# GPU-acceleration of dimensionality reduction algorithm EmbedSOM

Adam Šmelko
Martin Kruliš
Jiří Klepl
smelko@d3s.mff.cuni.cz
krulis@d3s.mff.cuni.cz
klepl@d3s.mff.cuni.cz
Department of Distributed and Dependable Systems, Charles University
Prague, Czechia

## Abstract

Dimensionality reduction methods have found vast applications as visualization tools in diverse areas of science. Although many different methods exist, their performance is often insufficient for providing quick insight into many contemporary datasets. In this paper, we propose a highly optimized GPU implementation of EmbedSOM, a dimensionality reduction algorithm based on self-organizing maps. We detail the optimizations of $k$NN search and 2D projection kernels which comprise the core of the algorithm. To tackle the thread divergence and low arithmetic intensity, we use a modified bitonic sort for $k$NN search and a projection kernel that utilizes vector loads and register caches. The evaluated performance benchmarks indicate that the optimized EmbedSOM implementation is capable of projecting over 30 million individual data points per second.

***CCS Concepts:*** • **Software and its engineering** → **Software product lines**; **Designing software**.

***Keywords:*** Dimensionality reduction, Output feedback, Single-cell cytometry, Visualization, GPU acceleration

**ACM Reference Format:**
Adam Šmelko, Martin Kruliš, and Jiří Klepl. 2024. GPU-acceleration of dimensionality reduction algorithm EmbedSOM. In *Proceedings of THE 14TH WORKSHOP ON GENERAL PURPOSE PROCESSING USING GPU (GPGPU '24).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*GPGPU '24, MARCH 2–3, 2024, EDINBURGH, UK*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Dimensionality reduction algorithms emerged as indispensable utilities that enable various forms of intuitive data visualization, providing insight that in turn simplifies rigorous data analysis. The development has benefited especially the life sciences, where algorithms like t-SNE [16] reshaped the accepted ways of interpreting many kinds of measurements, such as genes, single-cell phenotypes and development pathways, and behavioral patterns [2, 15].

The performance of the non-linear dimensionality reduction algorithms becomes a concern if the analysis pipeline is required to scale or when the results are required in a limited amount of time such as in clinical settings. To tackle the limitations of poor scalability, Kratochvíl et al. developed EmbedSOM [7], a dimensionality reduction and visualization algorithm based on self-organizing maps (SOMs) [5]. EmbedSOM provided a 10× speedup on datasets typical for single-cell cytometry data visualization while retaining the competitive quality of the results. Still, the parallelization potential of EmbedSOM remained mostly untapped as of yet.

This paper describes an efficient, highly parallel GPU implementation of EmbedSOM designed to provide real-time results on large datasets. The implementation is accompanied by performance benchmarks of individual optimizations to evaluate the optimal variants for different dataset sizes. Both the implementation and the empirical data are available in our GitHub repository[1].

In the paper, we first describe the EmbedSOM algorithm in Section 2. We specifically detail the CUDA-based GPU implementation of the algorithm in Section 3 and evaluate its performance in Section 4. Related work is discussed in Section 5 and Section 6 concludes the paper.

## 2 Landmark-directed dimensionality reduction

EmbedSOM is a visualization-oriented method of non-linear dimensionality reduction that works by describing a high-dimensional point by its location relative to landmarks equipped

[1] https://github.com/asmelko/gpgpu24-artifact

with a topology and reproducing the point in a low-dimensional space using an explicit low-dimensional projection of the landmarks with the same topology [7].

More formally, the EmbedSOM algorithm works as follows. Let $d$ be the dimension of the high-dimensional space and assume $\mathbb{R}^2$ is the low-dimensional space for brevity. EmbedSOM processes $n$ $d$-dimensional points in a matrix $X$ of size $n \times d$, and outputs $n$ 2-dimensional points in matrix $x$ of size $n \times 2$. The high- and low-dimensional landmarks similarly form matrices $L$ of size $g \times d$ and $l$ of size $g \times 2$, where usually $g \ll n$. Each point $X_i$ is transformed to a point $x_i$ as:

1. $k$ nearest landmarks are found for point $X_i$ ($k$ is a constant parameter satisfying $3 \leq k \leq g$)
2. the landmarks are ordered and a score is assigned to each of them, using a smooth function of the distance that assigns the highest score to the closest landmark and 0 to the $k$-th landmark (this ensures the smoothness of projection in cases when $k < g$ [7])
3. for each pair $(u, v)$ of the closest $k - 1$ landmarks (i.e., the ones with non-zero score), a projection of the point $X_i$ is found on the 1-dimensional affine space with coordinate 0 at $L_u$ and 1 at $L_v$; the 1-dimensional coordinate of the projection in this affine space is taken as $D_{uv}(X_i)$ and the same projected coordinates are defined in the low-dimensional space as $d_{uv}(x_i)$
4. point $x_i$ is fitted to the low-dimensional space so that the squared error in the coordinates weighed by nearest-landmark scores $(s_u, s_v)$ is minimized:

$$x_i = \arg\min_{p \in \mathbb{R}^2} \sum_{u,v} s_u \cdot s_v \cdot (D_{uv}(X_i) - d_{uv}(p))^2$$

Because $d_{uv}(p)$ is designed as a linear operator, the error minimization problem (step 4) collapses to a trivial solution of 2 linear equations with 2 variables. A complete algorithm may be found in the original publication [7, Algorithm 1].

## 3 GPU implementation of EmbedSOM

While EmbedSOM is relatively straightforward to parallelize for mainstream CPU architectures, several challenges appear when designing of an optimal implementation for contemporary GPUs. In this section, we outline the key optimizations that allowed us to run the high-performance dimensionality reduction in EmbedSOM, and give an overview of the relative performance gains achieved by the algorithm choice.

Technically, the algorithm consists of two main parts that provide distinct implementation challenges:

- **$k$-NN step** The search of $k$-nearest landmarks in $L$ for each data point from $X$ requires a highly irregular selection of indices of $k$ lowest values from columns of the dynamically computed distance matrix $L^T \cdot X$.
- **Projection step** Computation of the small linear system that is used to find the minimal-error-projection of a point, namely of projections $D_{uv}$ and the derivatives $\frac{\delta d_{uv}}{\delta x_i}$ (Section 2), is difficult to optimize due to irregular memory access patterns of collecting the data for the computation.

In the following two sections, we describe in detail the optimizations of the CUDA implementation.

### 3.1 $k$-NN selection step

The task of the first part of the algorithm is to find $k$ nearest landmarks (from $L$) for every data point in $X$. This comprises two sub-steps: computing Euclidean distances for every pair from $L$ and $X$ and performing point-wise reduction that selects a set of $k$ nearest landmarks for each of the $n$ points, based on the computed distances.

While the Euclidean distance computation is mathematically simple and embarrassingly parallel, achieving optimal throughput on GPUs is quite challenging [10]. In particular, the ratio between the data transfers and the arithmetic operations performed by each GPU core is heavily biased towards data transfers. The overhead of data transfers is best prevented by finding a good caching pattern for the input data that is able to optimally utilize all hardware caches (L1 and L2), shared memory, and core registers.

The parallel implementation of the $k$-nearest neighbors search is even more challenging. The $k$-NN problem is computed individually for each data point, which provides the space for possible parallelization. However, concurrently processed instances of a naïve $k$-NN implementation exhibit severe code divergence because the selection process is purely data-driven, and requires a high amount of memory allocated per core. Optimally, the $k$-NN selection is realized by customized versions of parallel sorting algorithms, which are well-researched and possess existing GPU implementations [13].

Our implementation chooses to optimize both sub-steps since the ratio of the amount of required computations can be easily biased by the configuration of parameters $d$ and $k$. In particular, processing high-dimensional datasets with a low $k$ parameter spends significantly more time in the distance computation, but lower-dimensional datasets with higher $k$ require more time in the nearest neighbor selection.

Concerning the perspective of software design, the implementation may use separate kernels for both sub-tasks or a single fused kernel. Kernel separation provides better code modularity and much flexibility in work-to-thread division and data caching strategy, at the cost of having to materialize all the computed distances in the GPU global memory, thus significantly increasing the total amount of data transfers. Contrary to that, a fused kernel may immediately utilize the computed distances in $k$-NN computation without transferring the data to global memory and interleaving the distance computations with $k$-NN may help to improve the ratio between computations and data transfers.

Since our initial observations showed that the overhead of the data transfers required for kernel communication is relatively high, we decided to implement only the fused variant for the sake of simplicity. The usage of separate kernels might be interesting in the future, especially for extreme values of $d$ that diminish the relative cost of the distance data transfer.

**3.1.1 Available algorithms for $k$-NN.** There are many approaches to $k$-NN selection, varying in complexity and parameter-dependent performance. We implemented several of the possibilities (as described in this section) to substantiate our choice of the algorithm for GPU EmbedSOM.

As a baseline (labeled **Base**), we used the most straightforward approach to GPU parallelization which simply invokes original sequential code for every data point concurrently. The Base kernel is spawned in $n$ threads (one for each data point), and each thread computes the distance between its data point and all landmarks while maintaining an ordered array of $k$ nearest neighbors. The array is updated by an insert-sort step performed for every new computed distance — i.e., by starting at the end of the array and moving the new distance-index pair towards smaller values until it reaches the correct position.

**Shared** algorithm is a modified version of the baseline algorithm that utilizes shared memory as a cache, following the recommended optimization practice of improving performance by caching data that are reused multiple times [12]. In this case, we cache the landmark coordinates, which are sufficiently small to fit in the shared memory for all tested parametrizations.

In **GridInsert** algorithm, we utilize the shared memory to cache both landmarks and points. However, the limited size of shared memory imposes limitations of the amount of cached data. Hence, the algorithm was parametrized by the block height $h$ (number of cached points from $X$) and the block width $w$ (number of cached landmarks from $L$). The algorithm runs in epochs, each of which first caches $h$ points and $w$ landmarks, and then computes $h \cdot w$ distance values using only data in shared memory. While the distances are computed concurrently by the whole thread block, we chose to avoid explicit synchronization in the $k$-NN step, using only $h$ threads to incorporate the newly computed distances into $h$ separate $k$-NN results using the insert-sort steps. The GridInsert should achieve better throughput in the distance computation thanks to the caching, at the cost of slightly suboptimal $k$-NN reduction; thus, giving the best performance on high-dimensional datasets and low values of $k$.

Finally, improvising on our previous work [8], we implemented **Bitonic** $k$-NN selection algorithm, which utilizes routines from the highly parallelizable bitonic sorting algorithm. Bitonic sorting is very suitable for parallel lockstep execution [10], and the capability to merge sorted sequences has allowed us to keep only $2k$ distances (instead of $g$) in the shared memory. This method benchmarked the best on the average, so it is selected as default for EmbedSOM and we describe it more thoroughly in the following.

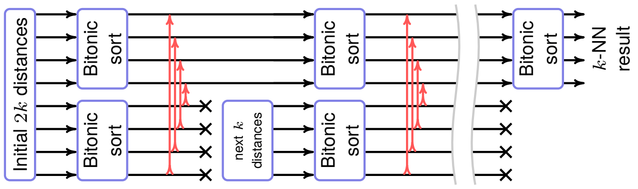

**Figure 1.** Bitonic algorithm for $k$-NN selection ($k = 4$). Each horizontal line represents a data item in the shared memory. Red lines represent comparators ensuring, that the intermediate $k$ 'best' neighbors and in the top buffer.

**3.1.2 Bitonic approach to $k$-NN.** The Bitonic approach can be seen as a combination of the benefits of the other algorithms: It does not require materializing all distances in the memory to do a full sort and even though it does not use an elaborate input caching strategy like GridInsert, it still gives interesting results because the data loading operations can be partially overlapped with bitonic sorting operations if enough warps are allocated to one streaming multiprocessor.

The bitonic comparator network provides a building block that, given two buffers of size $k$ of neighbor distances sorted by bitonic sort, selects the closest $k$ of the neighbors in a single (parallel) operation, allowing us to quickly discard neighbors that do not belong into the $k$-neighborhood. Applying this operation iteratively on $k$-sized blocks of distances sorted by the bitonic sort (as shown in Figure 1), we obtain a highly performing scheme that requires only $2k$ items present in the shared memory. In particular, the shared memory always contains a $k$-block of distances (and corresponding indexes) that holds $k$ so-far-nearest neighbors, and one block of $k$ distances that are computed from $L$; in each iteration, both blocks are sorted by the bitonic sorter in parallel and merged by the bitonic comparator to move the distances of new nearest $k$ neighbors into the intermediate block. The other block is then re-filled by a new set of $k$ distances from $L$.

Technically, each step of the sorting net requires $\frac{k}{2}$ comparators, thus optimally $\frac{k}{2}$ threads that work concurrently on the $h$-sized block. Hence, we allocate $k$ threads for each data point, which alternate their work between computing a block of $k$ distances and performing two bitonic sorts on two $k$-sized blocks in parallel. For simplicity, our implementation assumes that $k$ is always a power of 2, and excessive output of the sorter is discarded.

### 3.2 Projection step

The second part of the dimensionality reduction method is the actual projection into the low-dimensional space. The computation of the low-dimensional point position $x_i$ by EmbedSOM involves: (1) Conversion of the distances collected in the $k$-NN to scores; (2) Orthogonal projection of $X_i$ to $\binom{k}{2}$

lines generated by the $k$ neighbors to create contributions to the final approximation matrix; (3) Solution of the resulting small linear system using Cramer's rule.

Since the first and the last steps are embarrassingly parallel problems with straightforward optimal implementation and since the second step is the most time demanding (performing $O(k^2)$ operations on vectors of size $d$), we focus mainly on the orthogonal projections. Its computation is complicated by a highly irregular pattern of repeated accesses to an arbitrary $k$-size subset of $L$. We designed several algorithms that successively optimize the access patterns, detailed below.

The baseline algorithm **Base** uses the most straightforward parallel approach (similar to Base $k$-NN), where each thread computes the projection of one single point sequentially so the concurrency is achieved only by processing multiple points simultaneously. All data are stored in the global memory, and no explicit cache control is performed.

The irregular repeated access to the elements of $L$ hinders the performance of the baseline algorithm. In the **Shared** algorithm, we chose to reorganize the workload so that each projection is computed by a whole block of threads that cooperatively iterate over the landmark pairs. As a result, the input data of the orthogonal projection (i.e., the $k$ nearest neighbors from $L$ together with the distances, scores, and 2D versions of the landmarks) can be cached in shared memory. The intermediate sub-results represented by $2 \times 3$ matrices are successively added into privatized copies of each thread to avoid explicit synchronization and aggregated at the end using a standard parallel reduction, enhanced with warp-shuffle instructions (a similar scheme is used in optimal CUDA k-means implementation [9]).

Because the data transfers comprise a considerable portion of the Shared algorithm execution time, we have optimized the transfers using alignment and data packing techniques, yielding the **Aligned** algorithm. The implementation is based on using vector data types (e.g. `float4` in CUDA) to enable utilization of 128-bit load/store instructions, which improves overall data throughput. The vectorization comes only at a relatively small cost of aligning and padding the vectors to 16-byte blocks.

To further improve the data caching, we implemented algorithm **Registers**, where each thread computes more than one landmark pair in a single iteration so that the coordinates loaded into its registers can be shared as inputs among multiple landmark-pairs computations. The data sharing scheme is detailed in Figure 2. We found that it is optimal to group the threads into small blocks of $2 \times 2$ computation items, saving half of the data loads. Larger groups are theoretically possible, but even $3 \times 3$ caused excessive registry pressure and impaired performance on contemporary GPUs. The innermost loop of the algorithm iterates over $d$ so that only a single `float4` value per each landmark is kept in registers.

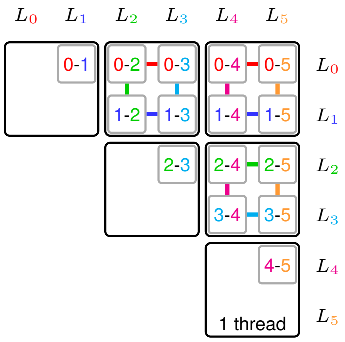

**Figure 2.** Detail of the caching of landmark data in Registers projection kernel. Multiple landmark pairs (small boxes) are processed by each thread (large boxes). Caching of the landmark data in registers allows the reuse of loaded data (color lines), thus reducing the amount of memory accesses.

## 4 Experimental Results

The main objective of the benchmarking was to measure the speedups achieved by different applied optimizations and to determine the optimal algorithms and their parameter setting for the sub-tasks of EmbedSOM computation.

The timing results, presented in the following sections, were collected as kernel execution times measured by a standard system high-precision clock. Each test was repeated 10× and the mean values are presented in the subsequent figures. The relative standard deviations of the measurements were less than 5% so we chose not to include them. Complete measurements are available in our GitHub repository[2].

Results were collected on NVIDIA Tesla A100 PCIe 80 GB running CUDA 12.2. All benchmarking datasets were synthetic, containing exactly 1Mi points ($n = 2^{20}$, reflecting the common sizing of real-world datasets [1]) with all coordinates sampled randomly from the same uniform distribution. The performance of the benchmarked algorithms is not data-dependent, except for the case of caching performance in the projection step, where the completely random dataset is the worst-case scenario.

### 4.1 Performance of $k$-NN selection

Here we give an overview of performance and viable parameter settings observed for the $k$-NN selection algorithms.

Notably, all algorithms for $k$-NN are affected by CUDA thread block sizing which affects warp scheduling and data reuse possibilities of the shared-memory cache. We observed that the total thread block size of 256 threads was either optimal or near to optimal for almost all tested configurations, except for GridInsert that performed the best with 64 threads for lower values of $d$ and $g$ parameters.

[2] https://github.com/asmelko/gpgpu24-artifact

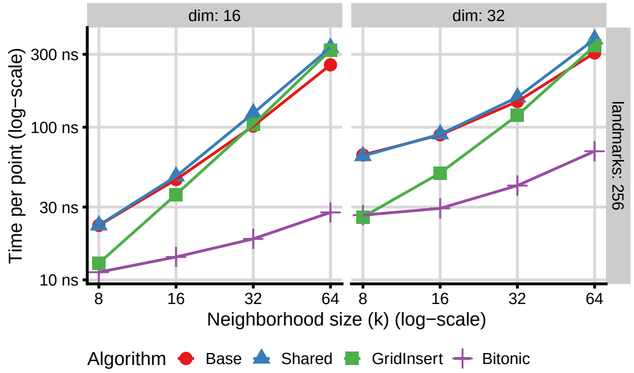

**Figure 3.** Amortized performance of $k$-NN step for a single input point using parameters usual in flow cytometry

Parameters $w$ and $h$[3] of the GridInsert algorithm determine the ratio between data transfers and computations, but may also affect the pressure on the shared memory. Empirical evaluation indicates that the algorithm performs the best when each parallel insertion sort is performed in a separate warp, so the code divergence in SIMT execution is prevented (i.e., $w$ is a multiple of 32) The optimal performance was observed for $w$ equal to 96 or 128; However, the speedup over $w = 32$ is relatively low.

A comparison of the best parametrizations of each algorithm on various configurations common in our target use cases is shown in Figure 3. The Bitonic algorithm significantly outperformed the other algorithms. The speedup of Bitonic over Base was between 3× to 20× and usually more than 2× over the second-ranking method.

The benchmarking also confirmed a rather huge scaling difference between algorithms based on divergent insertion sort and algorithms based on sub-quadratic parallelizable sorting schemes. We conclude that despite the simplicity that might enable GPU speedups in certain situations, the insertion sort is too slow for larger values of $k$ in this case.

As an interesting result, we observed that despite following the general recommendations, the straightforward use of shared memory (in the Shared algorithm) did not improve overall performance over the Base. Quite conversely, the overhead of explicit caching even caused a slight decrease in the overall performance.

We additionally report the performance measurements for two selected corner cases with extreme values of $g$ and $d$ (figure omitted due to the page limit). Mainly, the total volume of the computation required to prepare the Euclidean distances scales with $g \cdot d$, which becomes dominant when both are maximized. At that point, we observed that GridInsert provides comparable or mildly better performance than Bitonic, especially in cases where $k$ is small and the overhead of insertion sorting is not as pronounced.

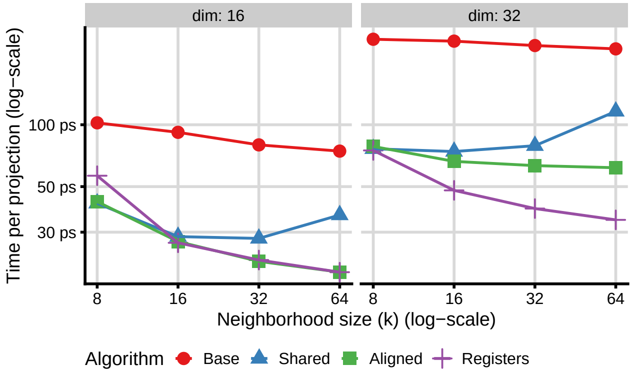

**Figure 4.** Amortized performance of a single projection operation in the algorithms that compute the projection step (showing the most important problem parametrizations)

Naturally, we should ask whether it could be feasible to combine the benchmarked benefits of GridInsert and Bitonic algorithms in order to get the best of both approaches (optimal inputs caching and fast $k$-NN filtering). While an investigation of this possibility could be intriguing, we observed that a fused algorithm would require very complicated management of the shared memory (which both algorithms utilize heavily), and the estimated improvement of performance was not sufficient to substantiate this overhead; we thus left the question open for future research.

### 4.2 Performance of projection step

The projection algorithms described in the previous section have only two execution parameters: The size of the CUDA thread block and the number of data points assigned to a thread block (threads are divided among the points evenly). We observed that selecting more than one point per thread block is beneficial only in the case of relatively small problem instances (low $k$ and $d$) because it prevents underutilization of the cores.

The optimal size of the CUDA thread blocks depends mainly on the parameters $k$ and $d$. In case of Shared algorithm, optimal values ranged from 32 (for $k = 8$, $d = 4$) to 64 ($k = d = 64$). With the caching optimizations in Aligned and Registers, the optimal thread block size was slightly higher, reaching 128 for the most complex problem instances. We assume this is a direct consequence of the improved memory access efficiency which gives space for parallel execution of additional arithmetic operations.

Figure 4 shows the performance of the best algorithm configurations for the representative parametrizations. All three algorithms perform almost equally for small $k$, giving around 3× speedup over Base. The importance of optimizations in Aligned and Registers grows steadily when parameter $k$ increases, up to around 10× speedup at $k = 64$. In conclusion, the optimal algorithm for the EmbedSOM projection

[3] Technically, parameter $h$ is determined by the thread block size divided by $w$, we thus optimize only $w$.

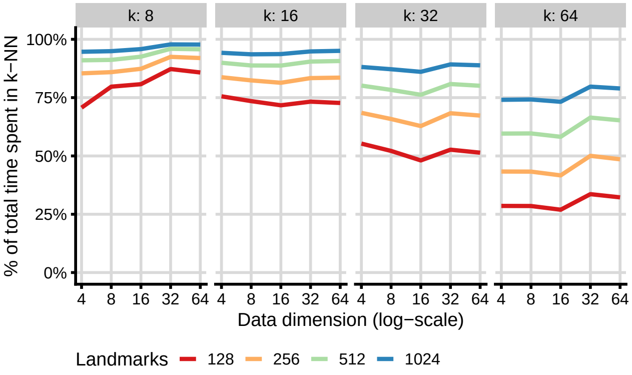

**Figure 5.** The relative time spent by the $k$-NN computation usually dominates the execution of GPU EmbedSOM, composed of Bitonic+Registers algorithms. Projection computation time becomes dominant only for relatively impractical parametrizations of low $g$ and high $k$.

is determined by the dimensionality of the dataset — Registers performs better at higher dimensions ($d \geq 32$) while Aligned was slightly better for lower dimensions.

### 4.3 Complete algorithm

A complete GPU implementation of the EmbedSOM algorithm is the combination of the best implementations of $k$-NN and projection steps. The selected algorithms Bitonic and Registers are simply executed sequentially on large blocks of $X$, sharing only a single data exchange buffer for transferring the $k$-NN data. Notably, since the data exchange between the algorithm parts is minimal, comprising only distances and neighbor indexes from the $k$-NN selection, we claim that no specific optimizations of the interface are required.

Finally, we highlight the relative computation complexity of both steps (Figure 5), which changes dynamically with $k$ and might be viable as a guide for further optimization. We observed that for common parametrizations ($k \simeq 20$, $g \simeq 500$), most of the computation time is spent in $k$-NN step, and projection performance becomes problematic only in cases of almost impractically high $k$. The results align with the asymptotic time complexities of the algorithms, roughly following $O(n \cdot d \cdot g \cdot \log_2 k)$ for the $k$-NN and $O(n \cdot d \cdot k^2)$ for the projection.

## 5 Related work

The essential component of our success is GPU acceleration of the projection computation which needs to be fast enough to re-calculate the embedding in real-time. In the following, we address the most relevant works that influenced or inspired our solution.

Being one of the most profound visualization methods, *t-SNE* was studied to explore the possibilities of having a fast GPU-enabled implementation. One of the initial implementations was t-SNE-CUDA library [3]. The most complicated step (computing the attractive forces of the N-body simulation) is handled as a multiplication of a sparse matrix and a vector by the CUSparse library. This work was slightly improved a year later [4] when the authors replaced the CUSparse library with their implementation of multiplication, which takes advantage of atomic operations to perform the reduction in scalar sums.

Perhaps the most popular contemporary method for data visualization is the *Uniform Manifold Approximation and Projection* algorithm (UMAP), which often produces better results than t-SNE at the cost of higher computational demands. There are two GPU implementations worth mentioning which were both made part of RAPIDS cuML library [11, 14]. They both use a similar approach, implementing a kNN approximation based on gradient descent methods. The first implementation [14] relies more on existing solutions and libraries, and the second one [11] is slightly more low-level as they implement the embedding using custom kernels.

Even though the presented methods (especially t-SNE) exceeded the speedup of two orders of magnitude, they are still quite far from real-time processing when the number of points reaches the order of millions. The proposed EmbedSOM projection is based on SOMs and linear projection based on $k$NN search [6, 7], which is technically closest to the work of Yeh et al. [17]. For the SOM part, we have adapted the state-of-the-art implementation of $k$-means algorithm [9] since SOM shares many of its steps. The crucial part of the projection is the $k$NN search, which is also repeated in the aforementioned papers; however, we have found that the solution based on bitonic-sorting [8] performs the best in our case.

## 6 Conclusion

We have presented a GPU implementation for the semi-supervised dimensionality reduction algorithm EmbedSOM where we optimized independently two kernels: A general $k$NN search and a 2D projection which may be used independently. The $k$-NN was solved by adapted bitonic sorting, which eliminates thread divergence. The projection kernel was optimized to fetch and use data most efficiently by utilizing vector loads and data reuse on the register level. A thorough benchmarking indicates that both kernels achieved a significant speedup over the baseline GPU implementation.

The proposed implementation should enable subsequent research in interactive dimensionality reduction tools where the user changes SOM parameters or landmarks and the projections are re-computed and visualized in real-time. The results show that the optimized EmbedSOM version can project more than 1 million individual data points each frame, while maintaining a frame rate above 30fps.

## Acknowledgments

This paper was supported by Charles University institutional funding SVV 260698/2023.

## References

[1] Aysun Adan, Günel Alizada, Yağmur Kiraz, Yusuf Baran, and Ayten Nalbant. 2017. Flow cytometry: basic principles and applications. *Critical reviews in biotechnology* 37, 2 (2017), 163–176.

[2] Jessica Cande, Shigehiro Namiki, Jirui Qiu, Wyatt Korff, Gwyneth M Card, Joshua W Shaevitz, David L Stern, and Gordon J Berman. 2018. Optogenetic dissection of descending behavioral control in Drosophila. *Elife* 7 (2018), e34275.

[3] David M Chan, Roshan Rao, Forrest Huang, and John F Canny. 2018. T-SNE-CUDA: GPU-Accelerated T-SNE and its Applications to Modern Data. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE Computer Society, 330–338.

[4] David M Chan, Roshan Rao, Forrest Huang, and John F Canny. 2019. GPU accelerated t-distributed stochastic neighbor embedding. *J. Parallel and Distrib. Comput.* 131 (2019), 1–13.

[5] Teuvo Kohonen. 1990. The self-organizing map. *Proc. IEEE* 78, 9 (1990), 1464–1480.

[6] Miroslav Kratochvíl, David Bednárek, Tomáš Sieger, Karel Fišer, and Jiří Vondrášek. 2020. ShinySOM: graphical SOM-based analysis of single-cell cytometry data. *Bioinformatics* 36, 10 (2020), 3288–3289.

[7] Miroslav Kratochvíl, Abhishek Koladiya, and Jiří Vondrášek. 2019. Generalized EmbedSOM on quadtree-structured self-organizing maps. *F1000Research* 8, 2120 (2019), 2120. [version 2; peer review: 2 approved].

[8] Martin Kruliš, Hasmik Osipyan, and Stéphane Marchand-Maillet. 2015. Optimizing sorting and top-k selection steps in permutation based indexing on gpus. In *East European Conference on Advances in Databases and Information Systems*. Springer, 305–317.

[9] Martin Kruliš and Miroslav Kratochvíl. 2020. Detailed Analysis and Optimization of CUDA K-means Algorithm. In *49th International Conference on Parallel Processing (ICPP '20)*. ACM.

[10] Martin Kruliš, Hasmik Osipyan, and Stéphane Marchand-Maillet. 2017. Employing GPU architectures for permutation-based indexing. *Multimedia Tools and Applications* 76, 9 (2017), 11859–11887.

[11] Corey J Nolet, Victor Lafargue, Edward Raff, Thejaswi Nanditale, Tim Oates, John Zedlewski, and Joshua Patterson. 2020. Bringing UMAP Closer to the Speed of Light with GPU Acceleration. *arXiv preprint arXiv:2008.00325* (2020).

[12] NVIDIA. 2013. CUDA C Best Practices Guide.

[13] Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. 2018. Survey of GPU based sorting algorithms. *International Journal of Parallel Programming* 46, 6 (2018), 1017–1034.

[14] Yezihalem Tegegne, Zhonglin Qu, Yu Qian, and Quang Vinh Nguyen. 2021. Parallel Nonlinear Dimensionality Reduction Using GPU Acceleration. In *Australasian Conference on Data Mining*. Springer, 3–15.

[15] Shadi Toghi Eshghi, Amelia Au-Yeung, Chikara Takahashi, Christopher R Bolen, Maclean N Nyachienga, Sean P Lear, Cherie Green, W Rodney Mathews, and William E O'Gorman. 2019. Quantitative comparison of conventional and t-SNE-guided gating analyses. *Frontiers in immunology* 10 (2019), 1194.

[16] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[17] Tsung Tai Yeh, Tseng-Yi Chen, Yen-Chiu Chen, and Wei-Kuan Shih. 2010. Efficient parallel algorithm for nonlinear dimensionality reduction on GPU. In *2010 IEEE International Conference on Granular Computing*. IEEE, 592–597.