The Report Committee for Umair Shahzad

certifies that this is the approved version of the following report:

Machine Learning Algorithms using Performance Portability Framework for Python

SUPERVISING COMMITTEE:

Milos Gligoric, Supervisor

George Biros, Reader

Machine Learning Algorithms using Performance Portability Framework for Python

by

Umair Shahzad

Report:

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2022

Acknowledgments

I would like to express my gratitude to my advisor Dr. Milos Gligoric for his mentorship as well as my labmates, Nader Al Awar and Pengyu Nie, for providing me with all the resources and insights for pursuing this project, and writing this report. Finally, thank you to my family and friends for their help and support during my masters.

Machine Learning Algorithms using Performance Portability Framework for Python

Umair Shahzad, M.S.

The University of Texas at Austin, 2022

Supervisor: Milos Gligoric

In the past decade, C++ has emerged as one of the main languages for high performance computing. Frameworks such as OpenMP [4], CUDA [10], and HIP [13] target different hardware architectures and have different APIs. The Kokkos [7] programming model provides a way to abstract the underlying APIs for different architectures and build code for different targets by configuring at compile time. Python programming language, on the other hand, is one of the top choices for developers today because of its ease of use and the fact that most recent machine learning and data science libraries are developed targeting this language. To provide performance portability in Python as well, PyKokkos [11, 12] was developed. PyKokkos allows developers to write portable kernels which are translated to C++ Kokkos at runtime.

This report provides an insight into the feasibility of implementing machine learning algorithms using PyKokkos. We first look into the performance difference in commonly used functions in basic machine learning algorithms like, variance, log, transpose, etc. when implemented in PyKokkos as compared to NumPy [9]. We then implement Scikit-learn's [6] Gaussian Naive Bayes and Logistic Regression using PyKokkos, and discuss the performance differences in the two implementations. The results show evidence that performance gains from using PyKokkos can be significant for individual functions, and further work is needed to enable efficient algorithms.

Contents

List of Tables

List of Figures

1 Introduction

Different frameworks written in C++ like CUDA, OpenMP, and HIP, all have a different set of APIs which target different underlying hardware architectures. This hinders the program's ability to be easily run on different architectures/targets without significantly changing it to make it compatible with other frameworks. This not only requires rewriting a large chunk of code but also introduces the need to learn new frameworks and maintain different versions of the same code for different frameworks. The Kokkos framework [7] was developed to solve the above issues and make code portable by providing layers of abstraction over the actual underlying frameworks. It builds code for different targets and architectures from the configuration input at *compile* time. This means that a single code written using the Kokkos framework can run on different architectures with very slight or no modifications at all.

Even though Kokkos was developed (in C++) to facilitate the scientific and machine learning community, it did not fully cater to everyone's needs. This is because people with limited coding and computer science knowledge generally find memory management and static typing difficult to work with. Due to the aforementioned challenges, one of the most prevalent languages among the scientific and machine learning community today is Python because of its ease of use, minimal learning curve, and availability of a huge set of tools and libraries which support data science, machine learning and scientific computing. To this end, PyKokkos was developed by Al Awar et al. [11, 12] to provide the performance portability provided by Kokkos in Python. PyKokkos allows developers to specify the execution space and write portable kernels while maintaining performance equivalent to that of Kokkos. PyKokkos achieves this by translating the kernels to C++ Kokkos at runtime.

In this report we explore how machine learning algorithms implemented in PyKokkos perform compared to the algorithms implemented by Scikit-learn [6]. Essentially, we compare our implementation of Gaussian Naive Bayes [8] and Logistic Regression [3], written using PyKokkos against the Scikit-learn's implementation of these algorithms. In order to do that, we first compare how some of the most common functions used by machine learning algorithms (like variance, transpose, mean) perform against the same implementations by one of the most commonly used array types in the Python scientific community: NumPy [9]. We then compare the performance of our implementation of the algorithms with the Scikit-learn's implementation. Finally, we evaluate the feasibility of using PyKokkos for implementing machine learning algorithms.

All the functions and machine learning algorithms implemented as part of this report are publicly available at https://github.com/kokkos/pykokkos, the official PyKokkos repository.

2 Background

This section introduces important concepts related to PyKokkos and its corresponding APIs that are used to implement the selected machine learning (ML) algorithms.

2.1 Writing a Parallel For Kernel using PyKokkos

The example in Figure 1 shows a transpose kernel written using PyKokkos. The `@pk.workunit` decorator, in the first line, defines the code which will be executed by each parallel running thread. The main function, in line 27, sets the default execution space to OpenMP. An execution space defines the type of processors which will be running the operation. Line 28 initializes a PyKokkos view. A view in PyKokkos is a multidimensional array-like data structure. The constructor of the view class is passed the dimensions as the first argument and the type as the second argument. Thus, the defined view is a two dimensional array with 10 rows and 5 columns, and data type `pk.double`. The view is filled with 1's and passed to the transpose function. To run the workunit on each of the rows of the view, in line 17, `parallel_for` is called. The constructor of the `parallel_for` is passed the execution policy as the first argument. The execution policy defines how the parallelism would be carried out. In our example, we are simply passing a number `n` to the policy which means that we run our operation for all the values in the range `0` to `n`. The second argument to the constructor is the name of the workunit. Finally, any data is passed as key-value arguments which is required by the parallel running threads for carrying out the computation in the workunit.

```python
@pk.workunit
def transpose_impl_2d_double(
        tid: int,
        view: pk.View2D[pk.double],
        out: pk.View2D[pk.double]):

    for i in range(view.extent(1)):
        out[i][tid] = view[tid][i]

def transpose(view):
    if view.rank() == 1:
        return view

    if view.rank() == 2:
        if str(view.dtype) == "DataType.double":
            out = pk.View(view.shape[::-1], pk.double)
            pk.parallel_for(
                view.shape[0],
                transpose_impl_2d_double,
                view=view,
                out=out)
            return out

    raise RuntimeError("Transpose supports 2D views only")

def main():
    pk.set_default_space(pk.ExecutionSpace.OpenMP)
    a: pk.View2D[pk.double] = pk.View([10, 5], dtype=pk.double)
    a.fill(1)
    transpose(a)
```

Figure 1: Transpose function (`RangePolicy`).

The workunit, in addition to the supplied parameters, has an integer as the first parameter. This integer represents the work index which is the unique thread ID. The next line runs a for loop from `0` to the number of columns in the input view. The `extent` function on a view simply returns the length of a particular dimension of the view. The value of the iterator in addition to the thread ID are used to then calculate the transpose of a single row. All of the threads perform this computation for each row. The final result is written to the `out` view.

This example can be run using CUDA simply by changing the default execution space to `pk.ExecutionSpace.Cuda`. PyKokkos internally manages the memory spaces and layouts to ensure optimal execution in different execution spaces.

2.2 Writing a Parallel Reduce Kernel using PyKokkos

Similarly, PyKokkos also supports the parallel reduce pattern. To better understand the working of a parallel reduce, let's consider an example of the `sum` function which takes as input a one dimensional view and returns the sum of all the elements of the view. Instead of calling `parallel_for`, we call the `parallel_reduce` function. The parameters of the `parallel_reduce` function are similar to those of a `parallel_for` parameters with the exception of the optional `initial_value` which simply sets the initial value of the result accumulator. The workunit associated with the parallel reduce pattern, in addition to the thread ID and any developer-defined inputs, also has the accumulator as the second parameter. The accumulator is responsible for storing intermediate values as a result of the reduction. Once the reduction is complete, the final value of the accumulator is returned.

2.3 PyKokkos Execution Policy

We provide some background on Range and Team execution policies.

2.3.1 RangePolicy

The `RangePolicy` [14] defines the execution space for a one dimensional iteration space only. Like in the previous example, when we pass a number `n` to the execution policy, we simply divide the indices from `0` to `n` among the threads which are responsible for carrying out the parallel work. We see that using the `RangePolicy`, the best we could do was parallelize for each row only. The work inside each thread is still iterative. The effect of this policy on performance for multidimensional arrays is clearly visible when the number of columns gets large.

2.3.2 TeamPolicy

The `TeamPolicy` [15] like the `RangePolicy`, defines the execution space for a one dimensional iteration space. The difference is that instead of assigning each row to a single thread, a team of threads is assigned each row. Team policies further allow us to use nested policies in the workunits which means that we can further parallelize and divide the work in each row among the team of threads responsible for the row. The example in Figure 2 shows the same transpose function from the previous example but this time using the `TeamPolicy`. The workunit associated with the `parallel_for` using the `TeamPolicy` has the team member as the first parameter instead of the thread ID. Calling `league_rank` on the team gets the row index the team is

```python
@pk.workunit
def transpose_impl_2d_double(
        team: pk.TeamMember,
        view: pk.View2D[pk.double],
        out: pk.View2D[pk.double]):

    n: int = team.league_rank()

    def team_for(i: int):
        out[i][n] = view[n][i]

    pk.parallel_for(
        pk.TeamThreadRange(team, view.extent(1)),
        team_for)

def transpose(view):
    if view.rank() == 1:
        return view

    if view.rank() == 2:
        if str(view.dtype) == "DataType.double":
            out = pk.View(view.shape[::-1], pk.double)
            pk.parallel_for(
                pk.TeamPolicy(view.shape[0], pk.AUTO),
                transpose_impl_2d_double,
                view=view,
                out=out)
            return out

    raise RuntimeError("Transpose supports 2D views only")
```

Figure 2: Transpose function (`TeamPolicy`).

responsible for. Next we define a function, the computational body, we want each of the team members to run. Finally we run the `parallel_for` which simply splits the indices from `0` to the number of columns of the input view among the threads in the team.

2.4 Scikit-learn

Scikit-learn [6] is an open-source library of a wide range of machine learning algorithms for Python. It uses NumPy [9] and Cython [5] internally for high performance. In this report, we compare the performance of Scikit-learn's version of Gaussian Naive Bayes and Logistic Regression with the PyKokkos equivalents to judge the feasibility of implementing performance portable ML algorithms with PyKokkos.

3 Technique

This section describes our approach to implementing the Machine Learning (ML) algorithms with PyKokkos. The entire process was divided into three steps.

The first step involved finding the most commonly used functions in some of the basic ML algorithms. This step entailed scanning the Scikit-learn's implementation of various ML algorithms, and filtering out all the operations being performed on the NumPy arrays.

The second step was to implement the functions filtered out in step one in the PyKokkos framework and evaluate the performance of the functions compared to their NumPy counterparts.

Lastly, we implemented the Scikit-learn's version of Gaussian Naive Bayes and Logistic Regression using PyKokkos and evaluated their performance against their Scikit-learn's versions.

3.1 Collecting Functions

In order to implement the actual ML algorithms, the first step was to discover the basic functions that are required to build a complete algorithm. To find all the functions which work on arrays, we implemented a simple Python script that finds all the uses of NumPy functions in a file using a simple regex. This script was run on Scikit-learn's code for Gaussian Naive Bayes and Logistic Regression. The functions collected from these algorithms are listed in Table 1.

Table 1: List of functions that was discovered in Scikit-learn.

Function	**Description**	**Input Shapes**
Add	Sums positionally corresponding elements of the two input views	(1D, 1D), (2D, 0D)
Subtract	Subtracts positionally corresponding elements of the two input views	(1D, 1D)
Multiply	Multiplies positionally corresponding elements of the two input views	(1D, 1D)
Matmul	Matrix multiplication	(1D, 2D)
Divide	Divides positionally corresponding elements of the two input views	(1D, 1D), (2D, 0D)
Negative	Computes element-wise negative	(1D)
Positive	Computes element-wise positive	(1D)
Power	Return a view with each val in viewA raised to the positionally corresponding power in viewB	(1D, 1D), (2D, 0D)
Fmod	Element-wise remainder of division when element of viewA is divided by positionally corresponding element of viewB	(1D, 1D)
Greater	Return the truth value of viewA > viewB element-wise	(1D, 1D)
Logaddexp	Return a view with $log(exp(a) + exp(b))$ calculate for positionally corresponding elements in viewA and viewB	(1D, 1D)

Table 1: cont.

Function	**Description**	**Input Shapes**
Logaddexp2	Return a view with $log(2^a + 2^b)$ calculated for positionally corresponding elements in viewA and viewB	(1D, 1D)
Floor divide	Divides positionally corresponding elements of viewA with elements of viewB and floors the result	(1D, 1D)
Sin	Element-wise trigonometric sin of the view	(1D)
Cos	Element-wise trigonometric cos of the view	(1D)
Tan	Element-wise tan of the view	(1D)
Logical and	Return the element-wise truth value of viewA $\wedge$ viewB	(1D, 1D)
Logical or	Return the element-wise truth value of viewA $\vee$ viewB.	(1D, 1D)
Logical xor	Return the element-wise truth value of viewA $\oplus$ viewB.	(1D, 1D)
Logical not	Element-wise logical_not of the input view	(1D)
Fmax	Return the element-wise fmax of the input view	(1D)
Fmin	Return the element-wise fmin of the input view	(1D)
Exp	Element-wise exp of the input view	(1D)
Exp2	Element-wise 2**x of the view	(1D)

Table 1: cont.

Function	**Description**	**Input Shapes**
Var	Computes variance along specified axis	(2D)
In1d	Test whether each element of a 1-D view is also present in a second view	(1D); 1D
Mean	Computes mean along specified axis	(1D)
Transpose	Reverses the axis of the input view	(2D)
Index	Index a view with another view	(1D, 1D)
Sum	Return the sum of all the elements of the input view. For 2D inputs; return sum along the specified axis	(1D), (2D)
Linspace	Return evenly spaced numbers over a specified interval	(1D)
Logspace	Return evenly spaced numbers over a specified interval on log scale	(1D)

3.2 Implementing Commonly Used Functions in Machine Learning Algorithms with PyKokkos

The second step involved implementing the functions collected in step one, using the PyKokkos framework. The implementation of each of the functions involved designing a function which takes as input the views and triggers the appropriate workunits based on the input types, dimensions, and axis.

An example implementation of the add function is provided in Figure 3. The add function takes as input two views and returns the sum of positionally corresponding elements of the viewA with viewB. In line 18, we check if the viewB is not of type pk.View, i.e., viewB is a scalar, and convert viewB to a view with a single dimension if the condition is true. The rest of the code in the body of the add function simply checks the dimensions of the input views and triggers the corresponding workunit. The need to create multiple workunits arises from the fact that the workunits are directly translated into C++ code and the function cannot be passed data of different types and dimensions than what translated function actually expects. For this very reason, PyKokkos ensures that the workunit function parameters are always type annotated otherwise it raises an error. Because of this, a workunit, for example, which expects a 1D view of type double cannot accept a view with a type different than double or a dimension other than one.

Once the add function body satisfies a condition, it creates a new output view of appropriate type and shape, and triggers the parallel_for on the corresponding workunit. There are three different workunits defined in the figure. The first workunit, add_impl_1d_double, expects all the views to be of type pk.double and single dimension. The body of the work unit simply sums the corresponding elements of the two input views and stores the sum in the output view. The second workunit, add_impl_1d_float, works exactly like the previous work unit but only for floats. Lastly, the third workunit, add_impl_2d_1d_double, only works when the viewA is two dimensional whereas the viewB is one dimensional. The body of this function simply iterates over each column of viewA and adds the scalar number in viewB to

```python
@pk.workunit
def add_impl_1d_double(tid: int, viewA: pk.View1D[pk.double],
        viewB: pk.View1D[pk.double], out: pk.View1D[pk.double],
            ):
    out[tid] = viewA[tid] + viewB[tid]

@pk.workunit
def add_impl_1d_float(tid: int, viewA: pk.View1D[pk.float],
        viewB: pk.View1D[pk.float], out: pk.View1D[pk.float]):
    out[tid] = viewA[tid] + viewB[tid]

@pk.workunit
def add_impl_2d_1d_double(tid: int, viewA: pk.View2D[pk.double],
        viewB: pk.View1D[pk.double], out: pk.View2D[pk.double]):
    for i in range(viewA.extent(1)):
        out[tid][i] = viewA[tid][i] + viewB[i % viewB.extent(0)]

def add(viewA, viewB):
    if not isinstance(viewB, pk.View):
        view_temp = pk.View([1], pk.double)
        view_temp[0] = viewB
        viewB = view_temp

    if viewA.rank() == 2:
        out = pk.View(viewA.shape, pk.double)
        pk.parallel_for(viewA.shape[0], add_impl_2d_1d_double,
            viewA=viewA, viewB=viewB, out=out)

    elif str(viewA.dtype) == "DataType.double" and
        str(viewB.dtype) == "DataType.double":
        out = pk.View([viewA.shape[0]], pk.double)
        pk.parallel_for(viewA.shape[0], add_impl_1d_double,
            viewA=viewA, viewB=viewB, out=out)

    elif str(viewA.dtype) == "DataType.float" and
        str(viewB.dtype) == "DataType.float":
        out = pk.View([viewA.shape[0]], pk.float)
        pk.parallel_for(viewA.shape[0], add_impl_1d_float,
            viewA=viewA, viewB=viewB, out=out)
    else:
        raise RuntimeError("Incompatible Types")
    return out
```

Figure 3: Add function implementation using PyKokkos.

every element in the row `tid`.

It is important to note that this implementation of the `add` function is execution space independent (i.e., portable). The developer using the add function can set the execution space of their choice and PyKokkos will ensure any views created in the function have the correct layout and memory space for optimal execution in the defined execution environment.

All functions which perform basic operations like `add`, `subtract`, `multiply`, `divide`, `square`, `float_divide`, `logical_and`, `logical_or`, `logical_not`, and `logical_xor` are implemented in the same way.

Next we have the functions like `fmod`, `fmax`, `fmin`, etc. which require using an operator from the C++'s `cmath` library. An example of such a function (`fmax`) is given in Figure 4. The basic `fmax` takes as input two views and returns a view which has the element-wise max of the elements in the two input views. The body of the `fmax` function is very similar to the body of the `add` function which we saw earlier. The difference is in the workunits of `fmax`. In line 4 and 9, the function `fmax` is called on the elements at index `tid` of both of the views. This function (`fmax`) is undefined in the Python scope but exists in the C++ realm as part of the cmath library. Since PyKokkos always includes cmath in the translated code and the workunits are translated directly into C++ code, the `fmax` function is readily available in the environment. This allows us to write workunits for functions like `sin`, `cos`, `tan`, `fmax`, `fmin`, `fmod`, `exp`, `pow`, `logaddexp`, `logaddexp2`, `exp2`, `log`, `log2`, `log10`, and `log1p` directly using the cmath library functions.

Furthermore, we implemented functions which use `parallel_reduce` instead of the `parallel_for` parallelization pattern. The matrix multiplication

```
@pk.workunit
def fmax_impl_1d_double(tid: int, viewA: pk.View1D[pk.double],
        viewB: pk.View1D[pk.double], out: pk.View1D[pk.double]):
    out[tid] = fmax(viewA[tid], viewB[tid])

@pk.workunit
def fmax_impl_1d_float(tid: int, viewA: pk.View1D[pk.float],
        viewB: pk.View1D[pk.float], out: pk.View1D[pk.float]):
    out[tid] = fmax(viewA[tid], viewB[tid])

def fmax(viewA, viewB):
    if len(viewA.shape) > 1 or len(viewB.shape) > 1:
        raise NotImplementedError("fmax() ufunc only supports 1D
            views")

    if str(viewA.dtype) == "DataType.double" and
        str(viewB.dtype) == "DataType.double":
        out = pk.View([viewA.shape[0]], pk.double)
        pk.parallel_for(viewA.shape[0], fmax_impl_1d_double,
            viewA=viewA, viewB=viewB, out=out)

    elif str(viewA.dtype) == "DataType.float" and
        str(viewB.dtype == "DataType.float":

        out = pk.View([viewA.shape[0]], pk.float)
        pk.parallel_for(viewA.shape[0], fmax_impl_1d_float,
            viewA=viewA, viewB=viewB, out=out)

    else:
        raise RuntimeError("Incompatible Types")
    return out
```

Figure 4: Fmax function implementation using PyKokkos.

and `sum` functions are two such examples where we use `parallel_reduce`. In Figure 5, the matrix multiplication function takes as input two views, and returns the scalar result. The function assumes that both of the views are one dimensional i.e., one view has `n` rows whereas the other view has `n` columns and only one row. Body of the matmul function is very similar to all the functions we have seen earlier like `add` and `fmax`. The only difference is the use of `parallel_reduce` instead of `parallel_for`. The workunits of the `parallel_reduce` function in line 18 and 21, in addition to `tid` (thread ID), also receive an accumulator as an input which is used to store the result of the reduction. The body of the workunits for `matmul` store the sum of the product of corresponding matrix indices. The `sum` function follows the same pattern as the matrix multiplication.

Additionally, we implemented a few functions which perform computations along various axes instead of a fixed one. The `mean` function is a good example of this. A two dimensional array can have its mean calculated along each column or each row depending on the requirements of the developer. To cater computation along multiple axes, we design multiple workunits and based on the axis input, and choose the work unit based on the input. Some other examples of such functions are `sum` and `variance`.

Finally, we implemented the functions which depend on the functions we have coded as of now and use their result as the inputs. One such example is the `variance`. The `variance` function gets a two dimensional view as input and returns the variance along an axis as output. To calculate the variance along an axis, our workunits, in addition to the original view, require a view of mean along the same axis, which is calculated using the mean function we implemented earlier.

```
@pk.workunit
def matmul_impl_1d_double(tid: int, acc: pk.Acc[pk.double],
        viewA: pk.View1D[pk.double], viewB:
            pk.View2D[pk.double]):
    acc += viewA[tid] * viewB[0][tid]


@pk.workunit
def matmul_impl_1d_float(tid: int, acc: pk.Acc[pk.float],
        viewA: pk.View1D[pk.float], viewB: pk.View2D[pk.float]):
    acc += viewA[tid] * viewB[0][tid]

def matmul(viewA, viewB):
    if len(viewA.shape) != 1 or viewA.shape[0] != viewB.shape[0]:
        raise RuntimeError(
            "Input operand 1 has a mismatch in its core dimension
                (Size {} is different from
                {})".format(viewA.shape[0], viewB.shape[0]))

    if str(viewA.dtype) == "DataType.double" and
        str(viewB.dtype) == "DataType.double":
        return pk.parallel_reduce(viewA.shape[0],
            matmul_impl_1d_double, viewA=viewA, viewB=viewB)

    elif str(viewA.dtype) == "DataType.float" and
        str(viewB.dtype) == "DataType.float":
        return pk.parallel_reduce(viewA.shape[0],
            matmul_impl_1d_float, viewA=viewA, viewB=viewB)

    else:
        raise RuntimeError("Incompatible Types")
```

Figure 5: Matmul function implementation using PyKokkos.

3.3 Implementing Machine Learning Algorithms

As the next step, we implemented two machine learning algorithms, namely Gaussian Naive Bayes and Logistic Regression, using PyKokkos. We then compared their performance with their Scikit-learn's counterparts; we provide the results of comparison in the next section.

To ensure that the performance comparisons were fair, we copied the exact implementation of the algorithms from the Scikit-learn's source code and converted them to PyKokkos from NumPy. In order to convert the algorithm, we replaced all the NumPy arrays with PyKokkos views and replaced any NumPy functions operating on the arrays with their PyKokkos equivalent. If a function was not a part of the PyKokkos library, it was implemented and then replaced in the algorithm. There were two different kinds of missing functions in the PyKokkos library, i) functions not existing at all, and ii) existing functions expecting an input of type which is not already supported by the workunits. An example of the former kind of the issue is the scalar multiplication with PyKokkos views. Since we used a regex to figure out the most commonly used functions in the basic machine learning algorithms, we missed some functions which were not invoked explicitly from the NumPy library. For instance, `n_ij = -0.5 * np.sum(np.log(2.0 * np.pi * self.var_[i, :]))`, this code snippet from the original Gaussian Naive Bayes source code [16], multiplies negative 0.5 with a NumPy array, but this operation was not supported in the PyKokkos ecosystem for the views i.e., a view did not support scalar multiplication. We implemented all the missing functions to ensure all the workunits required by the algorithms were available for the PyKokkos views. On the other hand, the latter case included functions like `divide`, or `log` on views of dimension greater than the ones already supported. We simply implemented new workunits which supported inputs of required dimensions.

The next step was to copy any functions from the Scikit-learn's library which our algorithms were dependent on and convert them to PyKokkos too to ensure a fair comparison between the implementations. For this report, we left out any functions which did not significantly impact the performance of our algorithms, and imported them directly from the Scikit-learn's library instead. An example is the `check_is_fitted` [17] method which simply checks if a list of attributes passed to it contains certain strings. Some other examples included the validation functions for the inputs and intermediate results. Most of these functions required NumPy arrays as input and returned NumPy arrays too. Since PyKokkos interoperates with NumPy, we did not have to change anything for these inputs. However, the output of these functions had to be converted to PyKokkos views from NumPy arrays for them to be compatible with PyKokkos kernels, which introduces some overhead.

Finally, we tested the result of Gaussian Naive Bayes and Logistic Regression implemented in PyKokkos with the result of their Scikit-learn counterparts to confirm the correctness of the implemented algorithms. This step involved training our algorithms on the Iris [1, 2] dataset, and comparing their mean accuracies and prediction results against the Scikit-learn's algorithms when they were trained and tested on the same inputs. The final implementations were then subject to stress testing, the results of which can be found in the next section.

4 Evaluation

All of the above algorithms and functions were benchmarked on Ubuntu 22.04, with Python 3.10.6, Kokkos 3.7.0, NumPy 1.23.3 and PyKokkos latest *develop* (SHA: 679a21ebf237241b93a467c6aa3a4577b0ba6ea5) branch. The system had a 2.2 GHz Intel Core i7 processor with 6 physical and 12 logical cores, and 16 GB 2400 MHz DDR4 memory. PyKokkos was configured to use all 12 cores for parallelization whereas NumPy was run with the default configuration.

4.1 Benchmarking Functions

For each of the functions that we implemented (see Table 1) in this study, we compared its performance against its NumPy equivalent. Even though each PyKokkos version of the function has multiple workunits associated with it, we only benchmarked a single workunit for each function. The Table 2 below shows the shape of the input arrays/views for different functions that were put to test.

We tested each of the functions against inputs (views/arrays) of 6 different lengths where the the length, in terms of number of elements, of the inputs ranged from 10^1 to 10^6. The data type of each of the input can be found in Table 2. For each function, we timed the execution of 50 individual runs for each of the input array/view size and reported two different results:

i) mean of the last 40 runs,

ii) median of the last 40 runs (see Table 5 in the appendix).

For both of the results, we skipped the time of the first 10 runs from the final result to make sure that any initializations or the initial translation of the workunits to C++ does not impact the final result. We do this for both PyKokkos and NumPy functions to maintain the integrity of the results. The final results for mean times are reported in Table 2.

Table 2: Performance comparison against NumPy (mean [ms]).

	10^1	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
Var *(2D view of shape[x, 10])*						
pk	0.18	0.20	0.22	0.36	1.3	34.0
np	0.1	0.11	0.13	0.83	8.6	96.0
Mean *(2D view of shape[x, 10])*						
pk	0.11	0.17	0.18	0.16	0.74	13.0
np	0.018	0.12	0.11	0.36	2.8	28.0
Sum *(2D view of shape[x, 10])*						
pk	0.083	0.13	0.09	0.12	0.78	10.0
np	0.029	0.083	0.022	0.18	1.6	16.0
Divide *(2D view of shape[x, 10])*						
pk	0.19	0.14	0.21	0.41	3.7	53.0
np	0.0034	0.0086	0.019	0.12	1.3	26.0
Power *(2D view of shape[x, 10])*						
pk	3.9	0.23	0.33	7.2	4.7	40.0
np	0.0021	0.0058	0.16	0.37	3.3	42.0
Log *(1D view of shape[x])*						
pk	0.19	0.14	0.15	0.68	5.6	50.0
np	0.019	0.026	0.081	0.59	5.5	83.0

Table 2: cont.

	10^1	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
Exp *(1D view of shape[x])*						
pk	0.092	0.15	0.18	0.18	0.95	3.1
np	0.003	0.024	0.031	0.18	1.6	10.0
Logical Not *(1D view of shape[x])*						
pk	0.084	0.11	0.23	0.12	0.36	0.49
np	0.0015	0.0014	0.0038	0.0061	0.03	0.46
Multiply *(1D view of shape[x])*						
pk	0.23	0.17	0.16	0.19	0.76	1.7
np	0.0018	0.0015	0.0031	0.0086	0.051	0.82
Index *(viewA shape[x], viewB shape[x])*						
pk	0.1	0.084	0.11	0.18	0.3	4.0
np	0.0005	0.00045	0.0017	0.016	0.15	1.7
Transpose *(2D view of shape[x, 10])*						
pk	0.24	0.099	0.12	0.27	4.5	34.0
np	0.0023	0.0018	0.0015	0.0016	0.0011	0.0012
All *(1D view of shape[x])*						
pk	0.022	0.044	0.3	2.6	20.0	220.0
np	0.026	0.036	0.0035	0.052	0.097	0.87
Add *(viewA shape[x, 10], viewB shape[10])*						
pk	0.095	0.25	0.32	0.3	3.2	15.0
np	0.0026	0.0043	0.015	0.41	1.7	25.0

Table 2: cont.

	10^1	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
In1d *(viewA shape[x], viewB shape[x])*						
pk	0.28	0.3	1.2	13.0	650.0	47000.0
np	0.043	0.076	0.11	0.54	13.0	84.0
Zeros *(2D view of shape[x, 10])*						
pk	0.028	0.028	0.038	0.031	0.27	17.0
np	0.00056	0.00099	0.0034	0.023	0.32	0.092

The results show that when the size of the inputs is small i.e., in the range 10^1 to 10^4, NumPy outperforms our PyKokkos implementation of the functions. We believe that this can be attributed to the fact that the PyKokkos version of the functions have an extra parallelization overhead.

However, for input sizes greater than 10^4, the result for many of PyKokkos functions is better than their NumPy equivalents. Examples of such functions are `mean`, `var`, `sum`, `exp`, and `add`. The reason for this better performance is that for larger input sizes the reduction in execution time from parallelizing the computation is greater than any addition in the execution time because of the parallelization overhead.

Furthermore, as can be seen in the Table 2, a few of PyKokkos version of the functions have performance comparable to those of their NumPy equivalents when the size of inputs is large. Some examples of such functions are `multiply`, `logical_not`, and `power`.

Finally, there are only a handful of cases where PyKokkos functions performed worse than the NumPy's functions for large sized inputs. Examples of such functions include, `all`, `zeros`, and `transpose`. The slow performance of the `all` function can be attributed to the fact that it checks all the elements of the array sequentially. Currently PyKokkos does not support the parallel reduction pattern which accepts any other operator than '+'. In case of `transpose`, our implementation allocates a new view and transposes by moving the data physically. On the other hand, NumPy transpose works by simply allocating a new array and changing the strides. Since PyKokkos does not support strides for views as of now, we had to stick to manually moving data in case of a transpose operation. Lastly, the implementation of the `zeros` function in PyKokkos is slower because of the additional step of Kokkos framework initializing the PyKokkos views and allocating memory for them. This initialization process is relatively slower than the NumPy's initialization of a zeros array.

4.2 Replacing `RangePolicy` with `TeamPolicy`

After figuring out the performance of our initial implementations, we worked on further optimizing the performance of our functions. As part of this effort, we identified issues with the execution policies of various workunits. The initial implementation ran all of the workunits with the `RangePolicy`, however, this was not optimal for cases where the workunits operated on multidimensional arrays. An example of such a case is the `sum` function. Using the `RangePolicy` we divided the work for each row to a thread and made each thread responsible for calculating the sum of the entire row. In order to fix all instances of this issue, we replaced the use of `RangePolicy` with a `TeamPolicy` and made the computation further parallel by parallelizing the work for each row too. The results after updating the execution policy for all the functions, which take as input multidimensional arrays, namely `var`, `mean`, `sum`, `divide`, and `power`, were benchmarked again.

Table 3: `TeamPolicy` performance comparison against NumPy (mean [ms]).

	10^1	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
Var *(2D view of shape[x, 10])*						
pk	0.18	0.25	0.24	0.34	2.8	25.0
np	0.051	0.056	0.15	0.79	7.8	99.0
Mean *(2D view of shape[x, 10])*						
pk	0.12	0.17	0.12	0.25	1.0	12.0
np	0.034	0.036	0.065	0.33	2.3	24.0
Sum *(2D view of shape[x, 10])*						
pk	0.085	0.12	0.14	0.16	0.95	13.0
np	0.024	0.025	0.04	0.23	1.6	18.0
Divide *(2D view of shape[x, 10])*						
pk	0.19	0.41	0.22	0.82	6.2	55.0
np	0.02	0.025	0.039	0.15	2.2	27.0
Power *(2D view of shape[x, 10])*						
pk	0.18	0.21	0.35	0.98	8.1	43.0
np	0.014	0.018	0.046	0.34	2.4	38.0

As reflected in Table 3, any further parallelization did not help improve the performance of these workunits. We believe that the bottleneck here is the CPU on which we are benchmarking. The low number of cores, i.e., 6 physical and 12 logical cores, do not leave a lot of room for further parallelization when the `TeamPolicy` is used.

4.3 Benchmarking the Algorithms

Lastly, we benchmarked the two machine learning algorithms, Gaussian Naive Bayes and Logistic Regression. The benchmarking methodology was very similar to what we saw above for the individual functions. We used the Iris [1, 2] dataset from the Scikit-learn's dataset library to train each algorithm 100 times on the dataset, and calculated the average time it took for the last 90 runs. We skipped the first 10 runs to make sure we do not include any iterations where a workunit was being translated or any initializations were taking place. We benchmarked the Scikit-learn's version of these algorithms in the same manner. The final execution times can be found in the Table 4.

Table 4: Algorithms performance (mean [ms]).

	PyKokkos	**Scikit-learn**
Gaussian Naive Bayes	3.4	0.81
Logistic Regression	26.1	19.6

The results show that the performance of the PyKokkos version of the machine learning algorithms is about four folds slower than the original Gaussian Naive Bayes and 1.3 times slower than the original Logistic Regression.

The reason for this performance can be attributed to the fact that all of the functions implemented using PyKokkos do not perform as well as their NumPy's equivalents. When the size of the views is smaller than 10^3 elements, as discussed above, the parallelization overhead takes over the performance gained by actually parallelizing. On the other hand, when the size of the views is larger, some of the functions like `max`, `all`, `zeros`, and `transpose` perform poorly because PyKokkos currently does not support the reduction operators required to implement them. Secondly, since we only implemented the nested library functions from Scikit-learn up to a depth of 1, all other functions require conversion of their output to PyKokkos views. This introduces an additional overhead which impacts the overall performance of the algorithms.

5 Future Work

We describe several directions for future work.

5.1 GPU Based Performance Benchmarks

For this report, all of the functions and algorithms implemented using PyKokkos were benchmarked on a CPU with a very small number of cores. However, in the real world, computer systems generally have access to graphics processing units (GPUs) as well. This means that such systems should be able to run the PyKokkos workunits with CUDA and HIP as the execution space. It would be interesting to see how PyKokkos version of the functions we implemented, compare in performance to their NumPy counterparts when the PyKokkos execution space is CUDA or HIP.

5.2 Reduction Operators

Currently, PyKokkos only supports addition as the parallel reduction operator. This means that the parallel reduction can only perform addition on the accumulator value. The algorithms, however, frequently require computing the min, max, product, logical conjunction/disjunction, binary and/or, etc. Without the support for such reductions, we came up with the serial versions of these reductions to temporarily use in the machine learning algorithms. This led to a degradation of the performance and had an impact on our benchmarks.

Since Kokkos already supports many of the reductions listed above, introducing them in PyKokkos should be straightforward. It would be interesting to see how using these operators improve the performance of the algorithms using them internally.

5.3 Broadcasting

Currently, all the functions implemented with PyKokkos assume that the inputs are broadcastable and make use of the modulus function in the workunits to ensure that the views are not indexed beyond their shapes. An example of this are the multiply, divide, add, power, etc. functions. With the `add` functions, for example, we calculate the sum as follows: `out[tid] = viewA[tid] + viewB[tid] % viewB.extent(0)]`. Even though this works fine with the assumption in place that viewB will always be the smaller of the two views in terms of the shape. However, if the same function is passed the inputs in reversed order, we might see unexpected results since some elements of viewA will never be indexed as `tid` will always be smaller than the total number of elements in ViewA. This will yield incorrect results. Such issues can be easily caught if PyKokkos builds a method to verify the broadcastability of the two views and uses the appropriate workunits based on that.

5.4 Implementing Nested Library Functions

As of now, the algorithms we implemented using PyKokkos still relied on the functions from the Scikit-learn library. We only implemented the dependent functions up to a depth of 1 from within the Scikit-learn library and skipped any further nested functions. Using these function adds extra overhead since their output has to be converted to PyKokkos views. This can, however, be avoided if we implement our own version of all the Scikit-learn's functions our algorithms are dependent upon.

6 Conclusion

Using PyKokkos, we implemented the Gaussian Naive Bayes and Logistic Regression algorithms to test the feasibility of using PyKokkos for the implementation of machine learning algorithms altogether. For common operations on views, we saw that PyKokkos, compared to NumPy, generally performs very well when the size of the inputs is large. The performance, however, is comparable to that of NumPy for smaller inputs. We then implemented the Scikit-learn's version of the Gaussian Naive Bayes and Logistic Regression using PyKokkos and compared the performance of our implementation with that of Scikit-learn's versions. We saw that PyKokkos version of the algorithms performed poorly because of PyKokkos lack of support for certain reduction operators, and addition of overhead for using Scikit-learn's validation functions.

This work has shown that PyKokkos is a powerful framework which can be used to code and test machine learning algorithms in different execution spaces with performance comparable to C++ Kokkos equivalent code, without worrying about portability.

All the functions and machine learning algorithms implemented as part of this report are publicly available at https://github.com/kokkos/pykokkos, the official PyKokkos repository.

7 Appendix

In this section, tables 5 and 6 show the *median* time of execution for the functions implemented using PyKokkos and their NumPy counterparts. Table 7, on the other hand, shows the *median* time of execution for PyKokkos's and Scikit-learn's implementation of the machine learning algorithms.

Table 5: Performance comparison against NumPy (median [ms]).

	10^1	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
Var *(2D view of shape[x, 10])*						
pk	0.17	0.17	0.17	0.29	1.2	37.0
np	0.037	0.053	0.11	0.77	8.5	96.0
Mean *(2D view of shape[x, 10])*						
pk	0.099	0.1	0.095	0.11	0.65	12.0
np	0.016	0.02	0.046	0.3	2.8	28.0
Sum *(2D view of shape[x, 10])*						
pk	0.075	0.13	0.077	0.1	0.7	9.8
np	0.0066	0.0083	0.022	0.17	1.6	16.0
Divide *(2D view of shape[x, 10])*						
pk	0.12	0.11	0.14	0.33	1.8	24.0
np	0.0034	0.0052	0.018	0.11	1.3	26.0
Power *(2D view of shape[x, 10])*						
pk	2.1	0.13	0.17	4.9	3.6	32.0
np	0.0021	0.0051	0.034	0.29	3.2	37.0

Table 5: cont.

	$\mathbf{10^1}$	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^4}$	$\mathbf{10^5}$	$\mathbf{10^6}$
Log *(1D view of shape[x])*						
pk	23.0	26.0	30.0	28.0	35.0	52.0
np	0.0018	0.0043	0.031	0.3	2.8	17.0
Exp *(1D view of shape[x])*						
pk	0.079	0.08	0.083	0.13	0.42	2.8
np	0.0017	0.0036	0.023	0.16	1.5	10.0
Logical Not *(1D view of shape[x])*						
pk	0.076	0.076	0.078	0.082	0.12	0.48
np	0.0013	0.0013	0.0017	0.0044	0.028	0.45
Multiply *(1D view of shape[x])*						
pk	0.11	0.11	0.11	0.14	0.32	1.6
np	0.0015	0.0015	0.0022	0.0069	0.048	0.83
Index *(viewA shape[x], viewB shape[x])*						
pk	0.079	0.076	0.088	0.1	0.3	4.0
np	0.00036	0.00044	0.0017	0.014	0.13	1.7
Transpose *(2D view of shape[x, 10])*						
pk	0.084	0.082	0.11	0.24	3.2	25.0
np	0.0017	0.0017	0.0015	0.0015	0.0011	0.0012
All *(1D view of shape[x])*						
pk	0.019	0.04	0.26	2.6	20.0	200.0
np	0.026	0.024	0.024	0.033	0.088	0.87

Table 5: cont.

	$\mathbf{10^1}$	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^4}$	$\mathbf{10^5}$	$\mathbf{10^6}$
Add *(viewA shape[x, 10], viewB shape[10])*						
pk	0.079	0.084	0.13	0.25	2.0	15.0
np	0.0021	0.0039	0.014	0.13	1.7	24.0
In1d *(viewA shape[x], viewB shape[x])*						
pk	0.11	0.11	0.4	11.0	620.0	44000.0
np	0.037	0.058	0.088	0.49	12.0	-
Zeros *(2D view of shape[x, 10])*						
pk	0.028	0.028	0.028	0.031	0.16	16.0
np	0.00054	0.00098	0.0028	0.023	0.29	0.062

Table 6: `TeamPolicy` performance comparison against NumPy (median [ms]).

	$\mathbf{10^1}$	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^4}$	$\mathbf{10^5}$	$\mathbf{10^6}$
Var *(2D view of shape[x, 10])*						
pk	0.17	0.20	0.22	0.29	2.7	22.0
np	0.037	0.043	0.091	0.71	7.7	94.0
Mean *(2D view of shape[x, 10])*						
pk	0.09	0.14	0.11	0.22	0.71	8.7
np	0.026	0.028	0.048	0.24	2.4	24.0
Sum *(2D view of shape[x, 10])*						
pk	0.063	0.1	0.11	0.14	0.63	8.4.0
np	0.018	0.02	0.031	0.14	1.6	18.0

Table 6: cont.

	10^1	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
Divide *(2D view of shape[x, 10])*						
pk	0.17	0.17	0.2	0.69	5.8	49.0
np	0.015	0.016	0.025	0.11	1.4	26.0
Power *(2D view of shape[x, 10])*						
pk	0.16	0.18	0.31	0.87	7.6	43.0
np	0.011	0.013	0.044	0.3	2.3	34.0

Table 7: Algorithms performance (median [ms]).

	PyKokkos	**Scikit-learn**
Gaussian Naive Bayes	2.8	0.68
Logistic Regression	26.0	19.0

References

[1] Edgar Anderson. "The Species Problem in Iris". In: *Annals of the Missouri Botanical Garden* 23.3 (Sept. 1936), p. 457. ISSN: 00266493. DOI: 10.2307/2394164. URL: https://www.jstor.org/stable/2394164?origin=crossref (visited on 11/30/2022).

[2] R. A. Fisher. "The Use of Multiple Measurements In Taxonomic Problems". en. In: *Annals of Eugenics* 7.2 (Sept. 1936), pp. 179–188. ISSN: 20501420. DOI: 10.1111/j.1469-1809.1936.tb02137.x. URL: https://onlinelibrary.wiley.com/doi/10.1111/j.1469-1809.1936.tb02137.x (visited on 11/30/2022).

[3] P. McCullagh and J. A. Nelder. *Generalized Linear Models*. en. Boston, MA: Springer US, 1989. ISBN: 9780412317606. DOI: 10.1007/978-1-4899-3242-6. URL: http://link.springer.com/10.1007/978-1-4899-3242-6 (visited on 11/30/2022).

[4] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[5] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. "Cython: The best of both worlds". In: *Computing in Science & Engineering* 13.2 (2011), pp. 31–39.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E.

Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[7] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of Parallel and Distributed Computing* 74.12 (July 2014). DOI: 10.1016/j.jpdc.2014.07.003. URL: https://www.osti.gov/biblio/1106586.

[8] G. Bonaccorso. *Machine Learning Algorithms*. Packt Publishing, 2017, pp. 120–132. ISBN: 9781785884511. URL: https://books.google.com/books?id=%5C_-ZDDwAAQBAJ.

[9] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[10] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 10.2.89*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[11] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. “A performance portability framework for Python”. en. In: *Proceedings of*

the ACM International Conference on Supercomputing. Virtual Event USA: ACM, June 2021, pp. 467–478. ISBN: 9781450383356. DOI: 10.1145/3447818.3460376. URL: https://dl.acm.org/doi/10.1145/3447818.3460376 (visited on 11/30/2022).

[12] Nader Al Awar, Neil Mehta, Steven Zhu, George Biros, and Milos Gligoric. "PyKokkos: performance portable kernels in Python". en. In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 164–167. ISBN: 9781450392235. DOI: 10.1145/3510454.3516827. URL: https://dl.acm.org/doi/10.1145/3510454.3516827 (visited on 11/30/2022).

[13] AMD. *HIP: C++ Heterogeneous-Compute Interface for Portability*. URL: https://github.com/ROCm-Developer-Tools/HIP.

[14] Kokkos. *Range Policy*. URL: https://kokkos.github.io/kokkos-core-wiki/API/core/policies/RangePolicy.html.

[15] Kokkos. *Team Policy*. URL: https://kokkos.github.io/kokkos-core-wiki/API/core/policies/TeamPolicy.html.

[16] Scikit-learn. *Gaussian Naive Bayes*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html.

[17] Scikit-learn. *Validation.py: check_is_fitted*. URL: https://github.com/scikit-learn/scikit-learn/blob/f3f51f9b611bf873bd5836748647221480071a87/sklearn/utils/validation.py#L1276.

Vita

Umair Shahzad is currently a Software Engineer at Educative Inc. (Seattle, WA). In this position he designs and develops scalable distributed services to make coding more accessible. Umair received his undergraduate degree from Lahore University of Management Sciences in Computer Science, with a focus on software engineering and systems. He currently lives in Seattle, WA.

Permanent Address: 9-K, Phase I, DHA, Lahore, Pakistan