Review of : Kernel methods through the roof: handling billions of points efficiently

Fotios Kapotos - Tristan Beruard
CentraleSupélec
France
fotiskapotos@gmail.com
tristanberuard@gmail.com

ABSTRACT

Kernel methods offer strong statistical guarantees but become computationally impractical on large datasets due to the quadratic storage and cubic time complexity of kernel matrix operations. The Falkon algorithm addresses these limitations by combining Nyström approximations, preconditioned conjugate gradient solvers, and GPU-oriented system design to scale kernel learning to billions of samples. In this review, we summarize the theoretical foundations of these methods, provide a detailed analysis of the Falkon algorithm, its efficient implementation, and some of its limitations along with relevant experiments on toy datasets. In addition, we reproduce selected experimental results from the original paper. Our benchmarks confirm the predicted complexity scaling laws and demonstrate the practical efficiency gains enabled by this approach. Reproducibility code is available at: https://github.com/fotisk07/GDA-Project.

ACM Reference Format:
Fotios Kapotos - Tristan Beruard. 2025. Review of : Kernel methods through the roof: handling billions of points efficiently. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Kernel methods offer a principled and statistically well-understood framework for non-linear learning, but their applicability to large-scale problems is severely limited by the quadratic memory and cubic time complexity induced by the kernel matrix. The recently proposed Falkon algorithm introduces a combination of Nyström approximation, preconditioning, iterative solvers, and GPU-centric system design to enable kernel methods to scale to datasets containing millions to billions of samples.

In this review, we first present the mathematical background underlying supervised learning with kernels and the Nyström approximation, highlighting the tension between statistical optimality and computational feasibility. We then detail the Falkon algorithm and its GPU implementation strategies, with particular emphasis on memory management, out-of-core computation, and multi-GPU parallelism. Finally, we reproduce selected experiments from the original paper and conduct additional benchmarks to empirically validate the predicted scaling laws and performance gains.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

2.1 Supervised Learning and Kernel Methods

In supervised learning, one seeks to learn a predictor $f : \mathcal{X} \to \mathcal{Y}$ from i.i.d. samples $(x_i, y_i)_{i=1}^n \sim \rho$ by minimizing the expected risk

$$L(f) = \int \ell(f(x), y)\, d\rho(x, y),$$

which in practice is replaced by the empirical risk

$$\hat{f} = \arg\min_f \frac{1}{n} \sum_{i=1}^n \ell(f(x_i), y_i).$$

Linear models $f(x) = w^\top x$ are efficient but limited in expressivity. Kernel methods overcome this by implicitly mapping inputs into a high-dimensional feature space via $\Phi(x)$ and considering predictors of the form $f(x) = w^\top \Phi(x)$. This defines a reproducing kernel Hilbert space (RKHS) $\mathcal{H}$ with kernel

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle.$$

Learning is typically posed as regularized Empirical Risk Minimization (ERM),

$$\hat{f}_\lambda = \arg\min_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^n \ell(f(x_i), y_i) + \lambda \|f\|_{\mathcal{H}}^2,$$

whose solution, by the representer theorem, admits the finite expansion

$$\hat{f}(x) = \sum_{i=1}^n \alpha_i k(x, x_i).$$

This formulation enables flexible nonlinear modeling with strong statistical guarantees, as illustrated by the Kernel Ridge Regression examples in Fig. 1, where the same kernel model successfully fits both smooth global structure and localized nonlinear targets without prior assumptions. However, this representation requires forming and factorizing the full kernel matrix K_{nn}, inducing $O(n^2)$ memory and $O(n^3)$ time complexity. As a result, standard kernel solvers become impractical at large scale, motivating the approximate methods studied in this work.

2.2 Statistical Guarantees vs Computational Bottlenecks

Kernel methods enjoy strong theoretical properties. Under mild regularity assumptions and with regularization parameter λ =

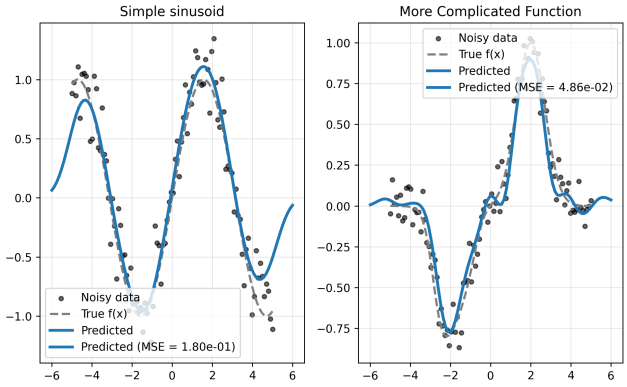

Figure 1: Kernel Ridge Regression applied to $f(x) = \sin(x)$ (left) and to a localized nonlinear target $f(x) = e^{-(x-2)^2} - 0.8\, e^{-(x+2+\sin(2x))^2}$ (right) using a Gaussian Kernel

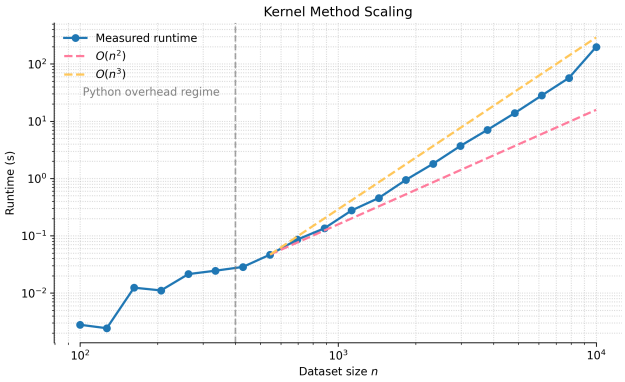

Figure 2: Vanilla KRR Scaling Law on *sin* function

$O(n^{-1/2})$, solutions of the regularized ERM problem achieve the learning rate [5]

$$L(\hat{f}_\lambda) - \inf_{f \in \mathcal{H}} L(f) = O(n^{-1/2}), \tag{1}$$

which is known to be optimal for a large class of problems. This statistical efficiency is one of the main reasons for the sustained interest in kernel-based learning.

However, the practical deployment of kernel methods is severely limited by their computational cost. In order to make the computational bottleneck explicit, we consider the squared loss case, which corresponds to kernel ridge regression (KRR). The regularized ERM problem becomes

$$\hat{f} = \arg\min_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} (f(x_i) - y_i)^2 + \lambda \|f\|_{\mathcal{H}}^2.$$

The representer theorem reduces the optimization problem to estimating the coefficient vector $\alpha \in \mathbb{R}^n$. Plugging this form into the objective yields the quadratic optimization problem

$$\min_{\alpha} \frac{1}{n} \|K_{nn}\alpha - y\|^2 + \lambda \alpha^\top K_{nn} \alpha,$$

where K_{nn} is the kernel matrix with entries $[K_{nn}]_{ij} = k(x_i, x_j)$.

Setting the derivative with respect to α to zero leads to the linear system

$$(K_{nn} + \lambda n I_n)\alpha = y, \tag{2}$$

whose solution fully determines the learned function as the initial optimisation problem is convex.

Solving this system directly requires storing the full kernel matrix and performing matrix inversion or Cholesky factorization. This leads to a computational complexity of $O(n^3)$ in time and $O(n^2)$ in memory, which becomes intractable as soon as n reaches the order of 10^5 samples. Even when exploiting modern GPUs, memory limitations and data transfers quickly become bottlenecks.

We confirmed these scaling limitations empirically through timing benchmarks of the vanilla kernel solver (Fig 2). In the region before 5×10^2 points, Python overhead dominates, so the scaling is noisy and inconclusive. However, after 10^3 points, we can clearly see a scaling trend around n^3. Pushing the benchmarking in the region of 10^5 points is impossible as saving the K matrix for 10^5 would require approximately 80GB in float32, demonstrating the failure of kernels method to handle very large datasets. This gap between statistical optimality and computational feasibility motivates the use of approximate kernel methods, designed to reduce memory requirements and computational complexity while preserving learning guarantees. Among these methods, the *Nyström* approximation plays a central role and forms the basis of the scalable solver studied in this work.

2.3 Nyström Approximation and Reduced Kernel Solvers

To alleviate the computational bottleneck of full kernel solvers, approximate methods based on subsampling have been developed. Among them, the Nyström approximation is one of the most widely used and theoretically well-founded approaches.

The central idea is to restrict the kernel expansion to a subset of $m \ll n$ inducing points $\{\tilde{x}_1, \ldots, \tilde{x}_m\} \subset \{x_1, \ldots, x_n\}$ sampled from the training set. The predictor is then approximated by

$$f(x) = \sum_{j=1}^{m} \alpha_j k(x, \tilde{x}_j),$$

leading to a reduced-dimensional parameter vector $\alpha \in \mathbb{R}^m$ instead of $\mathbb{R}^n$.

Denoting by $K_{nm} \in \mathbb{R}^{n \times m}$ the cross-kernel matrix between training samples and inducing points and by $K_{mm} \in \mathbb{R}^{m \times m}$ the kernel matrix of the inducing points, the kernel ridge regression problem becomes

$$(K_{nm}^\top K_{nm} + \lambda n K_{mm})\alpha = K_{nm}^\top y. \tag{3}$$

An example of KRR using Equation 3 is given in Figure 3.

This reformulation reduces computational requirements. Directly solving the reduced system costs $O(nm^2 + m^3)$ in time and $O(m^2)$ in memory (by computing K_{nm} in blocks). Crucially, recent theoretical results [4] show that optimal statistical learning rates can still be achieved with as few as

$$m = O(\sqrt{n})$$

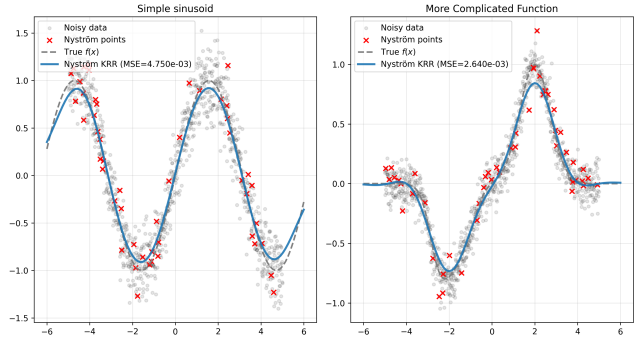

Figure 3: KRR with Nyström Approximation using 50 points out of 1000 total

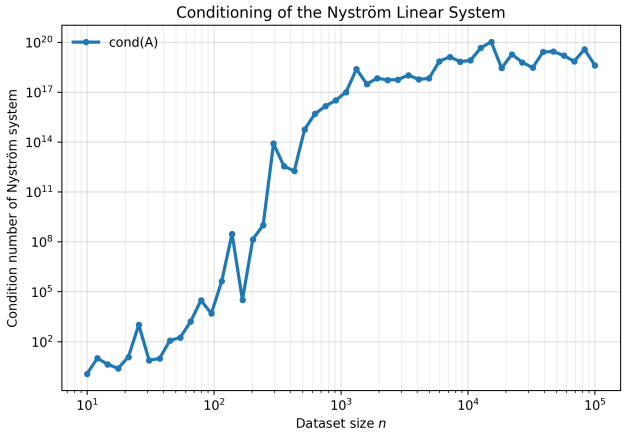

Figure 4: Evolution of condition number of $K_{nm}^\top K_{nm} + \lambda n K_{mm}$

inducing points, preserving the $O(n^{-1/2})$ excess risk bound while drastically lowering computational complexity.

Nevertheless, for large-scale settings even this reduced problem remains challenging, as the linear system size and the cost of matrix operations may still be significant for m in the thousands or tens of thousands. This motivates the use of iterative solvers combined with effective preconditioning schemes to further improve scalability, leading to the FALKON algorithm described in the next section.

2.4 Iterative Solvers and conditioning

One could try to solve Eq. (3) by inverting the matrix $K_{nm}^\top K_{nm} + \lambda n K_{mm}$. However since m can be quite large, storing the matrix and it is inverse in memory will be impossible. In addition, directly inverting this matrix can be unstable. This naturally leads us to consider iterative methods and to condition the matrix so that convergence will be quicker.

To that end, a natural choice is to employ the conjugate gradient (CG) algorithm. However, the convergence rate of CG depends critically on the condition number of the matrix to be solved. In kernel problems, the matrices involved are often poorly conditioned (see Fig 4), which results in slow convergence and defeats the computational advantages offered by iterative solvers. This issue motivates the use of preconditioning.

By setting $H = K_{nm}^\top K_{nm} + \lambda n K_{mm}$ and $b = K_{nm}^T y$, Eq 3 becomes

$$H\alpha = b \tag{4}$$

Setting $\alpha = P\beta$ and multiplying by P^T we get

$$P^T H P \beta = P^T b \tag{5}$$

In the ideal case we would have $P^T H P = I$ and then the system would converge in one iteration since $\beta = P^T b$. In that case

$$H = P^{-T} P^{-1}$$

Hence

$$H^{-1} = P P^T$$

But finding $PP^T = H^{-1}$ is as expensive as solving the system directly. We approximate P by $\tilde{P}$ such that

$$\tilde{P}\tilde{P}^T = (\frac{m}{m} K_{nm}^2 + \lambda n K_{mm})^{-1} \tag{6}$$

using the approximation $K_{nm}^\top K_{nm} \approx \frac{n}{m} K_{mm}^2$ This follows from a Monte-Carlo Nyström argument: writing $K_{nm}^\top K_{nm} = \sum_{i=1}^n r_i^\top r_i$, with r_i the rows of K_{nm}, and noting that $K_{mm}^2 = \sum_{j=1}^m r_{s_j}^\top r_{s_j}$ is the same sum restricted to the Nyström subset $\{s_j\}_{j=1}^m$, uniformly sampled from $\{1, \dots, n\}$. By linearity of expectation,

$$\mathbb{E}[K_{mm}^2] = \frac{m}{n} \sum_{i=1}^n r_i^\top r_i = \frac{m}{n} K_{nm}^\top K_{nm},$$

so that $\frac{n}{m} K_{mm}^2$ is an unbiased estimator of $K_{nm}^\top K_{nm}$

Thus, $\tilde{P}$ is such that $\tilde{P}^T H \tilde{P} = I$ and the system

$$\tilde{P}^T H \tilde{P} \beta = \tilde{P}^T b \tag{7}$$

is almost perfectly conditioned. The conjugate gradient converges in

$$T = O(\sqrt{\kappa(\tilde{P}^T H \tilde{P}) log\left(\frac{1}{\epsilon}\right)}$$

In our case we have $\kappa\left(\tilde{P}^T H \tilde{P}\right) \approx 1$ and $\epsilon = O\left(n^{-\frac{1}{2}}\right)$ following Eq. (1) hence (CG) converges in a number of steps T:

$$T = O(\log n)$$

2.5 The Falkon Algorithm

To recapitulate in order to solve Eq 3

$$(K_{nm}^\top K_{nm} + \lambda n K_{mm})\alpha = K_{nm}^\top y H \alpha = b$$

We introduce the condition matrix that satisfies Eq 6

$$\tilde{P}\tilde{P}^T = (\frac{n}{m} K_{nm}^2 + \lambda n K_{mm})^{-1}$$

and we solve

$$\tilde{P}^T H \tilde{P} \beta = \tilde{P}^T b$$

Using the Conjugate Gradient method. Once we have β we solve for α simply by

$$\alpha = \tilde{P}\beta \tag{8}$$

Let's now focus on the practical side of this calculation.

Using Cholesky factorization, we can find matrices T and A such that

$$K_{mm} = T^\top T \qquad \text{and} \qquad \frac{1}{m} TT^\top + \lambda I = A^\top A.$$

Hence

$$\begin{aligned}(\tilde{P}^T \tilde{P})^{-1} &= \frac{n}{m} K_{mm}^2 + \lambda n K_{mm} \\ &= nT^T \left(\frac{1}{m} TT^T + \lambda I \right) T \\ &= n \left(T^T A^T \right) (AT)\end{aligned}$$

It follows,

$$P = \frac{1}{\sqrt{n}} T^{-1} A^{-1} \tag{9}$$

Each iteration of CG requires evaluating the operator

$$\begin{aligned}v &\mapsto \tilde{P}^T H \tilde{P} v \\ &= \frac{1}{\sqrt{n}} A^{-T} T^{-T} \left(K_{nm}^\top K_{nm} + \lambda n K_{mm} \right) \frac{1}{\sqrt{n}} T^{-1} A^{-1} v \\ &= \frac{1}{n} A^{-\top} T^{-\top} K_{mn} K_{nm} T^{-1} A^{-1} v \; + \; \lambda A^{-\top} A^{-1} v.\end{aligned} \tag{10}$$

This operation can be decomposed into the following steps:

(1) **Apply** A^{-1}

$$v_1 = A^{-1} v.$$

Solving the triangular system costs

$$O(m^2).$$

(2) **Apply kernel blocks and** T^{-1}

$$v_2 = k(X_m, X)\, k(X, X_m)\, T^{-1} v_1.$$

This requires:
- solving $T^{-1} v_1$ at cost $O(m^2)$,
- one multiplication by $k(X, X_m)$ costing $O(nm)$,
- one multiplication by $k(X_m, X)$ costing $O(nm)$.

The total cost of this step is therefore

$$O(nm).$$

(3) **Apply** A^{-T} **and** T^{-T} **and add regularization**

$$v_f = \frac{1}{n} A^{-T} T^{-T} v_2 + \lambda A^{-T} v_1.$$

These are triangular solves whose cost is

$$O(m^2).$$

Combining all operations, the cost of one CG iteration is

$$O(nm + m^2).$$

With the Nyström choice $m = O(\sqrt{n})$, we obtain

$$\text{Cost per CG iteration} = O(n\sqrt{n}).$$

Since the preconditioned system has approximate condition number 1, CG requires $O(\log n)$ iterations to reach statistical accuracy $\varepsilon = O(n^{-1/2})$. Therefore, the total runtime of Falkon is

$$O(n\sqrt{n} \log n)$$

The memory footprint is dominated by storing the dataset and the matrices K_{mm}, A, and T, yielding

$$O(n) \text{ memory}$$

In order to get the final solution we simply use Eq 8

$$\alpha = \frac{1}{\sqrt{(n)}} T^{-1} A^{-1} \beta$$

and solve this system which has a cost of $O(m^2)$

It is important to note that in this review we present the corrected version of the Algorithm 1 in [3], accounting for normalisation and proper scaling. In this section we only dealt with the KRR problem which admits analytical solutions, as in the original paper. It is also possible to extend these results to other loss functions.

2.6 Reproduction of Theoretical Results and Algorithms

We implemented both the Nyström approximation and the FALKON algorithm in order to empirically validate the theoretical scaling laws, in the same spirit as the benchmark performed for vanilla KRR in Fig. 2. In the rest of this paper, we will cal our implementation of Falkon *FalkonCPU* and reserve the Falkon name (sometimes also noted FalkonGPU) for the GPU accelerated version.

Nyström method. The reproduction of the simple Nyström solver was straightforward. Our implementation follows directly the reduced system in Eq. 3 by explicitly forming the kernel blocks K_{mm} and K_{mn} and solving the resulting linear system. We used the numerical solver provided by numpy and did not encounter any numerical instability issues.

FalkonCPU algorithm. Reproducing Falkon was substantially more challenging, as it required reimplementing a fully matrix–free preconditioned Conjugate Gradient (CG) solver tailored to the operator

$$v \mapsto \tilde{P}^\top H \tilde{P} v.$$

Practical difficulties arose from two main sources. First, the Cholesky factorization of K_{mm} can fail due to finite-precision effects, since K_{mm} is only guaranteed to be positive semidefinite. This issue was resolved by adding a small diagonal stabilization ("jitter") term prior to factorization. Second, insufficient convergence of the CG procedure or an overly small Nyström rank m led to oscillatory artifacts in the resulting regressors. Adopting the theoretical prescription $m = O(\sqrt{n})$ and tightening the CG stopping criterion restored stable solutions that were consistent with the full KRR estimator.

The full implementation used for our benchmarks is provided in `/scr/kmtr/kernel_solvers`.

Benchmarking and empirical validation. The runtime benchmarks obtained from our implementations are reported in Fig. 5, together with empirical scaling exponents fitted for dataset sizes $n \geq 10^3$, where timing noise and Python overhead become negligible. In this regime, the measured slopes are

$$\alpha_{\text{vanilla}} \approx 2.93, \qquad \alpha_{\text{Nyström}} \approx 1.43, \qquad \alpha_{\text{FALKON}} \approx 1.42.$$

These results are consistent with the theoretical complexity predictions. The vanilla KRR solver exhibits its expected cubic behavior.

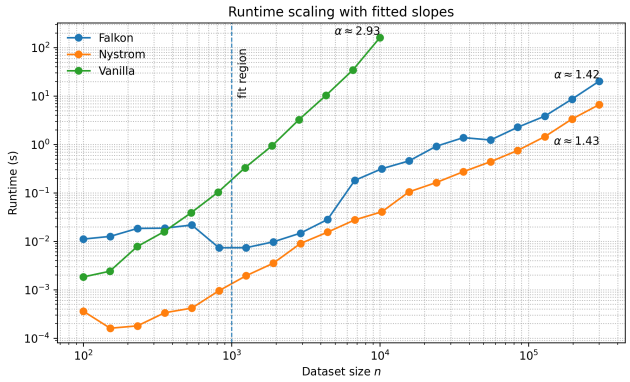

Figure 5: Runtime scaling of vanilla KRR, Nyström and FALKON solvers on CPU (log–log scale). Slopes are fitted for $n \geq 10^3$:

Classical Nyström (solving Eq. 3) has theoretical complexity (see 2.3)

$$O(nm^2 + m^3),$$

which with the standard choice $m = O(\sqrt{n})$ yields an overall quadratic rate $O(n^2)$. However, for the dataset sizes accessible in our experiments, the dominant cost is the kernel block construction $O(nm)$, leading to an apparent $O(n^{3/2})$ scaling in the measured runtimes.

The Falkon algorithm has a total complexity of (see 2.5)

$$O(nm \log n) = O(n^{3/2} \log n),$$

and the slowly varying logarithmic factor is not observable on a log–log fit over the limited experimental range, resulting in an effective slope close to 3/2.

Although FalkonCPU exhibits the expected asymptotic behavior, it is not systematically faster than the simple Nyström solver in absolute runtime. This discrepancy is explained by constant-factor effects: our Python implementation relies on an explicit matrix–free Conjugate Gradient loop with repeated triangular solves and kernel matrix–vector products, whereas the Nyström solver reduces to a single dense system solve handled by highly optimized numpy solvers.

3 METHOD

Motivation of their approach One key observation motivating their work is that the classical kernel methods have a very low operation density, meaning that each floating point operation requires loading a relatively large amount of data. In contrast, modern GPU are designed for the opposite : they deliver peak performance when many arithmetic operations can be performed on data that is already *in-place* (in fast memory), while data transfer - from for example the host RAM - is pretty slow. To better understand why kernel methods suffer from low arithmetic intensity, consider the computation of each coefficient of K_{nm} : in order to apply the operation $k(x_i, x_j)$, one must load one pair (x_i, x_j), so a memory cost of $2ds$ (note s the number of bytes per value, for float64, $s = 8$). For an RBF kernel that cost $3d$ in computation time since $|x - y|^2$ implies d differenciations , d multiplications, and about d additions. Therefore this implies for instance with $d = 1000$ stored in float64 so $\approx$ 4 KB (required for enought precision as we will see it later) :

$$\frac{3d}{2ds} = \frac{3000 \text{ FLOP}}{16000 \text{ bytes}} \approx 0.188 \text{ FLOP/byte}.$$

Such arithmetic density is very low compared to the regime where most GPU operate. Consider the classical matrix product (which GPU are known to be efficient for) : load once two matrix A and B, , and account for storing the result, the memory cost is $3N^2 s$, N being the size of the matrix, and s the number of bytes. The computation implies $2N^3$ floating point operations. In the case of $N = 1000$, this leads to

$$\frac{2N^3}{3N^2 s} = \frac{1}{12}N \approx 166.7 \text{ FLOP/byte}$$

This value is three orders of magnitude higher in terms of arithmetic density. The objective of the authors is therefore to restructure the kernel solver so that its computational pattern more closely resembles high-density GPU workloads. Although the RBF kernel evaluations themselves remain intrinsically low-density operations, the solver can still be redesigned to maximize the ratio between computations and data transfers—that is, the effective compute density. If implemented naively, the solver can imply large data movement for very little computation (as formalized in next subsection). By carefully choosing block sizes and processing the data in chunks, the authors ensure that operations respect GPU memory constraints while maintaining a favorable balance between memory footprint and transfer overhead.

This tradeoff is inherent: to minimize transfer time one would ideally store all inputs on the GPU, but memory constraints make this impossible for realistic n and m. Even storing everything in CPU RAM is nontrivial at scale, so the entire computational pipeline must be rethought.

3.1 RAM memory bottleneck

A major contribution of the authors is the reduction of RAM usage, which can be a dominant bottleneck in large-scale kernel methods.

Naive implementation and memory cost If we were to ignore memory constraints, an unoptimized implementation of the solver without any optimization would explicitly store and use each matrix that appears in the Falkon Algorithm, including the full kernel matrix K_{nm}. This is, of course, infeasible at realistic scales. For example, for typical large-scale settings targeted by Falkon (e.g., $n = 10^9$, $m = 10^5$), forming K_{nm} represents

$$nm = 10^{14} \text{ entries} \approx 800 \text{ TB}$$

in single precision, and would not fit in RAM memory for most of pratical situations ! This already motivates blockwise computations. Further, more subtle optimizations allow the total memory footprint to be reduced essentially to a single $m \times m$ matrix, with no additional multiplicative factors.

No need to store K_{nm} : compute it batch-wise. In the conjugate gradient iterations, K_{nm} only appear in vector matrix multiplication, and can thus be computed on the fly without explicit storage. A fundamental point of the method is to batch the computation of the vector-matrix, and compute $K_{qm} \in \mathbb{R}^{q \times m}$ with $q << n$. More

clearly, as denoting $M\beta = b$ the system to solve by conjugate gradient, the matrix M will never be stored fully, as it would be too big, only $M\beta$ will be, and computed by batches, using the accumulation formula, illustrated here :

$$K_{nm}^T K_{nm}(T^{-1}A^{-1}\beta) = \sum_{b=1}^{B} k(X_{b,:}, X_m)^\top \left(k(X_{b,:}, X_m)\, T^{-1}A^{-1}\beta\right)$$

We investigate this trade-off in Figure 6, which illustrates how memory usage and computation time vary with the batch size q. Experiments conducted on an NVIDIA T4 GPU (Google Colab) confirm the expected behavior: while smaller values of q reduce GPU memory consumption, they also lead to substantially longer computation times. This slowdown occurs because smaller batches require more frequent memory transfers, which become a significant bottleneck. This effect is clearly visible from the inversion in the ordering of the curves between the two plots.

Only keep A and T in memory If computing K_{nm} can be avoided, the matrix K_{mm} must be materialized in memory to construct the preconditioner. When m is large, the memory footprint of K_{mm} becomes a critical bottleneck, especially if we need to store it alongside A and T the matrix from the Cholesky Decomposition. A key observation made by the authors is that K_{mm} does not appear in the preconditioned linear system solved by conjugate gradient (as we saw just before in 10). So in order to solve the conjugate gradient descent, we only need in memory A and T, as the transpose and inverse operations can be done in-place (directly when computing the above equation) without any other necessary storage.

Strategy for preconditionner : store all the necessary matrix values in one place of size $m \times m$ The previous algebraic cancellation allows the solver to overwrite the memory area initially containing K_{mm} with intermediate matrices T, $TT^\top$, and finally A, without ever needing multiple copies. To address this issue, the authors design an in-place strategy that reuses a single $m \times m$ buffer throughout all stages of preconditioner construction. Concretely, the preconditioner $\tilde{P}$ of Equation 6 in the original paper is defined through the factorization

$$\tilde{P} = \frac{1}{\sqrt{n}} T^{-1}A^{-1},$$

where

$$T = \mathrm{chol}(K_{mm}), \qquad A = \mathrm{chol}\left(\tfrac{1}{m}TT^\top + \lambda I_m\right).$$

Starting from K_{mm} stored in the buffer of size $m \times m$, we can compute $T = \mathrm{chol}(K_{mm})$ by overwriting the upper triangular part of K_{mm} (as T is triangular). From T, $\frac{1}{m}TT^\top + \lambda I_m$ can be computed and stored by overwritting the bottom triangular part of K_{mm} that was left in the buffer. And from here we can finally compute A and overwrite $\frac{1}{m}TT^\top + \lambda I_m$ by A. The diagonal must be handled carefully—for example, by keeping a buffer of the diagonal of T when overwritting it with the one of $\frac{1}{m}TT^\top + \lambda I_m$ and then the one of A.

In addition to reducing the RAM footprint to a single $m \times m$ matrix, the implementation maintains only:

- the coefficient vector $\beta \in \mathbb{R}^m$,
- a buffer for the inducing points,
- and a data batch of size q used to compute batches of K_{nm} for matrix–vector products involving .

In this design, approximately $\frac{m^2}{m^2+m+qd+md} \approx 90\%$ of the total memory is devoted to the preconditioner, and all matrix–vector products required by conjugate gradient are computed on the GPU without storing K_{nm} explicitly.

One possible criticism arises here: although the authors emphasize that their design minimizes RAM usage as much as possible, the method still requires approximately 150 GB of memory when $m = 2 \times 10^5$. Such requirements make the approach impractical for most users and effectively force the choice of m to be smaller than the theoretically recommended $m \sim \sqrt{n}$. Choosing m significantly below this regime has well-known implications. In the Nyström framework, a too small number of inducing points deteriorates the approximation quality of both K_{nm} and the preconditioner P, which in turn worsens the conditioning of the linear system. As a consequence, the number of conjugate-gradient iterations increases, the preconditioner becomes less effective, and both the statistical performance and the computational efficiency may degrade.

This limitation may not be emphasized strongly enough in the paper, especially considering that we encountered this issue directly in our experiments: for large n, we were constrained to use very small values of m. In that sense, the theoretical recommendation becomes difficult to attain in realistic settings, and the trade-off between memory feasibility and approximation quality might deserves a more explicit discussion.

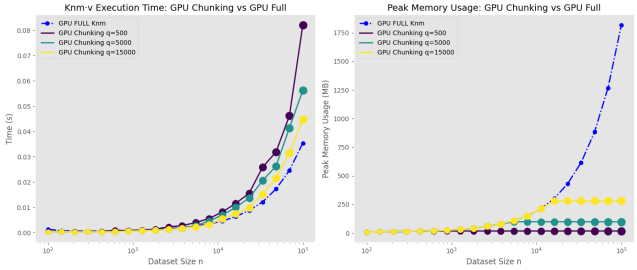

Figure 6: Comparison of the memory and time consumption for chunking vs non chunking the product $K_{nm}v$ with v a random vector

3.2 GPU memory and Multi-GPU support

If RAM memory is certainly very constrained in most compute system, the GPU memory is even more.

Motivation The preconditionner storage will typically require 150 GB of memory, so will not fit on a GPU. We need to deal with it stored on the RAM. Also we will need an efficient computation of the K_{nm} block wise product, that will need to fit on the GPU. For that the authors use Out-Of-Core (OOC) operations : the matrix is stored onto a relatively slower storage than the VRAM of the GPU (for example here the RAM, which has the advantage to be generally bigger in storage), but the operations are then processed onto the GPU. This process is not implemented in classical libraries as PyTorch, as a lot of its main mecanisms as AutoGrad (for backpropagation) or its Kernel assume the data all live in the same place, and let the user handle the transfer between storages if necessary. The authors do provide this implementation in their library.

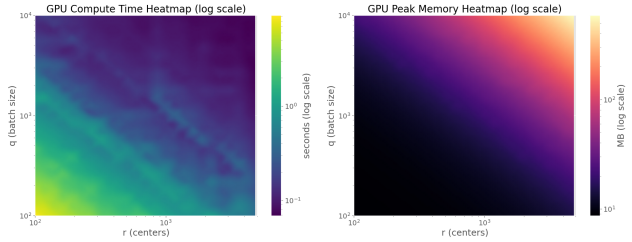

Figure 7: Computationnal time and memory cost on a T4 GPU, according to different range of q and r - parameters set to $n = 50000$, $m = 5000$ and $s = d = 20$

- **Optimized OOC. operations** To minimize the footprint of data-transfers, only the specific RAM-resident block parts required for the Conjuguate Gradient Descent are transferred onto the GPU (namely, the relevant blocks of A, T and X). The key detail is that the corresponding block of the kernel matrix is never transferred, but computed direclty onto the device. And the resulting partial output (a part of updated β) is transfered back to the host.

Therefore a key practical aspect is selecting the optimal chunk size, that is, choosing q, r, s the block dimensions along n, m and d (number of data points, Nyström center, and feature dimension). But we need to keep in mind that there exist a tradeoff between

- the computationnal complexity qrs (the computation of the batch of K_{nm})
- and the transfert time $qs + ds$ (the required time to transfer the batch of sample $X_{(i:i+q,j:j+s)}$ and ds

To maximize GPU utilization under the available memory budget, authors propose to find the optimal value of the tradeoff, i.e.

$$argmax(\frac{qrs}{qs + sd}), \text{with } qs + sd \leq G$$

G being the available GPU memory.
We explicitly write the equation of the split of the data : $X^{(b)}$ is batch of q rows, and the centers into blocks of r columns, $X_m^{(j)}$, we have :

$$K^{(b,j)} = k\left(X^{(b)}, X_m^{(j)}\right) \in \mathbb{R}^{q \times r}.$$

Writing $v = T^{-1}A^{-1}\beta = [v^{(1)}, \ldots, v^{(J)}]^\top$, the matrix–vector product $u = K_{nm}v$ is computed batch-wise as

$$u^{(b)} = \sum_{j=1}^{J} K^{(b,j)}\, v^{(j)}, \qquad b = 1, \ldots, B.$$

The second stage, needed for evaluating $K_{nm}^\top u = K_{nm}^\top K_{nm} v$ inside conjugate gradient, reuses the same tiles and accumulates over row-batches:

$$w^{(j)} = \sum_{b=1}^{B} \left(K^{(b,j)}\right)^\top u^{(b)}, \qquad j = 1, \ldots, J.$$

Furthermore for the split of size s along d, we can simply accumulate partial distance contributions over these blocks.

In Figure 7, we propose to a toy experiment to check over a true GPU the memory cost and the computationnal time for different values of q and r, without batching over the dimension d and with n and m fixed for simplicity. The heatmap is performed over a grid of size 25×25 and then interpolated with a factor 4 and cubic interpolation for smoother plots. The memory usage is as expected increased when q and r both increase. And in the opposite, the computationnal time does decrease with larger values of q and r.

OOC multi GPU Cholesky decomposition The authors implement an OOC Cholesky decomposition to factorize in a memory efficient way large matrix such as K_{mm}, while leveraging multi-GPU, where different GPUs can process different row of the input matrix. We need the cholesky decomposition of K_{mm} to compute T and then A.

Traditionally, a naïve Cholesky decomposition requires loading the entire $m \times m$ matrix into GPU memory, performing all operations in-core, and storing both intermediate and final triangular factors. This approach has a memory cost of approximately $O(m^2)$ which becomes prohibitive for typical used values of m.
The goal is to reduce the memory footprint of each batch operation on GPU by reducing the operation to a Cholesky decomposition over a sub-matrix of size $t \times t$. We note A for K_{mm} for simplicity here. A and L are divided into blocks noted $(A_{ij})_{1 \leq i,j \leq B}$ and $(L_{ij})_{1 \leq i,j \leq B}$. The matrix L is computed iteratively per column. The first iteration gives a good intuition on how it is performed :

- Compute $L_{11} = \text{Chol}(A_{11})$ (fits on GPU when t is correclty set
- Using the triangular structure of L, we have $A_{i1} = L_{i1}L_{11}^T$, so having L_{11}, we can compute $L_{i1}, \forall i$, as $L_{i1} = A_{i1}(L_{11}^T)^{-1}$, which can be done using TRSM.
- When going to next step, for L_{i2}, we will get $A_{i2} = L_{i1}L_{12}^T + L_{i2}L_{22}^T$, and so to avoid doubling the memory usage comparing to step 1, the idea is to sustract $L_{i1}L_{1j}^T$ doubling, for all i, j to each A_{ij}, and doing this at each iteration ensures that the temporary GPU memory usage remains bounded by approximately $2 \times t^2$.

The full algorithm iteratively does the above steps for all columns. The multi GPU advantage is exploited by distributing different block-rows of the matrix across GPUs, allowing them to process independent parts of the Cholesky decomposition in parallel.

3.3 Optimization of data transfers

GPU can mostly do 3 types of operations : (i) loading data from RAM, (ii) performing computations on device, and (iii) saving on RAM. These 3 processes can run in parallel, so to coherently exploit this concurrency, authors do schedule GPU tasks so that at step k,

- the GPU computes the current block operation
- simultaneously loads the block required for iteration $k + 1$
- saves the result from iteration $k-1$ back to the RAM memory

Assuming each process takes the same time t, this reduces the time spent by nearly a factor of three from $3t \times k$ to $t \times (k+2)$ (where the two additional units correspond to the initial data load and the final write). The hypothesis that each process takes the same time could be largely challenged as the transfert as the loading and saving part might take the most time, and overlapping the process in pratice could lead to smaller compute time gain.

3.4 Other improvements : numerical precision, dealing with thin submatrices and sparse datasets

Floating point precision GPU are known to be very efficient with low precision floating point numbers (WHY). For this, it is generally preferable to do the computation using 32-bit float number instead of 64-bit. However, the computation of the gaussian kernel does involve norm of the form :

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2\,x^\top y$$

where the three terms involve product of $|x|$ and $|y|$ that can be quiet large (specially in high dimension) and that might cancel if x and y are close. In that case there might not be enought significant digit in 32-bit, leading to negative value in kernel blocks. The figure 7 illustrates well with a toy example this phenomenon. Therefore, when computing K_{mm} per block, the authors do use mix precision : inputs are temporarily cast to float64 for the accumulation of the squared distance, and the resulting kernel value is cast back to float32 before storage.

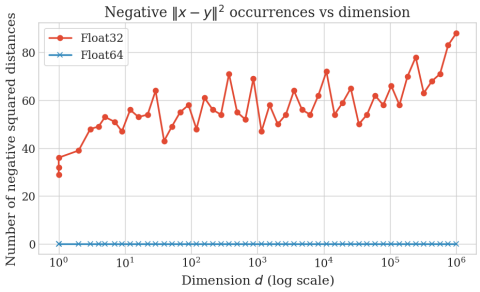

Figure 8: Toy experiment : Number of negative values of $||x - y||^2$ with 200 uniform draw x of $\mathcal{N}(0, 1)$ of size d for each point, and $y = x + \epsilon$ with epsilon drawn from $10^{-4}\mathcal{N}(0, 1)$

The authors also underly a thin matrixes issue : When matrices as K_{mm} are chunked into blocks, the final block along one dimension may contain only a small number of rows—resulting in a thin matrix. This breaks the computational symmetry across GPUs, and causes a lot of increase of cost. To mitigate this, authors rely on KeOps, which is a library that can deal efficiently with thin kernel matrixes.

Sparse matrixes that appears in some datasets are also dealt with wrapped code methods such as cuSPARSE, allowing the solver to handle sparse inputs without modification of the core algorithm.

4 EXPERIMENTATION

4.1 Practical details

Since FALKON comes as a Python library we decided to reproduce some of their experiments and compare against other algorithms, as well as our own implementations.

Datasets. Experiments are conducted on the following public benchmarks:

- **Year Prediction MSD (MSD)[1]**: Prediction of the release year of a song from audio features. A regression dataset containing approximately $n \approx 5 \times 10^5$ samples with $d = 90$ continuous features.
- **Higgs[?]**: a large-scale binary classification dataset with more than $n \approx 10^7$ samples and $d = 28$ features.
- **Mini Higgs** : A sub-sampled version of Higgs with $n \approx 10^5$ points

Each dataset is randomly split into training and test sets using fixed random seeds to ensure reproducibility. We adopt the same parameters as in the paper and choose 10% of MDS and 20% of Higgs as the test data.

For all experiments, input features are standardized independently per dataset using statistics computed from the training set. Target values are left unscaled. All models operate on the same normalized inputs to ensure fair comparison.

Models.

- **Falkon GPU**: official GPU implementation from the `falkon` library.
- **Nyström**: standard kernel ridge regression with Nyström approximation.
- **Falkon CPU (ours)**: our own implementation of the Falkon algorithm using conjugate gradient optimization.
- **GPyTorch**[2] : We used the SVGP model with Gaussian and Bernoulli likelihoods.
- **Linear Regression** : We use a simple Linear regression model as baseline.

All kernel models use a Gaussian RBF kernel $k(x, x') = \exp\left(-\frac{\|x-x'\|^2}{2\sigma^2}\right)$, with common hyperparameters across methods whenever applicable.

Compute. All experiments are performed on a DGX A100 machine with 40GB of VRAM. This is significantly less compute than [3], so we were not able to achieve the same performances. We can however reproduce the approximate scaling laws and approximate the final performances. For each dataset and model the best run was selected at the end

4.2 Results

Sanity Check. We first compared three approaches on the mini-HIGGS dataset: the reference implementation from the FALKON library, a straightforward Nyström solver relying on NumPy linear algebra, and our own FalkonCPU implementation. As expected, the NumPy-based solver produced prediction accuracies comparable to the official FALKON implementation, but exhibited significantly longer runtimes due to the absence of dedicated preconditioning and iterative solvers. Our custom CPU solver was both slower and less accurate than the two baselines, which can be attributed to inefficient memory handling, lack of optimized linear-algebra routines, and potential numerical inaccuracies in the implementation. Overall, this experiment confirms that our solver should be regarded as a proof-of-concept rather than a competitive baseline.

Scaling Laws. We attempted to reproduce the scaling curves reported in Fig. 1 of the original FALKON paper, which relate predictive performance to training time for increasing numbers of Nyström centers m. In practice, several implementation and hardware limitations hindered a faithful reproduction: frequent memory errors,

Method	1 - AUC	Runtime (s)
Nyström (NumPy)	0.25	25*s*
FALKON (official)	0.25	6
FALKON (ours, CPU)	0.28	60

Table 1: Sanity-check comparison on the `mini-HIGGS` dataset. 1−AUC and total runtime are reported for each method.

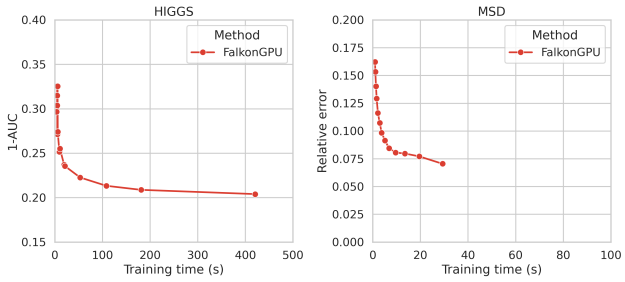

Figure 9: Scaling laws for FALKON on the HIGGS and MSD datasets

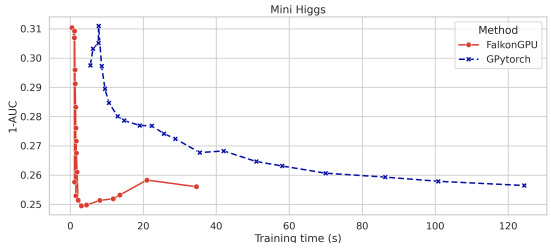

Figure 10: Comparison between FALKON and GPyTorch on the Mini-HIGGS dataset

CUDA out-of-memory failures, unstable runtimes, and difficulties with parallelization on our available infrastructure.

Despite these constraints, we were able to reproduce representative performance trends for FALKON on two datasets, HIGGS and MSD (Fig. 9). As expected, increasing training time (through larger m) leads to a steady reduction of the error metric.

To enable a direct comparison with another kernel-based method, we additionally replicated an experiment on Mini-HIGGS experiment (we faced issues running on the full HIGGS Dataset), benchmarking FALKON against GPyTorch (Fig. 10). FALKON achieves significantly better accuracy for comparable or even smaller training times, highlighting its practical efficiency for large-scale kernel learning.

Overall Performance. We report the results of our best-performing models and compare them with those obtained in the original FALKON paper. On HIGGS, our measurements are consistent with the expected performance trends reported in the literature, with FALKON substantially outperforming baseline methods for a given training budget.

Table 2: Performance and runtime comparison on large-scale datasets. Results report test error (relative error for MDS, 1−AUC for HIGGS) and total training time. "FALKON (paper)" corresponds to the reference values reported in the original publication.

	MDS ($n \approx 5 \times 10^5$)		HIGGS ($n \approx 10^7$)	
Solver	**Rel. Error**	**Time (s)**	**1−AUC**	**Time (s)**
FALKON	0.00545	83.45	0.223	623
Linear Regression	0.003	1.80	0.32	0.031
FALKON [3] (paper)	0.0048	62	0.180	443

In contrast, our experiments reveal that the MDS dataset is a weak benchmark for evaluating kernel methods. Reported scores can be matched or even exceeded by simple linear regression. Direct inspection of predictions shows very low correlation between input features and targets: near-constant predictors achieve comparable performance. This behavior is largely explained by the evaluation metric, which compresses errors due to the narrow target range ($\approx [1920, 2010]$), thereby masking differences between expressive models and trivial baselines. Finally, MDS is absent from the current official Falkon repository, further questioning its practical relevance as a benchmarking dataset.

5 CONCLUSION

This study reviewed and experimentally examined the FALKON framework for large-scale kernel learning. By combining Nyström approximations with preconditioned conjugate gradient solvers, FALKON reduces the effective complexity of kernel ridge regression to $O(n^{3/2} \log n)$, a result supported by our empirical scaling analyses. Our experiments confirmed the substantial performance gap between vanilla solvers, standard Nyström methods, and FALKON, while also illustrating the importance of GPU-oriented system optimizations for achieving practical speedups.

Despite these advances, our results highlight a key remaining limitation: even with FALKON 's optimizations, large-scale kernel methods still require significant memory resources to be practical, particularly for storing intermediate kernel blocks and preconditioners. This constraint limits their accessibility compared to more lightweight deep learning or linear approaches.

Overall, FALKON demonstrates that kernel methods can be pushed far beyond their traditional scalability limits, but memory requirements remain a central challenge for broader adoption.

REFERENCES

[1] T. Bertin-Mahieux. 2011. Year Prediction MSD. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50K61.
[2] Jacob R Gardner, Geoff Pleiss, David Bindel, Kilian Q Weinberger, and Andrew Gordon Wilson. 2018. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. In *Advances in Neural Information Processing Systems*.
[3] Giacomo Meanti, Luigi Carratino, Lorenzo Rosasco, and Alessandro Rudi. 2020. Kernel methods through the roof: handling billions of points efficiently. In *Advances in Neural Information Processing Systems 32*.
[4] Alessandro Rudi, Raffaello Camoriano, and Lorenzo Rosasco. 2016. Less is More: Nyström Computational Regularization. arXiv:1507.04717 [stat.ML] https://arxiv.org/abs/1507.04717
[5] Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, USA.