

Une implémentation GPU de la méthode de recherche approximative FlyHash

Arthur da Cunha, Damien Rivet, Emanuele Natale, Aurora Rossi

▶ **To cite this version:**

Arthur da Cunha, Damien Rivet, Emanuele Natale, Aurora Rossi. Une implémentation GPU de la méthode de recherche approximative FlyHash. CAID 2023 - 5e Conference on Artificial Intelligence for Defense, DGA Maîtrise de l'Information, Nov 2023, Rennes, France. hal-04328529

HAL Id: hal-04328529
https://hal.science/hal-04328529v1

Submitted on 7 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Une implémentation GPU de la méthode de recherche approximative FlyHash

Arthur da Cunha*
COATI, I3S & INRIA d'Université Côte d'Azur
Sophia Antipolis, France
arthur.carvalho-walraven-da-cunha@inria.fr

Emanuele Natale*
COATI, I3S & INRIA d'Université Côte d'Azur
Sophia Antipolis, France
emanuele.natale@inria.fr

Damien Rivet*
COATI, I3S & INRIA d'Université Côte d'Azur
Sophia Antipolis, France
damien.rivet@inria.fr

Aurora Rossi*
COATI, I3S & INRIA d'Université Côte d'Azur
Sophia Antipolis, France
aurora.rossi@inria.fr

***Abstract*—FlyHash is a locality-sensitive hashing algorithm inspired by the nervous system of the Drosophila fly. It has demonstrated to be particularly effective for similarity search, especially in the federated context where multiple players collaborate to solve a statistical learning task. FlyHash mainly relies on a process called winner-take-all, which is used to binarize information. However, the implementation of this process is a major challenge and limits the algorithm's usage in processing large data streams. In this paper, we propose a simple algorithm to make the winner-take-all operation efficient on GPUs. We create a FlyHash adaptation suitable for the CUDA architecture. We assess the speed of this version experimentally and present a comparison with the CPU version of FlyHash.**

***Index Terms*—Data mining, Hashing, winner-take-all, Distributed algorithms, Federated learning, GPU**

I. Introduction

Locality-sensitive hashing (LSH) is a technique in computer science introduced in [IM98] for hashing data so that similar data has a high probability of having similar hashes, while different data is likely to have different hashes. It is widely used, especially for similarity search in large databases using faster heuristics than traditional approaches, such as nearest neighbor searching. Locality-sensitive hashing is particularly effective in real-time applications, where the speed of similarity search is essential to handle a massive and continuous flow of incoming data.

In nature, animals are constantly faced with similarity recognition tasks. This is notably the case for fruit flies, which, when encountering new odors, seek to identify similarities with odors they have previously encountered in order to assess the potential quality of the available food. Observation of their olfactory nervous system revealed that some of these neural circuits bore striking similarities to well-known LSH algorithms. Based on these observations, researchers propose a new type of algorithm called FlyHash [SDN17].

FlyHash is based on the use of random projections followed by a binarization process, as is notably the case of one of the most well-known LSH heuristics SimHash [Cha02]. However, unlike SimHash, the binarization used by FlyHash is not based on thresholding, but on a process called winner-take-all (WTA), which we describe in detail below.

The adoption of winner-take-all is motivated on the one hand by its theoretical properties [YSRL11], and on the other hand for modelling the brain, in particular in the model of *Assemblies of Neurons* proposed by [PVM$^+$20] as well as in *Spiking Neural Networks* [Che17], artificial neural networks that are biologically closer to the real ones.

Besides, the classical winner-take-all implementation is known to represent a major bottleneck when one wants to process multiple data at once, or using large hashlengths (indeed the accuracy of such hashing algorithms improves significantly with the increase of the size of the generated hashes). In addition, the majority of data mining applications are now massively parallelized, via the use of distributed algorithms and the dominant hardware architecture is now the Graphics Processing Unit (GPU).

Our contribution with this work is to demonstrate that such WTA hashing schemes are compatible with the GPU architecture, allowing to implement them on a pipeline fully executable on GPU (Section III-B). Some works have focused on GPU implementation of winner-take-all including [MVSG$^+$09], but this direction remains relatively unexplored.

II. Motivations and applications

The FlyHash has been first studied in [SDN17], based on empirical observations of the functioning of the olfactory system of a fruit fly. In this section, we first provide a brief summary of such biological observations. We refer the reader to [SDN17] for a more detailed description. A formal description of the FlyHash algorithm inspired by these observations, together with our new implementation, is given in section

This work has been supported by the French government, through the UCA DS4H Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-17-EURE-0004, and through the AID INRIA-DGA agreement n°2019650072. The authors are grateful to the OPAL infrastructure from Université Côte d'Azur for providing resources and support.

* Authors are listed in alphabetical order.

III-A. Subsequently, we illustrate two important applications of FlyHash in Machine Learning, namely in an efficient and secure classification scheme in Federated Learning and as a potential subroutine to speed up the training of neural networks.

a) Neurobiological basis for Flyhash: The neurons which are activated by a given odor are determined by a three step procedure which is exemplified in Fig. 1. The first step

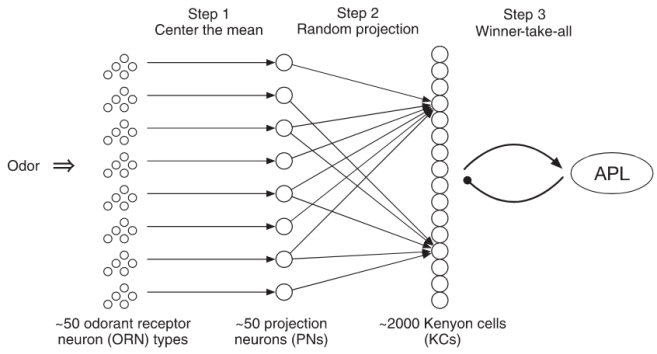


Fig. 1. The three steps of the olfactory system of a fruit fly as illustrated in [SDN17].

consists in direct nervous connections from odorant receptor neurons (ORNs) in the fly's nose to projection neurons (PNs) in particular structures called glomeruli. There are circa 50 ORN types which are activated by different odors. Thus, each input odor can be thought as having a corresponding location in a 50-dimensional space determined by the 50 ORN. For each odor, the distribution of ORN firing rates across the 50 ORN types follows an exponential distribution with a mean that depends on the concentration of the odor [SDN17]. For the PNs, such distribution of firing rates across the 50 PN types is also exponential, but for all odors and all odor concentrations the mean is approximately the same. The second step, which is where neurobiology offer us the main algorithmic insight, then involves an expansion in the number of neurons by a factor roughly 40: Fifty PNs project to 2000 so-called Kenyon cells (KCs), connected by a sparse binary random connection matrix, with each KC receiving and summing the firing rates from about six randomly selected PNs. The final third step is then performed by strong inhibitory connections from a single inhibitory neuron, called APL (anterior paired lateral neuron), which results in the aforementioned winner-take-all (WTA) operation. As a consequence, all but the highest-firing KCs are silenced, with the firing rates of the still-active KCs corresponding to the neurons activated by the input odor.

b) kNN-like classification in Federated Learning: FlyHash has recently found an important application as the basis of the FlyNN algorithm introduced in [SR21], who exploited the work of [DSSN18], taking advantage once again of biological observations, to design a classification algorithm (see Fig. 2).

FlyNN has been deployed in the context of federated learning in [RS22] and is currently the state of the art ap-

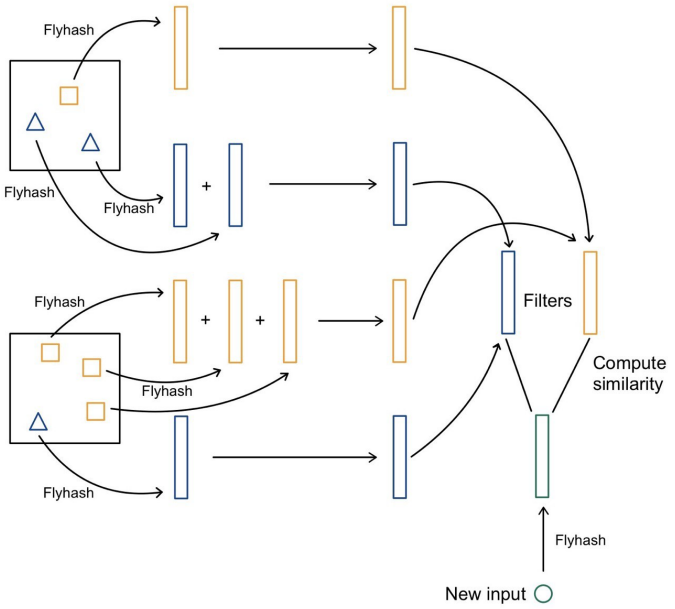


Fig. 2. Illustration of the FlyNN scheme to perform approximate k-Nearest Neighbors classification using FlyHash [RS22]. Each party, here represented by a square, hashes the data point it owns for each class (here orange squares and blue triangles) and sums them into a single hash vector. The parties then combine their hash vectors to obtain a single hash vector for each class across the entire dataset, called *filter*. The class of a new data point is then determined by the filter that has the highest similarity to the new data point's hash vector.

proximation of the k-nearest neighbor classification algorithm in the federated setting. FlyNN has in particular the advantage of being usable in the context of one-shot federated learning, where the communication among clients is restricted to a single round, and to be readily combinable with differential privacy schemes [RS22].

c) Speeding up neural network training via Locality-sensitive Hashing: An important application of LSH schemes has recently been provided in [DMZS21], in which such schemes are leveraged to speed up training and inference of large artificial neural network architectures on the CPU, based on the empirical observation that only a small fraction of neurons is activated per layer. A simplistic diagram of the SLIDE framework proposed in [DMZS21] is outlined in Fig. 3. The purpose of the present work is not only to increase the efficiency of the FlyHash algorithm as an alternative that can be leveraged in SLIDE, but also to make it possible to leverage the aforementioned framework to speed up training on the GPU.

III. Description of the Algorithms

A. FlyHash

Formally, the FlyHash algorithm takes as input a vector in $\mathbb{R}^d$ and returns its hash, which is a binary vector of length N (the *hashlength* parameter). The algorithm has two main parts, a projection and a winner-take-all binarization part.

The projection matrix is a random binary matrix M of size $N \times d$ with a fixed number s (the *projection* parameter) of ones

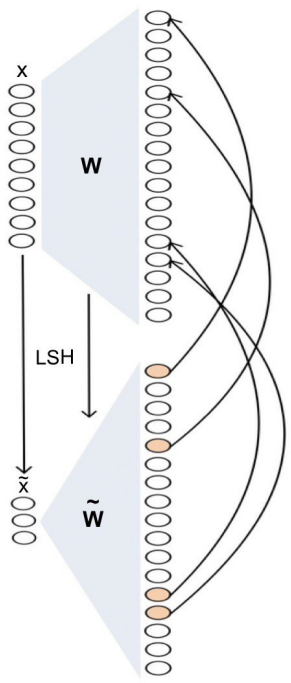


Fig. 3. Diagram illustrating the core idea of the SLIDE framework, in which activating neurons are predicted by computing the scalar product between the projection of the input $\tilde{x}$ and the projection of the layer weight matrix $\tilde{W}$ via a Locality-sensitive Hashing scheme. The actual input of the neurons that are predicted to activate is then computed via the original input x and the corresponding rows of the original weight matrix W.

in each row. The first part of the algorithm is the multiplication between M and the input vector.

The winner-take-all binarization is then applied to the outcome of the previous step, which is in $\mathbb{R}^N$, and transformed into a $\{0,1\}^N$ vector by setting the k highest entries to one and the others to zero. The parameter k is also called *number of winners* parameter.

Algorithm 1 contains the FlyHash pseudocode. The WTA function is explained in detail in the next section.

Algorithm 1 FlyHash

Input: $X \in \mathbb{R}^{d\times b}$, $M \in \{S \in \{0,1\}^{N\times d} :$ each row of S contains s ones$\}$, $k \in [1, N]$
Output: $X \in \{0,1\}^{N\times b}$
$A = M \times X$
return WTA(A, k)

B. A parallelized winner-take-all

We implement the FlyHash algorithm on the GPU to process large amounts of data. Our main contribution is a parallelized winner-take-all binarization algorithm that, rather than taking as input a single vector as mentioned before, processes a batch of vectors in a matrix X of size $N \times b$, where b is the batch size. The binarization step is thus applied to each column simultaneously. More specifically, we perform a parallel binary search for the values that, when used to threshold the respective columns, give the desired number of ones k.

Algorithm 2 contains the winner-take-all pseudocode. It starts by computing, for each column, the lower bound lb and the upper bound ub of the search interval by taking, respectively, the minimum and the maximum with a small margin $\varepsilon > 0$ to allow for strict inequalities. It then calculates the middle value mid and updates the extremes according to the current number of ones tot (corresponding to the number of values greater than mid): if they are greater than k, we increase the lower bound of the interval by setting it equal to the middle value; instead, if they are less than k, we decrease the upper bound to be equal to the middle value. The process is repeated a given number of times, which is at most 278 for single-precision floats, but in practice it can be set to 64 if a small fraction of erroneous entries can be tolerated (for example, the average fraction of erroneous entries caused by such a limitation is around 2.095×10^{-7} when the output is of size 20000×5000).

Algorithm 2 Winner-take-all (WTA). Functions preceded or followed by a dot (Julia's broadcasting operator) are applied element-wise.

Input: $X \in \mathbb{R}^{N\times b}$, $k \in [1, N]$
Output: $X \in \{0,1\}^{N\times b}$
$lb = \texttt{minimum}(X,\ \text{dims} = 1). - \varepsilon$
$ub = \texttt{maximum}(X,\ \text{dims} = 1). + \varepsilon$
$mid = (lb. + ub)./2$
for _ in 1 : 64 **do**
$tot = \texttt{count}(X. > mid,\ \text{dims} = 1)$
$lb = \textbf{ifelse}.(tot. > k, mid, lb)$
$ub = \textbf{ifelse}.(tot. < k, mid, ub)$
$mid = (lb. + ub)./2$
end for
return $X. > mid$

IV. Experiments

In our experiments, we compare the performance of the FlyHash algorithm on an `Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz` CPU and a `NVIDIA Quadro RTX 8000` GPU with `CUDA version: 11.8`, focusing on the processing of large amounts of data. We implemented the algorithms in the Julia programming language, which is now one of the most popular languages for scientific computing, and we relied on CUDA.jl package for the GPU part. On the algorithm engineering side, some optimizations have been made to speed up the process, such as pre-allocating variables for the GPU version. As for the CPU implementation, it uses efficient partial sorting algorithms to select the k winners [RS22]. Our code is available in the following Github repository: https://github.com/AInnervate/flyhash.jl, along with the code to replicate the experiments explained in more detail below.

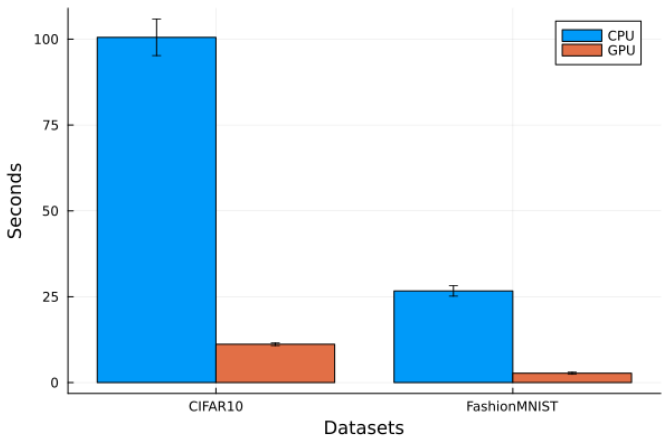


Fig. 4. Comparison on two popular datasets. Hash factor parameter is set to $h = 32$.

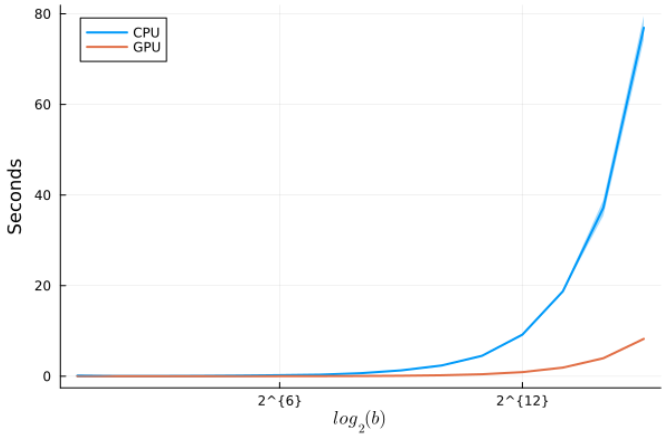


Fig. 5. Comparison as batchsize increases with fixed hash factor $h = 128$ and input dimension $d = 1024$.

First, we test the speed of the two versions of the algorithm on well-known datasets taken from the Machine Learning Datasets Julia library MLDatasets.jl. One of the two datasets we examine is the FashionMNIST dataset, which is a collection of 60000 greyscale images with a size of 28×28. We therefore transform each image into a vector of dimension $d = 28 \times 28 = 784$ and collect them in a matrix of dimension 784×60000. Then we pass it as input to the FlyHash algorithm, choosing the following parameters according to previous work [SDN17]: the hash length N is equal to the hash factor $h = 32$ multiplied by the input dimension d, the projection parameter s is set to 5% of the input dimension d and the number of winners k is set to 5% of the hash length N. The same configuration is used to compute the time to process the CIFAR10 dataset, which is a collection of 50000 coloured images with a size of 32×32, so the input dimension of the matrix is $d \times b$, where $d = 32 \times 32 \times 3 = 3072$ (the factor 3 comes from the fact that the images are coloured) and the batchsize is $b = 50000$.

In addition to those datasets, we perform experiments on synthetic data, where each entry is uniformly sampled in [0,1]. To understand how the two architectures behave when dealing with large amounts of data, we run the code varying the batchsize b in the range $\{2^1, ..., 2^{15}\}$ and fixing the other parameters: the projection parameter and number of winners are as above, the hash factor is set to $h = 128$ and the input dimension is $d = 1024$.

The results are shown in Figure 4 and Figure 5. They are obtained by averaging 10 independent runs after first running the code without taking the time to avoid Julia's just-in-time compilation overhead. Both plots show low standard deviation values, in the first case as bars and in the second as shadows.

V. Conclusions

In this work, we proposed a GPU implementation of the famous FlyHash locality sensitive algorithm. Experiments show that our GPU version can run one order of magnitude faster than the best CPU version, thus allowing to speed up the use of the FlyHash algorithm in settings where GPUs are more easily available than dozens of CPU cores. In future work, we expect that the GPU code can be further improved in terms of performance by writing a lower-level CUDA kernel.

Important directions consist in exploring the implications of our implementations for the use of FlyHash in real-world applications, such as the ones mentioned in Section II.

References

[Cha02] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing*. ACM, 2002.

[Che17] Yanqing Chen. Mechanisms of winner-take-all and group selection in neuronal spiking networks. *Frontiers in computational neuroscience*, 2017.

[DMZS21] Shabnam Daghaghi, Nicholas Meisburger, Mengnan Zhao, and Anshumali Shrivastava. Accelerating SLIDE Deep Learning on Modern CPUs: Vectorization, Quantizations, Memory Optimizations, and More. *Proceedings of Machine Learning and Systems*, 3:156–166, March 2021.

[DSSN18] Sanjoy Dasgupta, Timothy C. Sheehan, Charles F. Stevens, and Saket Navlakha. A neural data structure for novelty detection. *Proceedings of the National Academy of Sciences*, 2018.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98. Association for Computing Machinery, 1998.

[MVSG+09] O. Moslah, A. Valles-Such, V. Guitteny, S. Couvet, and S. Philipp-Foliguet. Accelerated multi-view stereo using parallel processing capabil-

ities of the gpus. In *2009 3DTV Conference*, 2009.

[PVM+20] Christos Papadimitriou, Santosh Vempala, Daniel Mitropolsky, Michael Collins, and Wolfgang Maass. Brain computation by assemblies of neurons. *Proceedings of the National Academy of Sciences of the United States of America*, 2020.

[RS22] Parikshit Ram and Kaushik Sinha. Federated nearest neighbor classification with a colony of fruit-flies. In *Thirty-Sixth AAAI Conference on Artificial Intelligence*. AAAI Press, 2022.

[SDN17] Charles F. Stevens Sanjoy Dasgupta and Saket Navlakha. A neural algorithm for a fundamental computing problem. *Science*, 2017.

[SR21] Kaushik Sinha and Parikshit Ram. Fruit-fly inspired neighborhood encoding for classification. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, 2021.

[YSRL11] Jay Yagnik, Dennis Strelow, David A. Ross, and Ruei-sung Lin. The power of comparative reasoning. In *2011 International Conference on Computer Vision*, 2011.