Kernel PCA for NVIDIA RAPIDS cuML

Garrison Hess: glhess@andrew.cmu.edu
Tomas Johannesson: tjohanne@andrew.cmu.edu

1. Summary

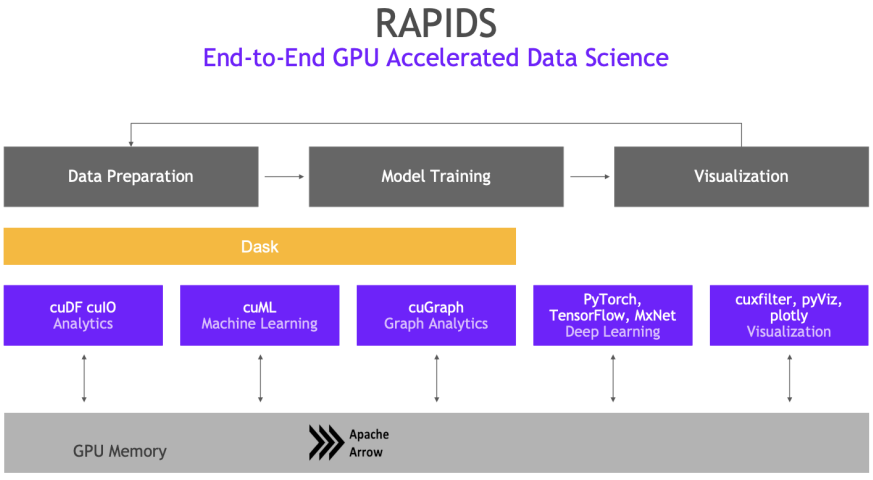
We implemented Kernel Principal Components Analysis for the NVIDIA RAPIDS cuML library. We began our work by implementing traditional PCA using CUDA, cuBLAS, and cuSOLVER. Our PCA achieves comparable performance to cuML's PCA. Our Kernel PCA significantly outperforms scikit-learn's implementation and is currently a pull request on the cuML GitHub repository.

2. Background

2.1. NVIDIA RAPIDS and cuML

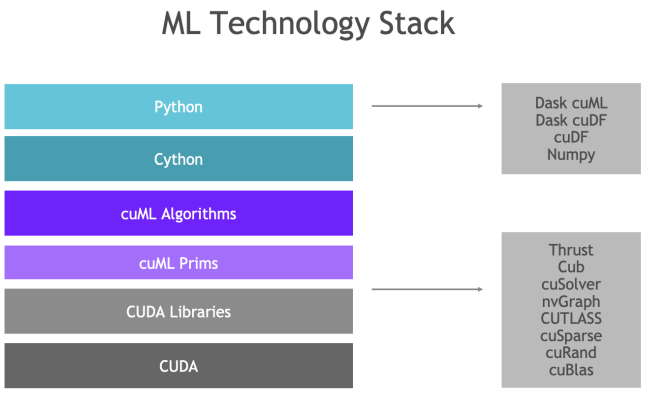
Our work focuses on accelerating two canonical machine learning algorithms through the use of Graphics Processing Units (GPUs). Nvidia's RAPIDS is the largest project working to utilize GPUs for general purpose workloads, including machine learning. We focus specifically on the RAPIDS cuML project, which is an open source software library that gives developers the option of running traditional ML tasks, without having to write cuda code. RAPIDS is built directly on top of CUDA, but provides a convenient Python API for developers.

Figure 1: RAPIDS Project Layout



RAPIDS cuML builds on top of CUDA and several CUDA libraries, supporting many machine learning algorithms. As the below picture depicts, the foundation is CUDA along with several libraries we directly use in this project (cuBLAS, cuSOLVER, and Thrust).

Figure 2: NVIDIA ML Technology Stack



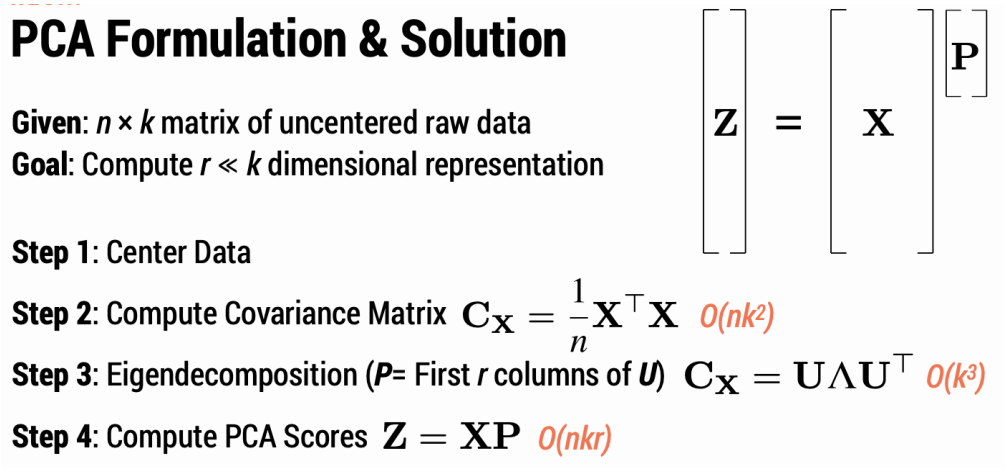
2.2. Scikit-learn

Scikit-learn is the traditional library used to develop machine learning algorithms using CPUs. It provides an excellent benchmark for cuML and our work alike. Further, the Scikit-learn API is directly matched by cuML to enable seamless transition for developers.

2.3. Principal Components Analysis (PCA)

Principal Components Analysis is a dimensionality reduction technique used to decompose an input matrix into orthogonal components. PCA is typically used to extract principal components from data for either visualization or use in downstream ML tasks. The key benefit of PCA is that it preserves the maximum amount of variance in the original data, with respect to the amount of data one chooses to keep. Therefore, PCA enables us to compress a large, high-dimensional matrix into a smaller one that is more amenable to downstream tasks. In addition, it's workload entails matrix multiplication and eigenvalue solving, which benefit significantly from parallelization. The following figure visualizes the traditional PCA workflow.

Figure 3: PCA Formulation and Solution



2.4. Kernel Principal Components Analysis (KPCA)

Kernel PCA is a modified version of PCA, which entails mapping the original data matrix through a kernel function. These kernel functions allow higher dimensional representations of the data, without requiring the direct representation of the data in the higher dimensional space. This aspect of kernel functions is generally referred to as the "Kernel Trick." Using the Kernel Trick, we can construct a potentially infinite-dimensional representation of the data, while simply storing it in a symmetric NxN matrix. These kernel functions are applied as dot products between observations in the original NxM input data matrix. The following figure shows the general form of a kernel matrix. This is also known as a Gram Matrix, as it takes the form XX^T where X is the initial data matrix.

Figure 4: Kernel Matrix Representation

$$\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} = \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \end{pmatrix} \in \mathbb{R}^{n\times n}$$

2.5 Hardware and Software

Development and experiments were conducted on bare-metal with the following workstation:

CPU - Ryzen 5950X, 16-cores, 2-threads, Base Clock 3.4GHz, 64MB L3 Cache (total)
GPU - RTX 3090, 24GB GDDR6X Memory, 10496 CUDA Cores, Ampere Architecture
Memory - 64GB DDR4 3600MHz
PCI-E - Gen 4.0, 16 lanes
CUDA - Version 11.2

3. Approach

3.1. PCA Implementation with cuBLAS and cuSOLVER

We began our work at the lowest layer of the NVIDIA machine learning stack, implementing traditional PCA using CUDA, cuBLAS, and cuSOLVER. This involved learning multiple new, complex APIs and extensive iterating on developing a working and performant solution. Each of CUDA, cuBLAS, and cuSOLVER plays a unique role in our implementation. First, CUDA provides us with the most general toolkit for writing efficient kernels for general purpose data processing. We use cuBLAS specifically for it's general matrix-multiplication (GEMM) functionality. In addition, we use cuSOLVER for its singular value decomposition (SVD) functionality. We use custom CUDA kernels for centering the data, and other smaller tasks.

3.1.1 cuBLAS GEMM

The cuBLAS GEMM function computes $C=\alpha op(A)op(B)+\beta C$. We set alpha to 1 and beta to 0, and set "op" to a no-op. With these parameters we get standard matrix multiplication. The "op" function exists because cuBLAS matches the Basic Linear Algebra Subprograms (BLAS) column-major layout. This is inherited from FORTRAN, and requires awareness of data layout at all times.

3.1.2 cuBLAS SVD Jacobi (GESVDJ) and SVD Approximate (GESVDA)

Our primary algorithm for SVD is the iterative Jacobi implementation. This algorithm computes the Gram Matrix of the input data (X^TX), and computes the left and right singular vectors and the singular values. Each iteration of the Jacobi algorithm requires matrix multiplications with a rotation matrix. The most important sequential dependency of the whole algorithm exists between the Jacobi "sweeps." Fifteen sweeps is the default for cuML as it has proven to work well in practice. We use this same value throughout our experiments. It is also worth noting that there is a full (non-iterative) algorithm that uses QR-factorization, but it does not scale as well to larger datasets. Finally, we incorporated the approximate SVD algorithm, which currently only offers a strided and batched implementation "gesvdaStridedBatched." It is

primarily intended for either processing many small matrices, or a tall and skinny matrix (with fewer than 32 columns due to warp sizes).

3.2. KPCA Implementation with cuML, RAFT, Thrust, and others

3.2.1. Learning the Landscape

Stepping into the cuML codebase took a significant amount of effort, learning several additional APIs and multiple adjacent libraries that support the RAPIDS project as a whole. However, our efforts in the first phase of the project provided an excellent foundation for learning cuML. We steadily read cuML and RAFT documentation and source code while building our initial PCA implementation.

3.2.2. Diving into cuML

Our first step with cuML was to build it from source with CUDA 11.2 in a conda environment. While doing so, we made sure to thoroughly read all contributor guidelines for CUDA/C++ developers. We learned structure of the codebase, the tools used, and the relevant APIs from adjacent codebases, including RAFT, Thrust, RMM. The first several days of being in cuML were almost overwhelming, but the pieces ended up coming together nicely. We were able to lean heavily on good documentation, and clean code examples. We wrote our implementation in about a week, including extensive tests in GTest.

3.2.3. Kernel PCA Pull Request

We reached out to an NVIDIA cuML engineer, Divye Gala, who met with us regarding our proposed KPCA implementation, and helped guide us to our pull request. We submitted our pull request recently, and have passed through the first phase of review and refactoring. In this review and refactoring process, we learned several newer and better ways of performing various things like memory allocations and device transfers. We spent a good amount of time simply re-writing our KPCA in a way that does not use older conventions in the same code-base. We also refactored and expanded our tests for the pull request. Per our discussions with Divye, we are about a week or two away from merging into the nightly branch “rapids-0.20”. As our KPCA implementation is correct and performant, our remaining contributions to the pull request will primarily involve the Cython and Python bindings that comprise the top layers of the NVIDIA ML stack.

4. Results

4.1 Datasets

4.1.1 Iris

The Iris dataset contains 150 samples of flower petal data, each with 4 features. At 150-by-4, this is our smallest dataset.

4.1.2 Eigenfaces

The Eigenfaces dataset contains 400 face images at 64-by-64 resolution. At 400-by-4096, this is our second smallest dataset.

4.1.3 MNIST

The MNIST dataset contains 70,000 handwriting images at 28x28. At 70,000-by-784, this is our second largest dataset.

4.1.4 11-785 Faces

We add our own custom dataset, which originally comes from CMU's 11-785 Deep Learning course. We took 10,000 jpg images at 64-by-64 and converted them into a 10,000-by-4096 data matrix.

4.2. Eigenfaces (PCA)

We present some samples of the initial Eigenfaces dataset, along with the PCA representations that we construct with our algorithm and Scikit-learn.

Figure 5: Initial Eigenfaces Samples

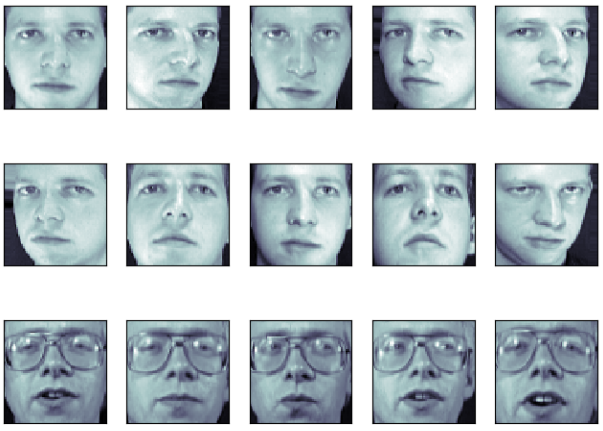


Figure 6: Our PCA - First 30 Principal Components of 400 faces

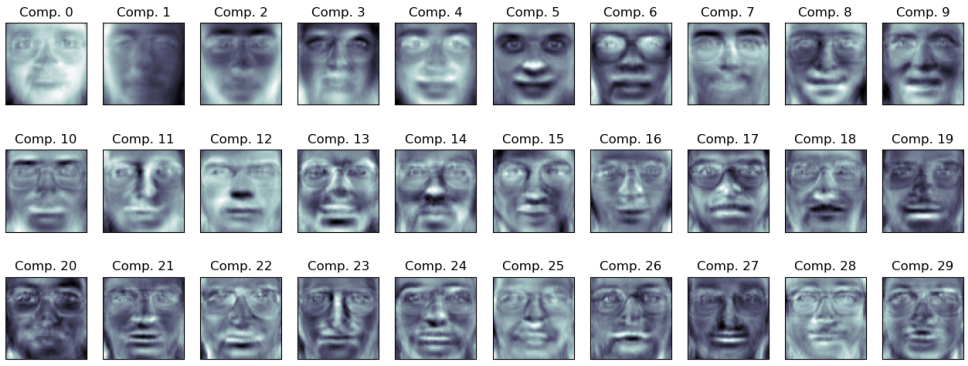
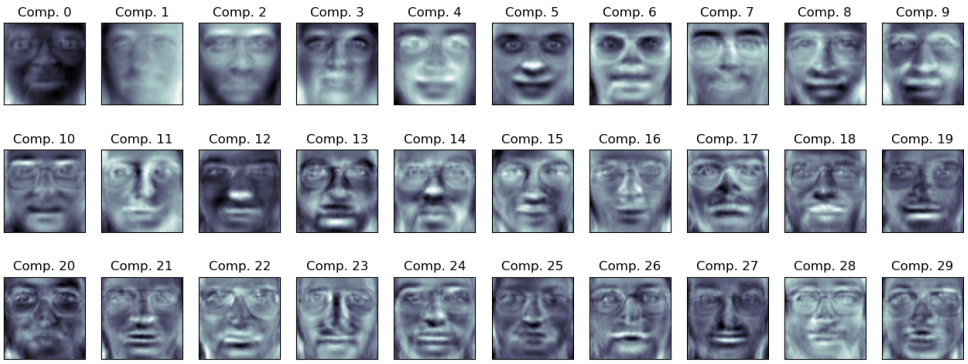


Figure 7: Scikit-learn PCA - First 30 Principal Components of 400 faces

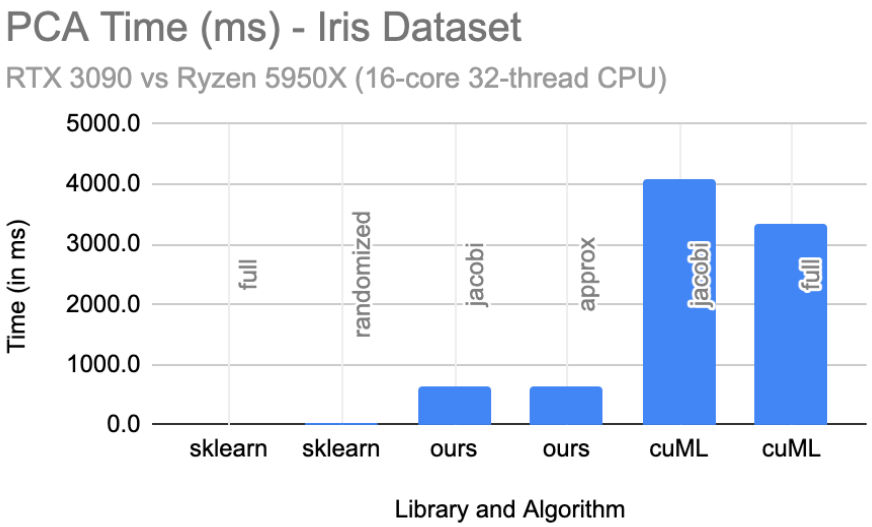


As the above images show, our PCA implementation extracts the same principal components from the data as the Scikit-learn implementation. Specifically, our results are accurate to 1e-7, which is more than enough for PCA use-cases. It is also worth noting that Scikit-learn applies a post-processing step to the eigenvectors so all its solvers return the same-signed output. This can be seen in principal component 0, as our image is light and the Scikit-learn output is dark. This is not related to correctness, as the eigenspace constructed is equivalent. Accordingly, the same faces in each component are visible regardless of lightness or darkness.

4.3. PCA Benchmarking

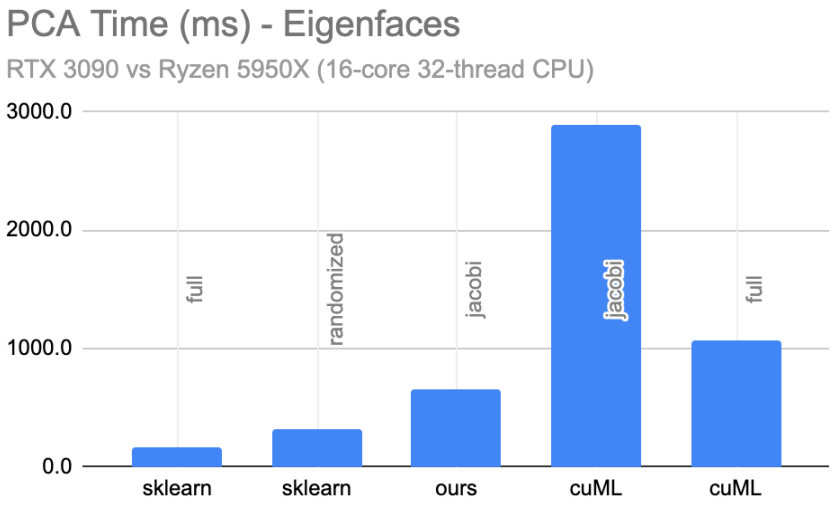
We created a python scripts to run benchmarks for cuML PCA

Figure 8: PCA Performance - Iris Dataset



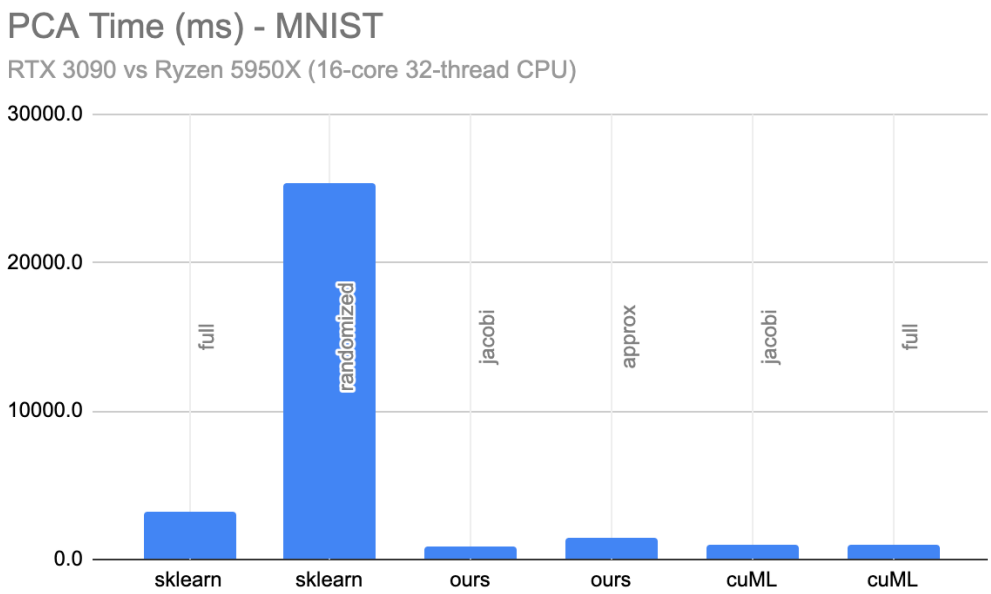
As expected, scikit-learn significantly outperforms the GPU implementations for a tiny dataset like Iris. Interestingly, cuML took significantly longer to get started. Our repository is far more barebones than theirs, which likely explains the discrepancy. Our execution time is dominated by the PCI-e 4.0 transfer latency.

Figure 9: PCA Performance - Eigenfaces Dataset



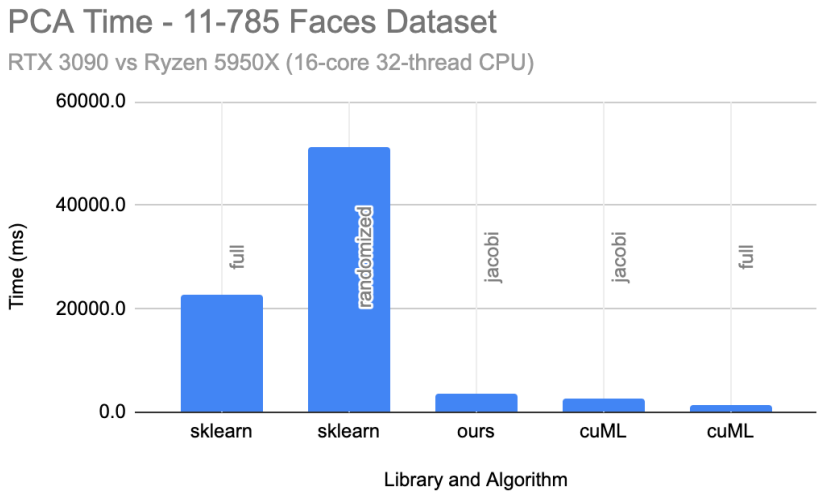
The Eigenfaces dataset makes for a more interesting case, but scikit-learn still outperforms the GPU implementations. Recall that this is against an extremely powerful CPU, so a weaker CPU may not beat the GPU implementations on Eigenfaces.

Figure 10: PCA Performance - MNIST Dataset



Finally, MNIST shows us the power of the GPU. Further, cuML has caught up to our barebones implementation at this point, as the startup costs are amortized over the longer execution time. The poor performance of the scikit-learn randomized solver is interesting, and is likely because we are retrieving all 784 principal components of the dataset. If we were retrieving a small subset of principal components, the randomized solver would fare much better in the benchmark.

Figure 11: PCA Performance - Faces Dataset



The 11-785 Faces dataset is where the gap truly widens. Scikit-learn is significantly slower for our 10,000x4096 dataset. This dataset is not even extremely large, so this gap is very relevant for practical data science workloads. Further, it is of note that cuML's divide-and-conquer eigensolver performed so well here. This is the canonical cuSOLVER GESVD call we mentioned above. Typically the Jacobi solver performs well on larger datasets, but the DnC method worked very well here. The Jacobi solver would also show significant improvements if we decreased the number of principal components from the full 784.

4.4. PCA Profiling

Our PCA function, perfom_pca, call performs four major functions/operations: mean_shift(), perform_svd(), CUDA memory operations after SVD, and pca_from_S_U(). These calls can be found in perform_pca() in [pca/src/pca.cu in our PCA repo]. The execution used to obtain the measurements below uses the Jacobi solver and runs on MNIST 784. Below is a table of total execution time for each function. We make sure to call cudaDeviceSynchronize() before measuring.

Figure 12: Profiling Times by Function

Function Name	Time(ms)
perform_pca()	**173.496**
mean_shift()	27.481
perform_svd()	131.082
Memory after svd	0.44132
pca_from_S_U	14.4895

perform_pca() shows the total time, it includes all the other operations/functions in the table. mean_shift(): is a function that zero means shifts the data on a column by column bases. It first takes a host matrix and puts it in device memory. mean_shift() has two calls to cublas and makes two calls to CUDA kernels we wrote. Those four operations make good use of parallelism, but there exists a sequential dependency between the operations, i.e., each operation must finish before the next can start, because they are reliant on computed data from the previous operation.

perform_svd(): here we call the SVD solver. This function takes the longest time out of all of our functions. We found the best ways to improve the performance of the Jacobi solver was to lower the number of sweeps and relax the tolerance requirement. We experimented with batching in our SVD approximate solver but the memory alignment requirements of its input is unfortunate for C++ programming. Each batch was next to each other in memory, but each batch had to maintain column order internally. Our function to convert a matrix into column major, included an option to split it into batches. We however found that the fine tuning options, tolerance and max

sweeps, in the Jacobi algorithm provided better performance, and with the added benefit of limiting memory realignment operations.

Transferring two of the SVD output matrices to host memory was a relatively low cost operation.

pca_from_S_U: We implemented a CUDA kernel to multiply a vector to a matrix, where each element in the matrix mapped to a column. Originally we wrote a function that converted the vector into a diagonal matrix and performed the needed operation with cuBLAS. However we found that approach used additional memory, and required memory transfer, which could be avoided by using a CUDA kernel. Our CUDA kernel approach also allows us to do the multiplication for N components instead of for each column. We include scripts to profile our code with Nsight Systems and Nsight Compute in our repository.

4.5. KPCA Benchmarking

In this section, “cuML” refers to our cuML KPCA implementation. We ran benchmarks subsets of the MNIST-784 dataset, i.e., we only ran KPCA on the first rows of MNIST. We performed experiments from 100 to 10,000 rows.

Note that the x axis of the following plots is in logarithmic scale. Also note that we did not run against a sequential, single-core KPCA, as this would be an inappropriate baseline for a practical data science workload. We passed in the option to sklearn to use all available processors for computation. The image below is from a Scikit-learn execution on MNIST with 10,000 rows. Each of the Ryzen 5950X’s 32 cores is at around 100% use for most of the execution.

Figure 13: Workstation CPU/RAM Performance Visualization with htop

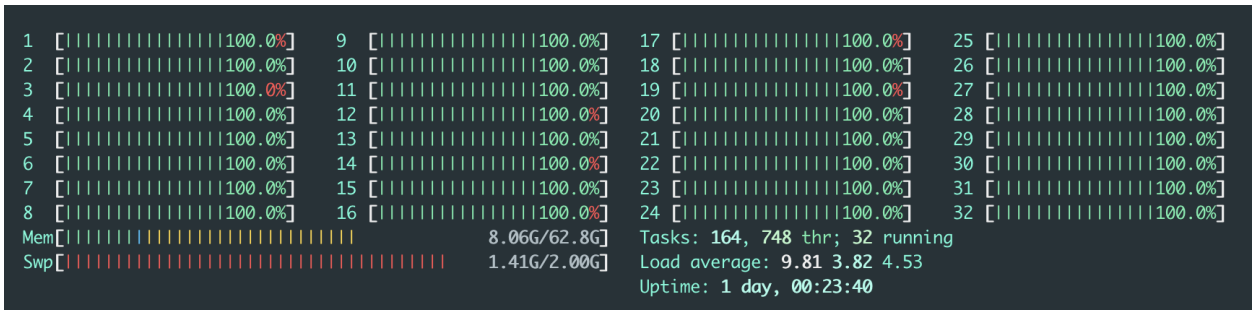
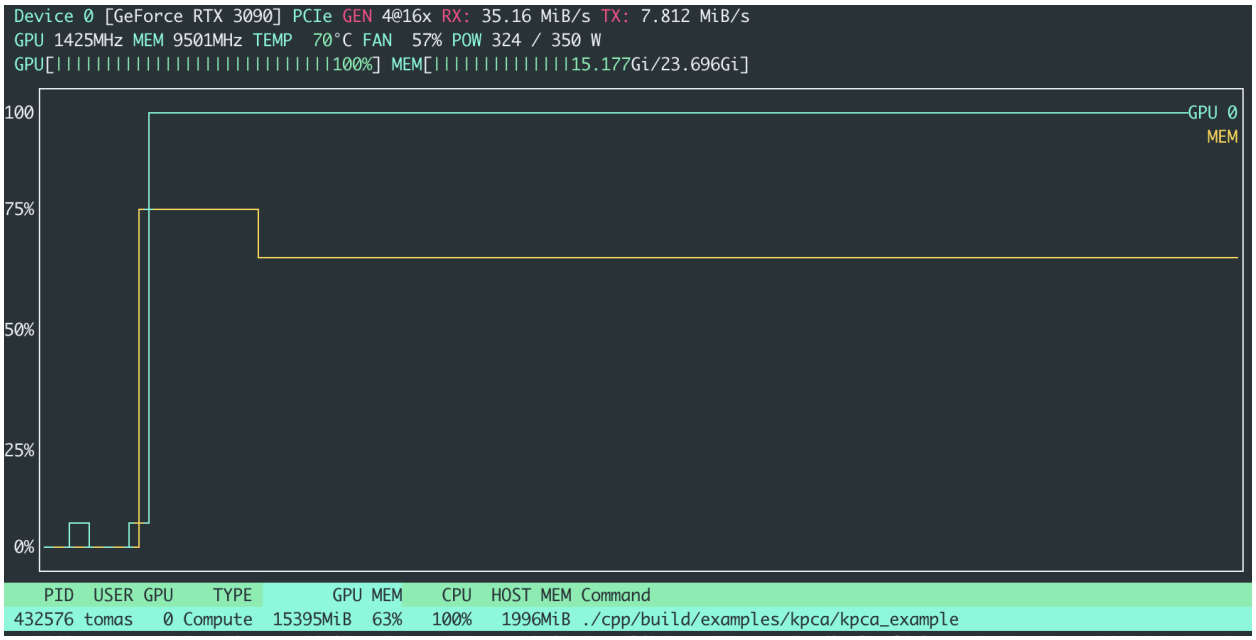
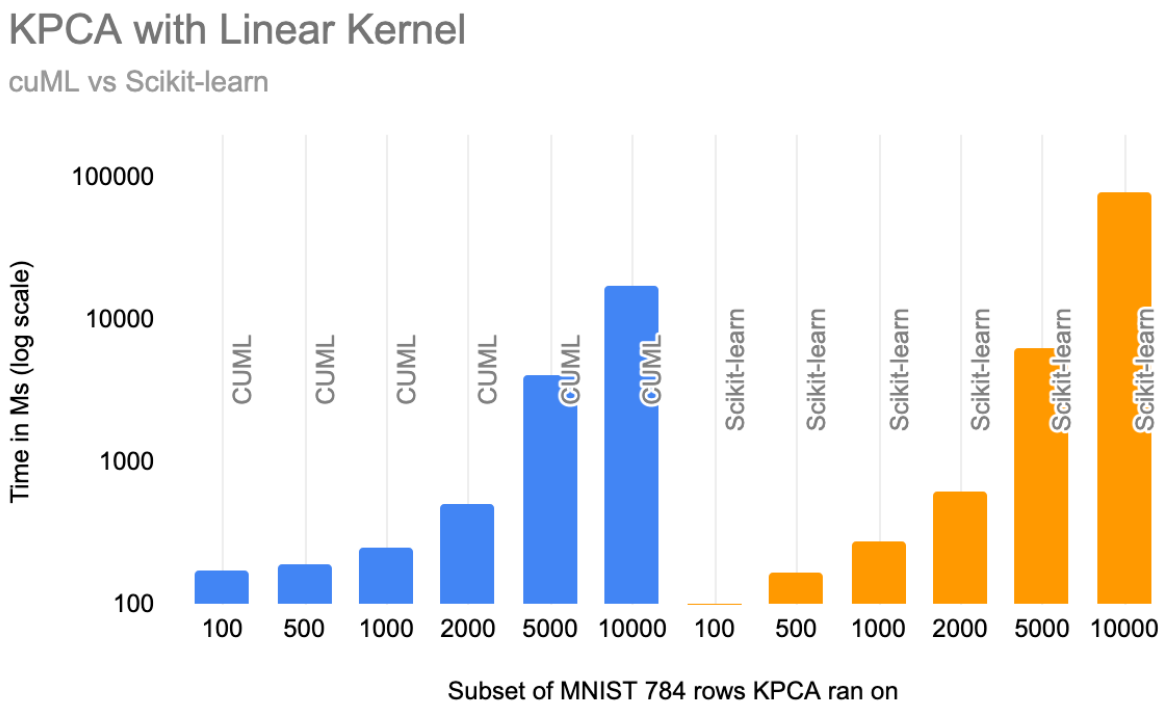


Figure 14: Workstation GPU Performance Visualization with nvtop



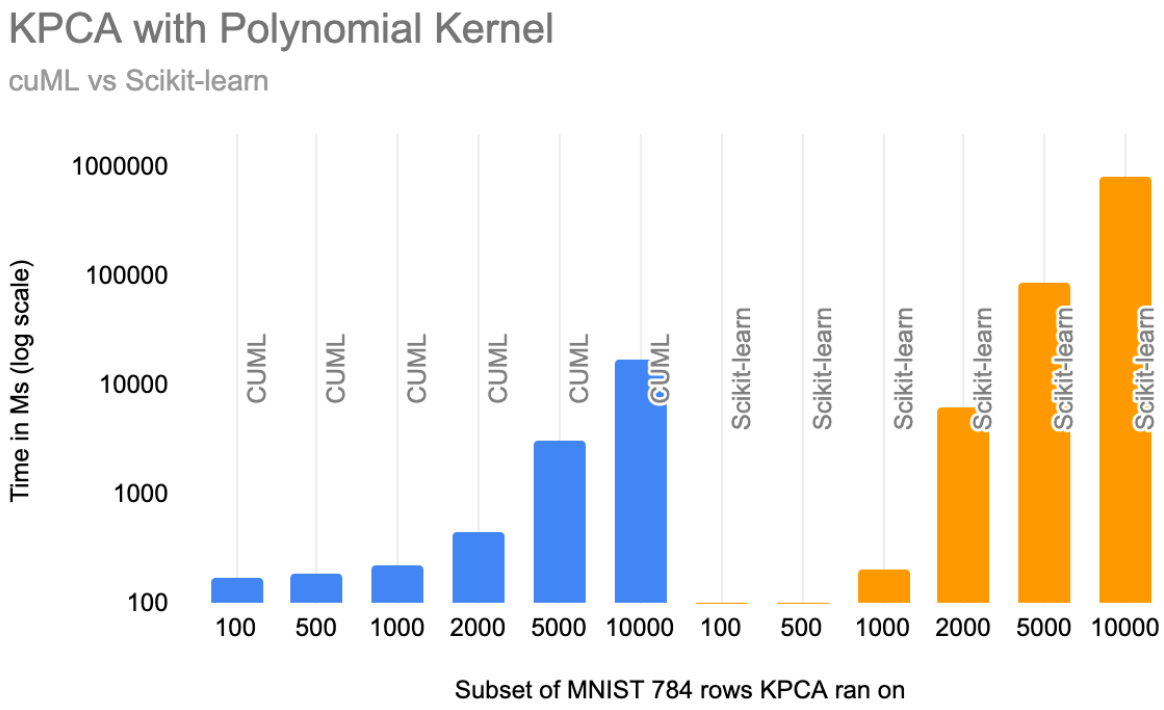
4.5.1 KPCA with Linear Kernel



The above figure shows KPCA execution times with a linear kernel. Overall execution time changes little from 100 rows to 500 rows for cuML. This is in line with our expectations. Our KPCA algorithm has a certain overhead associated with running on GPU, mainly memory

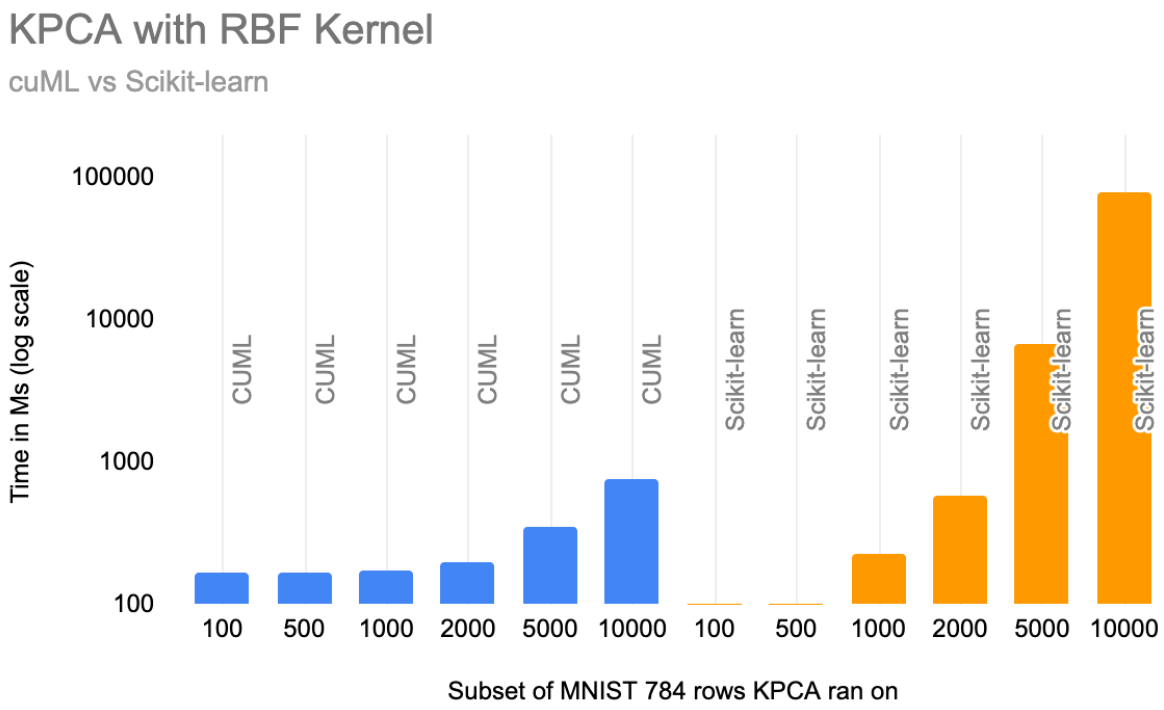
overhead. However with larger data our implementation outperforms Scikit-learn. For 10,000 rows our KPCA implementation finishes the task in around 17 seconds, while Scikit-learn's takes 79.5 seconds.

4.5.2 KPCA with Polynomial Kernel



The above figure shows KPCA execution times with a polynomial kernel. Much of the same applies to this benchmark test as the one in 4.5.1. Noteworthy is how poorly Scikit-learn KPCA handles larger data with a polynomial kernel. Executing 10,000 rows of MNIST took over 13 minutes for Scikit-learn compared to 17 seconds for our implementation. We believe this is because the polynomial kernel works with more non zero components. As an example, on the 150x4 (samples x features) iris dataset and with every non zero component kept, kpca with a linear kernel outputs alpha values of 150x4, while kpca with the polynomial kernel outputs alpha values of 150x148. This creates more gpu friendly computation for the eigensolver.

4.5.3 KPCA with RBF Kernel



Note that the y axis is still logarithmic. The difference in execution time is so great, that when a linear y axis is used the cuML columns disappear from the image.
The trend from 4.5.1 and 4.5.2 continues, our implementation outperforms scikit-learn onlarger datasets. Executing 10,000 rows of MNIST took over a minute for Scikit-learn compared to 764 ms for our implementation.

The RBF kernel creates output data with more features, like the polynomial kernel does. Continuing with the iris example from 4.5.2, the shape of the output alpha values from the linear kernel are 150x4, while the shape with RBF is 150x34.

5. References

1. Face recognition using eigenfaces - Computer Vision and Pattern Recognition, 1991
2. Parallel GPU Implementation of Iterative PCA Algorithms, 2008
3. rapidsai/cuml: cuML - RAPIDS Machine Learning Library
4. Kernel Principal Component Analysis
5. GPU-based parallel kernel PCA feature extraction for hyperspectral images
6. cuSOLVER :: CUDA Toolkit Documentation
7. cuBLAS :: CUDA Toolkit Documentation
8. cuML Kernel PCA Pull Request
9. rapidsai/cuml: cuML - RAPIDS Machine Learning Library
10. rapidsai/raft: Rapids Analytics Framework Toolset
11. sklearn.decomposition.KernelPCA — scikit-learn 0.24.2 documentation
12. UCSD ML Systems - RAPIDS Talk: The Platform Inside and Out Release 0.16
13. CMU 10-405/10-605 - PCA and Kernel PCA slides
14. [feature request] Kernel pca · Issue #1317 · rapidsai/cuml

6. Division of Work

Equal work was performed by both project members.

7. Repositories

Our custom PCA repository
Our cuML fork
- Feature branch
- Benchmarking branch
- Our pull request into cuML