that we already assume obvious scalar replacement, strength reduction, and code motion. Moreover, we count floating-point comparisons as additions. Since eigendecomposition is not in the scope of our optimization, we skip the cost analysis for it. Denote the dimension of data points in graph similarity and Laplacian as d, and in k-means the dimension of points to be clustered is k.

For similarity graph construction we need to compute pairwise similarities between all pairs of points. In the Gaussian similarity metric we regard $-1/(2\sigma^2)$ as a constant factor. Thus, for each pair of data points, in order to calculate the Euclidean distance between two points and multiply the

constant factor, we need $(d + d - 1)$ ADDs and $(d + 1)$ MULs (We assume basic scalar replacement). Additionally, we need 1 EXP for the exponential function. Since we have n^2 pairs of similarities we need $(2d - 1) \cdot n^2$ ADDs, $(d + 1) \cdot n^2$ MULs and n^2 EXPs.

For computing unnormalized Laplacian $\mathbf{L} = \mathbf{D} - \mathbf{W}$, we need to compute the matrix $\mathbf{D}$ based on $\mathbf{W}$ first. For that we need n ADDs for each row of $\mathbf{W}$ and n rows, resulting in n^2 ADDs in total. Then a simple matrix subtraction $\mathbf{D} - \mathbf{W}$ takes n^2 ADDs. Thus, for $\mathbf{L}$ we need $2n^2$ ADDs when $\mathbf{W}$ is available.

For Lloyd *k*-means initialization, while sampling the i-th ($i \leq 1$, for $i = 0$ no flops are needed) cluster center, including floating-point comparisons for finding $\min_i \text{dist}(x, c_i))$ and distance computation, we need $(2k \cdot i \cdot n + 3n)$ ADDs, $k \cdot i \cdot n$ MULs, $i \cdot n$ SQRTs and $(n+1)$ DIVs. Notice that we ignore the floating point comparisons for $u.a.r.$ sampling since it is a pseudorandom procedure so we would not the exact flops executed. However for the i-th cluster center, we need at least 1 ADDs for comparison and at most n ADDs. Now we sum up each category from $i = 0$ to $k - 1$ and we get $(k^3n - k^2n + 3kn - 3n)$ ADDs, $((k^3n - k^2n)/2)$ MULs, $((k^2n - kn)/2)$ SQRTs and $(kn + k)$ DIVs.

Other than initialization, denote the number of iterations in Lloyd's algorithm as #iter. In the assign step, $(2k-1)kn$ ADDs and k^2n MULs are needed. In the update step, k DIVs and $(n - k - 1)k$ ADDs are needed. Thus, $(2k^2n - k^2 - k) \cdot$ #iter ADDs and $k^2n \cdot$ #iter are executed in the Lloyd *k*-means iterations, excluding initialization.

As of asymptotic time complexity, for the graph construction and Laplacian it is $\mathcal{O}(n^2d)$ and for the Lloyd *k*-means in each iteration it is $\mathcal{O}(nk^2)$.

The cost analysis and asymptotic time complexity for Lloyd's algorithm serves as a worst case analysis for the Hamerly *k*-means algorithm [15], which is introduced for the purpose as an algorithmic optimization to speed up the low dimension *k*-means inherent in spectral clustering.

3. PROPOSED METHOD

This section will explain our approach to optimizing the spectral clustering algorithm. We perform optimizations on the graph construction, Laplacian calculation, and *k*-means steps of spectral clustering. We do not optimize the spectral decomposition step and instead use Spectra, which is already extremely efficient.

It is important to note that although the *k*-means step might at first glance appear to be the most expensive, in reality the graph construction and the Laplacian construction take up a substantial part of the total runtime of the algorithm, as demonstrated in Table 1. Note also that due to the nice properties of the Laplacian, *k*-means usually converges in less than 50 iterations. Therefore, we place our priorities on *both steps* of the algorithm. We start by optimizing obvious bottlenecks, such as the euclidean distance and Gaussian similarity, then move on to optimizing graph construction and *k*-means in a broader sense.

Table 1: Runtime proportions of algorithm components, on a dataset of 5000 points, 72 clusters and 300 dimensions

Computation	Percentage of total runtime
Graph Construction	**28%**
Distance Computations	97%
Matrix Subtraction	3%
Eigenvalue decomposition	**60%**
k-means (100 iterations)	**12%**
Initialization (*k*-means++)	25%
Assign Step	72%
Update Step	3%

3.1. Major bottlenecks

Let us begin by discovering major bottlenecks in the algorithm. Table 1 shows a listing of major components of the algorithm. For the graph construction part, we see that distance calculations constitute the majority of the computation. This includes both the euclidean distance and Gaussian similarity. For the *k*-means part, the main bottleneck is euclidean distance computation, which is present in both the initialization the assign step and takes up to 97% of the runtime. We will therefore discuss how to speed up euclidean distance computation as well as Gaussian similarity (consisting of one euclidean distance and one exponential).

Euclidean distance computations. The euclidean distance, $d(u, v) = \|u - v\|_2$, is used both in the graph construction part (as part of the Gaussian similarity computations) and in the *k*-means part. Therefore, our first step was to optimize the euclidean distance computations as much as possible. We will see later that we also reduced the amount of distance computations insofar as possible in the *k*-means part.

A naïve implementation of a function computing the euclidean distance between two length-q vectors u and v is shown in Algorithm 2. The main bottleneck of this naïve implementation lies in the fact that s is the sole accumulator, and therefore each iteration of the loop is dependent on the previous. An easy way to solve this is by unrolling and using separate accumulators, then summing the accumulators at the end. We determined empirically that using an unroll value of 8 with 8 accumulators yielded the best speedup. Of course, after this unrolled loop, there is a simple loop similar to the one in Algorithm 2 to handle the tail computations if q is not divisible by 8.

However, this unrolled version is suboptimal compared to the naïve implementation when u and v are less than 8-dimensional, since the unrolled loop must first be skipped,

Algorithm 2: Naïve euclidean distance

Input: Two vectors u and v

$s \leftarrow 0$

for $i \in \{1 \dots q\}$ **do**

    s += $(u_i - v_i)^2$

return $\sqrt{s}$

the loop for the tail computation must be entered, and all the accumulators must be summed. For example, if $q = 2$, the naïve code requires only 7 flops, whereas the unrolled version requires 7 flops from the tail computation and then 7 floating-point additions to add all the accumulators together in addition to the overhead of initializing the accumulators to zero and determining whether to skip the unrolled loop.

We therefore decided to implement two different euclidean distance functions, one optimized version for data whose dimension is at least 8, and another version (corresponding to the naïve version) for data whose dimension is less than 8. We decide which one to use only at the beginning of each stage of the algorithm instead of using a conditional statement inside the euclidean distance function. Therefore, we have two different versions of graph construction/Laplacian calculation (low-dimensional vs. high-dimensional) and the same distinction for the *k*-means step.

One final optimization we made was to vectorize the high-dimensional euclidean distance function (of course, it remains unrolled). Also, since the Gaussian similarity only requires the square of the euclidean distance, we provide another version of the euclidean distance without the square root at the end, used solely for computing Gaussian similarity.

Exponential calculation. A slightly less prominent bottleneck lies in the exponential computation which is part of Gaussian similarity. Computation of the exponential using the `exp()` function from the C math library is slow, and, more importantly, cannot be vectorized. We discovered empirically that an approximate exponential is sufficient to produce good clustering results in the case of normalized data (input range $\in \{-700, 700\}$).

A suitable choice of approximate exponential lies in the Schraudolph fast exponential algorithm [16]. To compute e^x, this algorithm uses two double-precision floating-point constants c_1 and c_2 and stores an integer representation of $c_1 x + c_2$ in the most significant 32 bits of an IEEE 754-compliant double-precision floating-point number (the part corresponding to the sign bit, the exponent field, and part of the mantissa field). The exact details, including the choice of the constants c_1 and c_2, are beyond the scope of this paper, and we advise the reader to refer to [16].

Clearly, the Schraudolph exponential is quite efficient, as it only requires a multiplication, an addition, a conversion to a 32-bit integer, and an update to the most significant

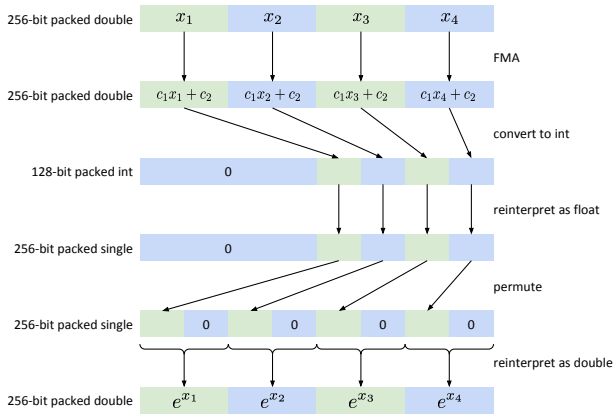

Fig. 1: Vectorized Schraudolph exponential

32 bits of a double. It also claims to approximate the true exponential e^x well for values of x between -700 and 700.

The other big advantage of the Schraudolph exponential is that it lends itself well to vectorization. We can easily compute the exponential of four numbers at once, since updating arbitrary 32-bit fields in an AVX packed double can be done simply by reinterpreting the vector as a packed single. The only overhead comes from some permutations necessary to bring the vectors into the right order, although we note that these permutations will not even be necessary in AVX-512 due to the introduction of the `vcvtpd2qq` assembly instruction. An overview of the vectorized Schraudolph exponential is presented in Figure 1. Note that the steps starting with 'reinterpret' are not actually machine instructions (they are used by the compiler) and thus have 0 latency.

3.2. Similarity graph and Laplacian calculation

Oneshot construction. Recall that the unnormalized Laplacian is computed as $\mathbf{L} = \mathbf{D} - \mathbf{W}$. Note that our graph does not have self-loops, i.e. the diagonal values of $\mathbf{W}$ are all zero. Recall also that $\mathbf{D}$ is a diagonal matrix whose entries are composed of the sums of the rows of $\mathbf{W}$. Hence, the off-diagonal entries of $\mathbf{L}$ are simply the negated values from $\mathbf{W}$, and the diagonal elements of $\mathbf{L}$ can be computed by keeping track of the elements of $\mathbf{W}$ in an accumulator as we compute them. Although the flop count remains identical, this avoids explicitly computing and storing three different matrices, which is better for spatial locality, as we compute the off-diagonal elements in sequential order and do not hop between the rows of three different matrices (as would happen when explicitly computing $\mathbf{L} = \mathbf{D} - \mathbf{W}$). Furthermore, we use an accumulator array for better spatial locality as opposed to simply updating the diagonal entries of $\mathbf{L}$.

One last optimization lies in the fact that $\mathbf{L}$ is symmet-

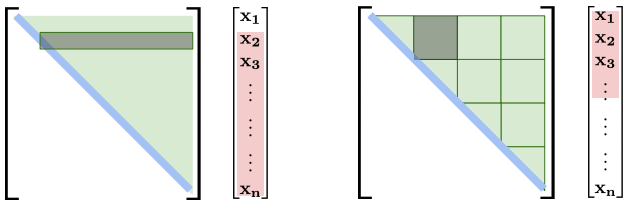

Fig. 2: *Left*: non-blocked Laplacian computation. The diagonal entres of $\mathbf{L}$ coming from $\mathbf{D}$ (i.e. the sums of the rows of $\mathbf{W}$) are shown in blue, while the off-diagonal elements (negative Gaussian similarities coming from $\mathbf{W}$) are shown in green. Computing one row of $\mathbf{L}$ requires bringing almost the entire dataset $\mathcal{X}$ into cache, shown in red. *Right*: blocked Laplacian calculation. Computing one block of $\mathbf{L}$ only requires bringing a small part of the dataset $\mathcal{X}$ into memory, and these elements, shown in red, are reused to compute the entire block.

ric. Since we use a matrix-free eigensolver, we can simply avoid computing the lower triangle of $\mathbf{L}$ and define a new matrix-vector product that does not use the lower triangle of $\mathbf{L}$. However, this means that updating the diagonal values of $\mathbf{L}$ becomes slightly more complex, as the lower triangle is not calculated anymore. Instead, we update the accumulator both for the column and for the row currently being operated on. Note that since we are not computing the lower triangle anymore, the flop count is now reduced compared to the baseline implementation. The final (non-vectorized) algorithm is shown in Algorithm 3.

Algorithm 3: Oneshot Laplacian computation

Input: An array $\mathcal{X}$ of points, a similarity measure s
accumulators $\leftarrow$ array of length n
for $i \in \{1 \dots n\}$ **do**
  for $j \in \{i+1 \dots n\}$ **do**
    $\mathbf{L}_{ij} \leftarrow -s(x_i, x_j)$
    accumulators[i] += $s(x_i, x_j)$
    accumulators[j] += $s(x_i, x_j)$
for $i \in \{1 \dots n\}$ **do**
  $\mathbf{L}_{ii} \leftarrow$ accumulators[i]
Output: Graph unnormalized graph Laplacian $\mathbf{L}$

Cache blocking. Let us now analyze the cache behavior and how it can be optimized. Consider computing the first row of the Laplacian (except for the first element, which is on the diagonal). It requires computing the similarity between x_1 and *every other point* x_2 to x_n. Then, computing row 2 requires computing the distance of x_2 to all subsequent points x_3 to x_n, and so on. This is clearly very bad for cache locality, as when n and d are sufficiently large, earlier data points needed by the next iterations will have been evicted from the cache by the later data points needed by the current iteration. In other words, assuming that the points array is sufficiently large, computing each row of the Laplacian will require reloading almost all points from memory. Of course, this effect diminishes as the algorithm progresses because the number of off-diagonal elements in the upper triangle of the Laplacian decreases as we progress through the rows, but for large values of n, the cache behavior is still a concern.

To alleviate these concerns, we implement a variant of the oneshot Laplacian computation which iterates over the matrix in blocks of size $b \times b$. Computing all the values within each block only requires at most $2b$ data points, and if b is chosen appropriately, these data points can always remain in cache. These $2b$ data points are then used multiple times within a block, and the rows of the blocks are contiguous in memory, leading to good spatial and temporal locality as opposed to the naïve version, as illustrated in Figure 2.

Vectorization. The natural next step in optimization would be to vectorize the Laplacian computation. Since we are using AVX/AVX2, one vector register can hold four doubles. Hence, the most natural way to vectorize the computation is to compute four off-diagonal entries of the Laplacian at once. Since the off-diagonal entries $\mathbf{L}_{ij}$ correspond to the negative similarity between points x_i and x_j. The first step is to calculate the euclidean distance between x_i and points x_j to x_{j+3}. We do this using the vectorized euclidean norm from Section 3.1 in such a way that the four euclidean distances end up in one AVX register. We can then multiply by -0.5 and use the vectorized approximate exponential from Section 3.1 to compute four entries of the Laplacian at once.

Of course, this can be done on top of cache blocking so that the performance gains from each are complimentary. It is also important to note that since we have two different euclidean distance functions (depending on whether the input data is high- or low-dimensional), we also introduce two versions of the Laplacian computation for high- and low-dimensional data which use the corresponding version of the euclidean distance function.

3.3. *k*-means

In the following, we show how avoiding distance computations and vectorizing *k*-means improved performance. Using Lloyd's algorithm as presented in Section 2, every iteration requires computing the euclidean distance from each point to each center. This is suboptimal on an algorithmic level, as many of the distance computations are in fact unnecessary [8]. We therefore seek out another algorithm which has the following properties: (1) it avoids unnecessary distance computations, since these take up the bulk of the computation time, (2) it works well for low-dimensional input data (recall that in spectral clustering, the input di-

mensionality for the k-means step is k as opposed to d, and spectral clustering is typically not used for large numbers of clusters [9]), and (3) it computes exactly the same result as Lloyd's algorithm, i.e. it is not an approximation.

Hamerly *k*-means. As a first optimization on the algorithmic level, we implemented Hamerly k-means, an exact k-means algorithms which satisfies the three constraints above. It relies on the triangle inequality inherent to any distance metric to avoid unnecessary distance computations by maintaining upper and lower bounds. The exact mathematical details are not relevant to this work and we advise the reader to refer to [2] for details. However, we present the rough pseudocode of the algorithm in Algorithm 4 to refer back to when discussing further optimizations. We advise the reader to refer to [2] for detailed pseudocode. We first define the notations. $c(i)$ is the center of cluster C_i (where $1 \leq i \leq k$). $c'(i)$ is the vector sum of all points in cluster C_i. $p(i)$ is the distance that $c(i)$ moved compared to last iteration, $s(i)$ is the distance of c_i to its second closest cluster center. $a(j)$ is the index of the cluster where data point $x(j)$ is assigned (where $1 \leq j \leq n$). $u(j)$ is the upper bound of the distance between $x(j)$ and its cluster center $c(a(j))$, and $l(j)$ is the lower bound.

In the initial version of Hamerly k-means that we implemented, we made sure to perform basic C optimizations such as code motion and arithmetic simplifications, and we consider this version of our implementation to be the baseline implementation of Hamerly k-means.

Cache behavior of Hamerly *k*-means. We performed a cache analysis of the Hamerly k-means algorithm and did not find major cache-based optimizations to perform. Each of the loops iterates over an array which is contiguous in memory (which provides good spatial locality), and there is some temporal locality due to the fact that the loops which iterate over a n internally iterate over arrays of size k, where typically $k << n$. Unfortunately, the temporal locality is inherently limited by the fact that the algorithm requires completely finishing one iteration before moving on to the next, precluding any possibility of cache blocking (e.g. computing 10 updates for a small number of points).

Vectorized Hamerly *k*-means. Having performed basic C optimizations and a cache analysis, we move on to vectorization. We again distinguish between high and low dimensional instances and only vectorize cases where the number of clusters is greater or equal to 8. The initialization (line 1) requires picking points with probability proportional to their distance to other centers. For this we need to calculate a cumulative sum as a normalizer to have valid probabilities. The cumulative sum can be vectorized by shifting the vector to the right by 1 cell and zeroing the elements on the left. We then add this temporary vector to the sum and do a right shift by one again until we have the cumulative sum of the entire vector in the last cell. This requires 4 additions, 3 permutations, and 3 logical operations in each iteration (see Appendix 7.2 for a pictorial representation). Another optimization we made was to align all arrays to a 32-byte boundary so as to be able to use aligned vector load/store instructions. This allows us to easily vectorize the other operations needed in hamerly. We discovered that splitting and reordering some of the computations can help us vectorize the code more smoothly. The maximum on line 7 can be precomputed and stored in an array using vectorization. Vectorizing lines 8 to 14 did not reduce the runtime and only caused more overhead. It is straight forward to vectorize the remaining routines, e.g. by unrolling by 8 (as opposed to 4 in order to reduce dependencies) and replacing all scalar instructions with their equivalent vector instructions and replacing conditionals with masks. Computing the sum of clusters c' can be implemented via a vectorized reduction and calculating how much each center moved (*Move-Centers*, line 17) computation and updating the upper and lower bounds (*Update-Bounds*, 18) are simple arithmetic computations.

Algorithm 4: Hamerly k-means

```
   Input: An array X of points, number of clusters k
 1 c ←INITIALIZE-CENTERS(k, X)        // Proc. 1
 2 INITIALIZE-BOUNDS(u, l, a)         // Proc. 2
 3 while not converged do
 4     for i ∈ {1, ..., k} do
 5         s(i) ← min_{i'≠i} dist(c(i'), c(i))
 6     for j ∈ {1, ..., n} do
 7         m ← max(s(a(j))/2, l(j))
 8         if u(j) > m then           // bounds test
 9             u(j) ← dist(x(j), c(a(j)))
10             if u(j) > m then       // bounds test
11                 a' ← a(j)
12                 a(j) ← arg min_t dist(x(j), c(t))
13                 u(j) ← dist(x(j), c(a(j))
14                 l(j) ← min_{t≠a(j)} dist(x(j), c(t))
15                 if a(j) ≠ a' then
16                     update c'(a'), c'(a(j))
17     MOVE-CENTERS(c', q, c, p)        // Proc. 3
18     UPDATE-BOUNDS(p, a, u, l)        // Proc. 4
   Output: k clusters of points
```

Procedures 2–4 in Appendix 7.1

4. EXPERIMENTAL RESULTS

In this section, we present the runtime and performance of each optimization for both the graph construction and k-means. Table 2 gives an overview of the final proportions of each stage of the algorithm for a dataset with $n = 5000$, $k = 72$, and $d = 300$.

Table 2: Runtime proportions of the final version

Computation	Percentage of total runtime
Graph Construction	**13%**
Fast Gaussian similarity	85%
Eigenvalue decomposition	**86%**
k-means (100 iterations)	**1%**
Initialization (k-means++)	23%

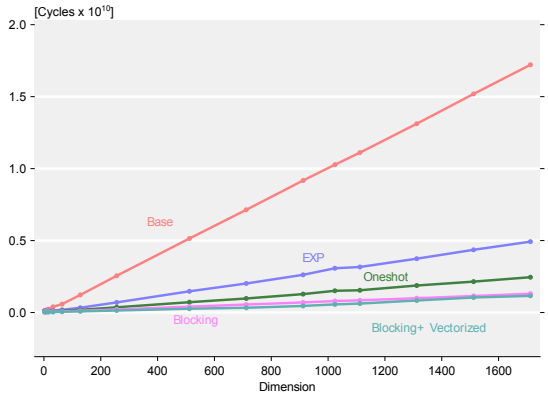

Fig. 3: Runtime plots for Laplacian construction on a dataset of growing d with $k = 6$ and $n = 5000$. A speedup of 14.8x is achieved relative to the base implementation.

Experimental setup. All the experiments were conducted on a machine using an Intel Core i5-8257U CPU with 1.4 GHz base frequency with a first level cache size of 32 KB. The code was compiled using *gcc 9.3.0* with the following flags: **-O3 -ffast-math -march=skylake -mfma**. Three groups of experimental datasets were used with different free parameters. One set had a growing number of data points n, one a growing number of clusters k and the last a growing dimensionality of the data d. All datasets were created with *scikit-learn*'s `make_blobs` function. The graph construction was tested on the growing n and growing d datasets and the k-means implementation was tested on the growing n and growing k. The growing n dataset had a fixed dimension of 300 with a fixed number of clusters of 72 and $n \in \{100, 200, \ldots, 6000\}$. The growing k dataset has a fixed number of 2500 points and a dimension of 2 and $k \in \{2, 3, \ldots, 99\}$ and the growing d dataset has 5000 points generated from 6 clusters and $d \in \{2^1, 2^2, \ldots, 2^9\} \cup \{512, 712, \ldots, 1912\}$.

Laplacian construction. We observe in Figure 3 (labeled *EXP*) that we can achieve a runtime reduction of 3.5x by optimizing the euclidean distance and exponential since they are a substantial part of the computation. Further reducing the flop count (labeled *Oneshot*) and performing blocking and vectorization gives us a total speedup of 14.8x compared to the base implementation. Looking at the performance plots in Figure 4 we see that that the biggest

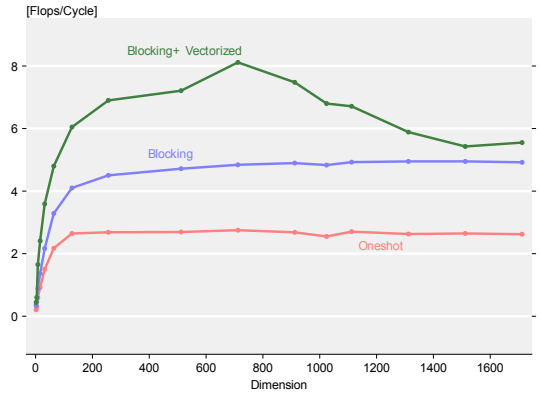

Fig. 4: Performance plots for Laplacian construction on a dataset of growing dimension with $k = 6$ and $n = 5000$

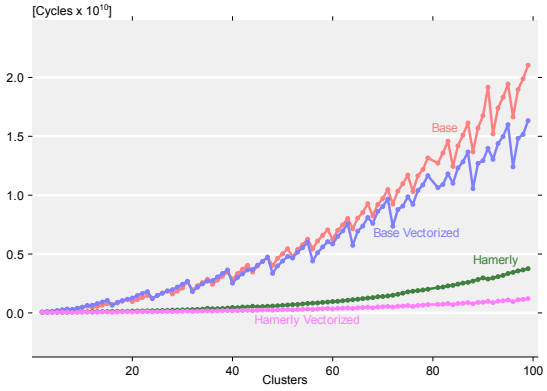

Fig. 5: Runtime plot for Hamerly k-means on a dataset with growing k and $n = 2500$.

boost in performance comes from vectorization and blocking. We reach a top performance of 8 flops/cycle for 712-dimensional points since we both unroll by 8 for vectorization and also make 32×32 blocks. Having an instruction mix of one vectorized FMA and one vectorized addition per cycle puts the peak achievable performance at 12 flops per cycle and shows that we reach 68% of peak performance. We see that the performance drops below 6 flops/cycle after 1112 dimensional points since 4 lines of this dataset already fill the cache completely and blocking becomes less beneficial. We observe that vectorization gives us a maximum performance boost of 1.67x on top of blocking alone and nearly a 3x performance boost compared to oneshot computation. The base implementation is not included in the performance plot since it has a different flop count than the optimized versions. We observe a similar behavior for a dataset with growing n. For completeness we report these results in Appendix 7.3.

***k*-means.** For k-means we implemented Lloyd and

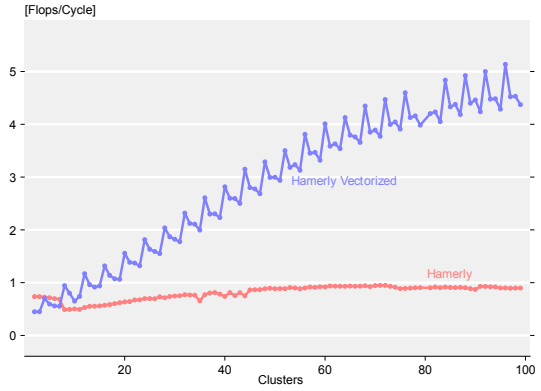

Fig. 6: Performance plot for Hamerly k-means on a dataset with growing k and $n = 2500$.

Hamerly's algorithm and optimized the faster one which turned out to be Hamerly. Since the best possible optimization in k-means is avoiding unnecessary computations, the flop counts are lower for the faster algorithm. As we observe in Figure 5, the Hamerly implementation is 4.34 times faster than Lloyd k-means with optimized euclidean distance computations. By vectorizing Hamerly, we gain a maximum runtime reduction of 19x compared to the base implementation of Lloyd. We observe in Figure 6 that a performance gain is achieved by vectorizing Hamerly k-means. The zigzagging pattern is due to the unrolling that is present in the vectorized implementation. The peaks are at multiples of 8 since the unrolling parameter is 8 and no tail computations need to be performed at those points. A second smaller peak is achieved at multiples of 4 due to the vectorized tail computation. A maximum performance gain of about 5.7 times is achieved due to vectorization. By checking the assembly code generated for sequential and vectorized Hamerly, we observe that the vectorized version contains 1000 FMAs more than the sequential version and this gives it a big boost in performance for larger number of k and makes the code more scalable. The experiments on the growing n dataset are reported in Appendix 7.3.

Roofline analysis. We conclude our discussion with a roofline analysis in Figure 7 in order to determine how well our algorithm is using the available resources. As discussed before, for both k-means and Laplacian construction, our instruction mix consists of one vectorized FMA and one vectorized addition per cycle, which puts the performance roof at 12 flops/cycle. The memory bandwidth was obtained from the *STREAM* benchmark [17] at approximately 18 GB/s. For the Laplacian construction, the operational intensity can be determined exactly as $\frac{1.5d(n-1)+4n-3}{4n+8d+8}$. Our code reaches 68% of peak performance for growing input dimensionality and there is a drop for points with a dimension greater than 712, presumably due to only a couple points being able to fit in L1 cache. The operational intensity for k-means cannot be calculated exactly since the number of distance calculations Hamerly's algorithm skips is not known a priori and has to be determined empirically. The memory overhead is $8(nk + 2k^2 + 3k + n)$ bytes. We reach 29% of peak performance. Considering that k-means has very limited temporal locality due to dependencies between subsequent iterations, it is very difficult to gain better performance on a single core. In Appendix 7.3 we compare our implementation to two popular and well established benchmarks [1, 2] and show that we obtain superior results in terms of runtime.

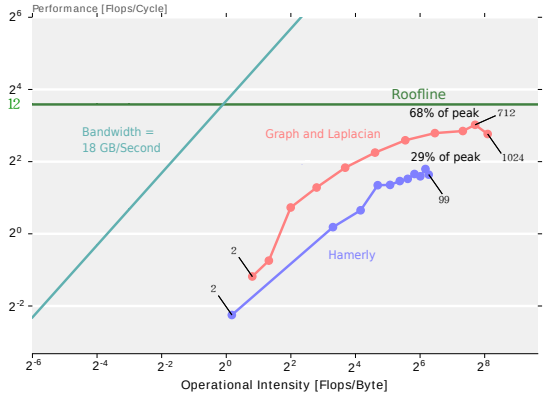

Fig. 7: Roofline analysis with $d \in \{2, \ldots, 1024\}$ and $k \in \{1, \ldots, 99\}$

5. CONCLUSIONS

In this case study of spectral clustering we have seen that one can achieve a high performance and fast implementation of spectral clustering without any multiprocessor programming. Since in spectral clustering the actual clustering of the dataset converges quite fast after building the graph [9] having a high performing graph construction is of utmost importance and the optimizations that we perform result in an efficient, fast and scalable Laplacian calculation. Further, by implementing an algorithm that avoids a lot of the distance calculations, we can remove a lot of the overhead in the clustering and by vectorizing the remaining computations achieve a scalable and fast k-means implementation. We achieve 68% of peak performance for Laplacian construction and drastic runtime reductions for the k-means step. Our optimizations can be applied in many other problem settings since both k-means and graph Laplacians are highly used in a variety of modern algorithms [18, 19, 20].

6. REFERENCES

[1] David Cournapeau, "scikit-learn," `https://github.com/scikit-learn/scikit-learn`, 2007.

[2] Greg Hamerly, "Fast *k*-means Clustering Toolkit," `https://github.com/ghamerly/fast-kmeans`, 2014.

[3] Sarah A. Vitak, Kristof A. Torkenczy, Jimi L. Rosenkrantz, Andrew J. Fields, Lena Christiansen, Melissa H. Wong, Lucia Carbone, Frank J. Steemers, and Andrew Adey, "Sequencing thousands of single-cell genomes with combinatorial indexing," *Nature Methods*, vol. 14, no. 3, pp. 302–308, feb 2017.

[4] Chongzhi Zang, Tao Wang, Ke Deng, Bo Li, Sheng'en Hu, Qian Qin, Tengfei Xiao, Shihua Zhang, Clifford A. Meyer, Housheng Hansen He, Myles Brown, Jun Shirley Liu, Yang Xie, and X. Shirley Liu, "High-dimensional genomic data bias correction and data integration using MANCIE," *Nature Communications*, vol. 7, no. 1, pp. 1–8, apr 2016.

[5] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu, "Speeding up k-Means algorithm by GPUs," *Journal of Computer and System Sciences*, vol. 79, no. 2, pp. 216–229, mar 2013.

[6] Dominik Alfke, Daniel Potts, Martin Stoll, and Toni Volkmer, "NFFT Meets Krylov Methods: Fast Matrix-Vector Products for the Graph Laplacian of Fully Connected Networks," *Frontiers in Applied Mathematics and Statistics*, vol. 4, aug 2018.

[7] Min Li, Chao Yang, Qiao Sun, Wen Jing Ma, Wen Long Cao, and Yu Long Ao, "Enabling Highly Efficient k-Means Computations on the SW26010 Many-Core Processor of Sunway TaihuLight," *Journal of Computer Science and Technology*, vol. 34, no. 1, pp. 77–93, jan 2019.

[8] Charles Elkan, "Using the triangle inequality to accelerate k-means," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*. 2003, ICML'03, p. 147–153, AAAI Press.

[9] Ulrike Von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, vol. 17, no. 4, pp. 395–416, 2007.

[10] Andrew Y Ng, Michael I Jordan, and Yair Weiss, "On spectral clustering: Analysis and an algorithm," in *Advances in Neural Information Processing Systems*, 2002.

[11] Yixuan Qiu, "Spectra: C++ Library For Large Scale Eigenvalue Problems," `https://spectralib.org`, 2015.

[12] R. B. Lehoucq, D. C. Sorensen, and C. Yang, "ARPACK Users Guide: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods," 1997.

[13] Danny C. Sorensen, *Implicitly Restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations*, pp. 119–165, Springer Netherlands, Dordrecht, 1997.

[14] Stuart P Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[15] Greg Hamerly, "Making *k*-means even faster," in *Proceedings of the 2010 SIAM International Conference on Data Mining*, 2010, pp. 130–140.

[16] Nicol Schraudolph, "A Fast, Compact Approximation of the Exponential Function," *Neural computation*, vol. 11, pp. 853–62, 06 1999.

[17] John D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," `https://www.cs.virginia.edu/stream/`.

[18] Rasmus Kyng, Richard Peng, Sushant Sachdeva, and Di Wang, "Flows in almost linear time via adaptive preconditioning," *CoRR*, vol. abs/1906.10340, 2019.

[19] Daniel A. Spielman and Shang-Hua Teng, "Spectral sparsification of graphs," *CoRR*, vol. abs/0808.4134, 2008.

[20] Mahdi Saatchi, Mathew C. McClure, Stephanie D. McKay, Megan M. Rolf, JaeWoo Kim, Jared E. Decker, Tasia M. Taxis, Richard H. Chapple, Holly R. Ramey, Sally L. Northcutt, Stewart Bauck, Brent Woodward, Jack CM Dekkers, Rohan L. Fernando, Robert D. Schnabel, Dorian J. Garrick, and Jeremy F. Taylor, "Accuracies of genomic breeding values in American Angus beef cattle using K-means clustering for cross-validation," *Genetics Selection Evolution*, vol. 43, no. 1, pp. 40, 2011.

7. APPENDIX

7.1. Hamerly subroutines

Procedure 2: INITIALIZE-BOUNDS

for $j \in \{1, \ldots, k\}$ **do**
  $c'(j) \leftarrow \vec{0}$
for $i \in \{1, \ldots, n\}$ **do**
  $a(i) \leftarrow \arg\min_j \text{dist}(x(i), c(i))$
  $u(i) \leftarrow \text{dist}(x(i), c(a(i)))$
  $l(i) \leftarrow \min_{j \neq a(i)} \text{dist}(x(i), c(j))$
  $c'(a(i)) \leftarrow c'(a(i) + x(i)$

Procedure 3: MOVE-CENTERS

for $j \in \{1, ..., n\}$ **do**
  $c^* \leftarrow c(j)$
  $c(j) \leftarrow c'(j)/|C_j|$
  $p(j) \leftarrow \text{dist}(c^*, c(j))$

Procedure 4: UPDATE-BOUNDS

$r \leftarrow \arg\max_t p(k)$
$r' \leftarrow \arg\max_{k \neq r} p(k)$
for $i \in \{1, ..., |u|\}$ **do**
  $u(i) \leftarrow u(i) + p(a(i))$
  if $r = a(i)$ **then**
    $l(i) \leftarrow l(i) - p(r')$
  else
    $l(i) \leftarrow l(i) - p(r)$

7.2. Vectorized cumulative sum

Figure 8 shows the vectorized cumulative sum mentioned in Section 3.3.

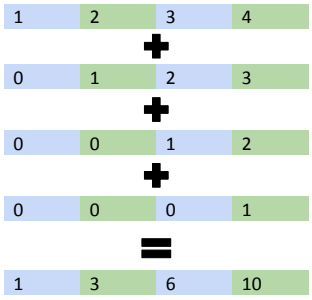

Fig. 8: Vectorized cumulative sum calculation

7.3. Further experimental results

The performance plots of Laplacian construction for growing number of data points can be found in Figure 9. The blocked vectorized code is optimal and achieves the best performance across the entire range of points. If n is big enough it reaches a top performance of 7 flops per cycle which is about 58% of peak performance for our instruction mix. A slight fall in performance is observed at 4000 points which is most likely due to interference from other processes or the kernel.

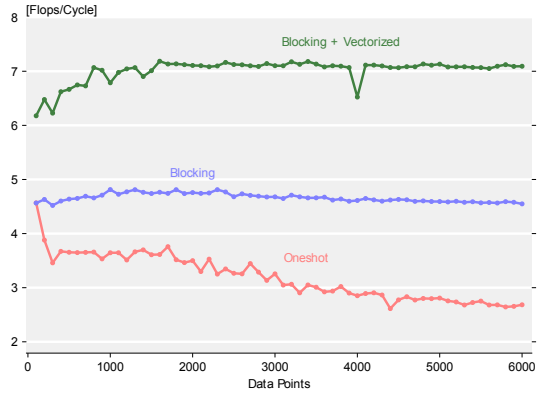

Fig. 9: Performance plots for Laplacian construction on a dataset of growing number of points with 72 clusters and 300 dimensional points.

The runtime results for Laplacian construction with a growing amount of data points are reported in Figure 10. A maximum runtime reduction of 5.87 times is reached relative to the base implementation.

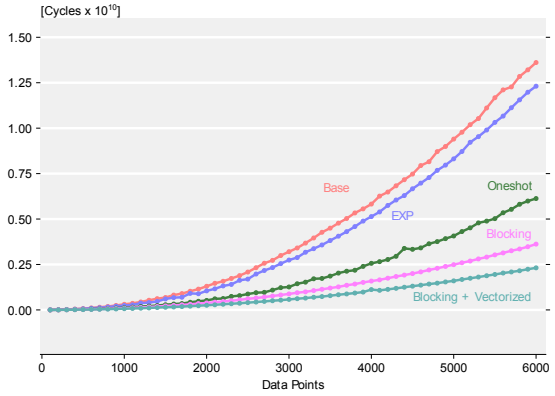

Fig. 10: Runtime plots for Laplacian construction on a dataset of growing number of points with 72 clusters and 300 dimensional points. A speedup of 5.87x is achieved relative to the base implementation.

Our vectorized Hamerly implementation achieves superior runtime results also in the case of growing number of data points. We see in Figure 11 that the runtime grows more slowly with increasing number of points due to the

Hamerly algorithm being able to skip a lot of computations. We also see in Figure 12 that we reach a peak performance of about 6 with the vectorized code which is 50% of peak performance.

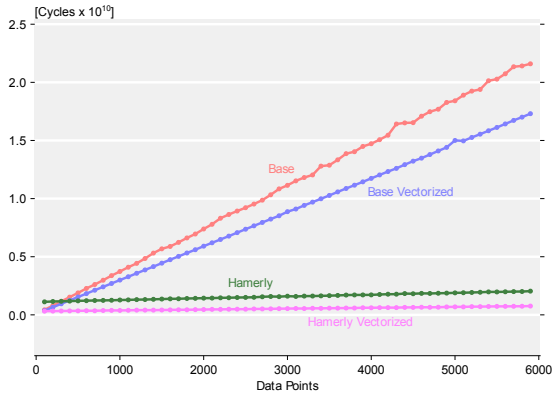

Fig. 11: Runtime plots for different k-means implementations on a dataset of growing number of points with 72 clusters and 300 dimensional points.

Comparison to benchmarks. We compare our best implementation to two well established and popular benchmarks the *Fast K-means Clustering Toolkit* [2] and *scikit-learn* [1]. These experiments were conducted on a machine with an Intel Core i7-8550U CPU with 1.80 GHz base frequency and a first level cache size of 32 KB. The experiments are run on the same datasets as before.

Our graph construction is 3 times faster than *scikit-learn* as evident in Figure 14. Early stopping was disabled in comparing the different k-means implementations. We compare to *scikit-learn*'s base (Lloyd) and Elkan [8] implementation which is an alternative to Hamerly. We also compare to

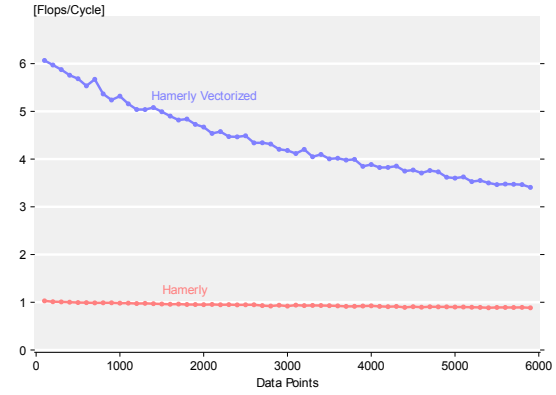

Fig. 12: Performance plots for Hamerly k-means on a dataset of growing number of points with 72 clusters and 300 dimensional points.

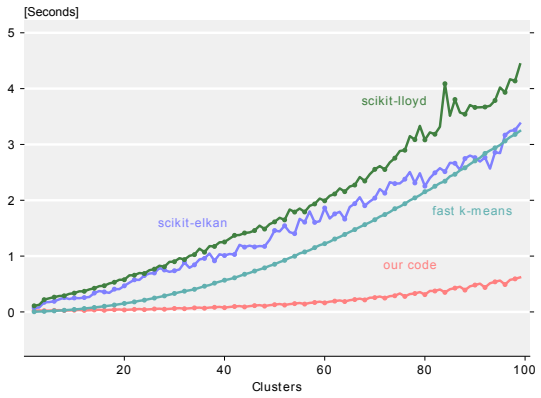

Fig. 13: Runtime comparison for different k-means implementations from scikit-learn, which is a popular machine learning library, fast k-means and our code. Fast k-means is a Hamerly k-means implementation [2] from its author in C++. This is on a dataset with growing number of clusters and 2500 points. Recall that for the k-means step, the number of clusters is equal to the dimensionality.

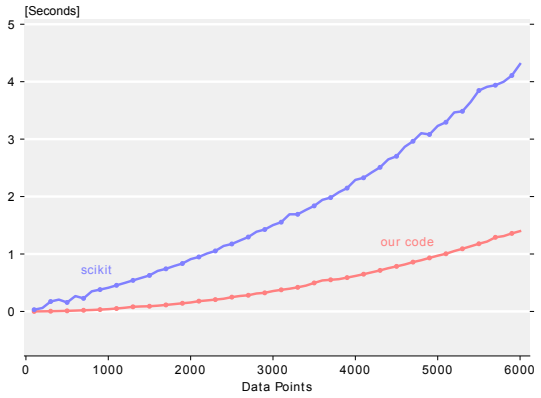

Fig. 14: Runtime comparison against *scikit-learn*'s graph construction. This is on a dataset of growing number of points with 72 clusters and 300 dimensional points.

the Hamerly implementation of *Fast K-means Toolkit*. We observe in Figure 13 that our implementation beats *scikit-learn*'s base implementation by 7.2x and is 5.46 times faster than Elkan and also 5.24 times faster than the *Fast K-means Toolkit*.

8. CONTRIBUTIONS OF TEAM MEMBERS

Julien. I implemented the straightforward C code for Lloyd k-means. I worked on non-SIMD optimizations for k-means and implemented the base Hamerly k-means. I worked on SIMD optimizations in Euclidean Distance and Gaussian Similarity computations. I helped Leo on vectorizing Laplacian.

Leo. Worked on cache blocking for the Laplacian. Vectorized Schraudolph exponential. Worked with Julien on vectorizing Laplacian. Worked on data alignment for SIMD optimizations. Integrated Spectra into codebase to work in the presence of Laplacian optimizations (missing lower triangle). Helped Pouya and Zuowen with the vectorized cumulative sum.

Pouya. ... I worked on non-SIMD optimizations for the Laplacian, sequential schraudolph's EXP, some other exp implementations that weren't used, some approximate square root implementation that wasn't used. I worked on SIMD optimizations in Hamerly k-means and the initialization along with Zuowen.

Zuowen. I optimized unrolling version of Euclidean distance, made the low-dim and high-dim distinction. I implemented a straightforward C code for Elkan algorithm as an algorithmic optimization. I worked on SIMD optimizations in Hamerly k-means and the initialization along with Pouya.