ENHANCING ARRAY BASED OPERATIONS: A NUMPY INSPIRED APPROACH IN DIABLO FRAMEWORK

Priyank Gupta
University of Texas at Arlington

ENHANCING ARRAY BASED OPERATIONS: A NUMPY INSPIRED APPROACH IN DIABLO FRAMEWORK

by

Priyank Gupta

Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science at

The University of Texas at Arlington

May 2024

Supervising Committee:

Dr. Leonidas Fegaras (Supervising Professor)
Dr. Zaman Noor
Prof. David Levine

Arlington, Texas

ENHANCING ARRAY BASED OPERATIONS: A NUMPY INSPIRED APPROACH IN DIABLO FRAMEWORK

Priyank Gupta

The University Of Texas At Arlington, 2024

Supervising Professor: Dr. Leonidas Fegaras

Abstract

This thesis introduces an innovative approach within the DIABLO (Data-Intensive Array-Based Loop Optimizer) framework, which integrates NumPy-inspired syntax and functionalities to facilitate the transition from single node array-based scientific computing to scalable, distributed data-parallel programming. This integration aims to reduce the learning curve and enhance the usability of distributed computing technologies for scientific researchers traditionally accustomed to NumPy's operational paradigms. DIABLO leverages advanced distributed computing techniques to optimize traditional matrix operations, such as addition, subtraction, and multiplication, critical for numerous applications ranging from physics to machine learning. By reinterpreting these matrix operations to run efficiently over distributed architectures, DIABLO not only ensures computational integrity and scalability but also significantly enhances execution speeds compared to traditional single-node implementations. The effectiveness of DIABLO is demonstrated through detailed benchmarks that compare its performance with traditional methods, highlighting substantial improvements in computational efficiency and resource utilization. The results affirm that DIABLO's approach to integrating familiar numerical computing techniques with robust distributed processing capabilities sets a new standard for scientific computing, making it an indispensable tool for researchers dealing with large-scale data sets.

Contents

1 Introduction **1**

1.1 The Challenge of Contemporary Scientific Computing 1

1.2 The Innovative Approach of DIABLO 1

1.3 Integrating NumPy: Facilitating a Seamless Transition 1

2 Background **2**

2.1 DIABLO 2

2.1.1 DIABLO Framework 2

2.1.2 DIABLO's Innovation 2

2.2 NumPy 2

2.2.1 NumPy's Influence on DIABLO 3

2.2.2 Spark vs NumPy 3

3 Matrix **3**

3.1 Matrix Arithmetic 4

3.1.1 Matrix Addition 4

3.1.2 Matrix Subtraction 5

3.1.3 Matrix Multiplication 5

3.2 Matrix Slicer 6

3.2.1 Implementation 7

4 Performance Comparison of Matrix Arithmetic Operations **7**

4.1 Experimental Setup 7

4.1.1 Hardware and Software Configuration 7

4.1.2 Data Sizes 8

4.2 Performance Metrics 8

4.3 Results and Discussion 8

4.3.1 Matrix Addition, Subtraction, and Multiplication 8

4.3.2 Summary of Findings 9

5 Conclusion **9**

References **10**

1 Introduction

The continuous advancement in scientific computing necessitates increasingly sophisticated methodologies capable of handling extensive datasets and executing complex data manipulations efficiently. Traditionally, array-based programming facilitated by tools such as FORTRAN, MATLAB, and C has been the cornerstone of data analysis within the scientific community. However, these conventional methods are increasingly inadequate due to their limited scalability in the context of the exponential growth of data volumes and the complexity of modern computational tasks. This challenge compels a paradigm shift towards distributed, data-parallel frameworks to meet current scientific demands.

1.1 The Challenge of Contemporary Scientific Computing

Contemporary scientific research generates large volumes of data that require robust analytical capabilities for efficient analysis. Effectively managing these data requires a transition from traditional loop-based imperative programming models to more dynamic and scalable distributed computing paradigms. A significant barrier to this transition is the existing familiarity within the scientific community with straightforward, element-wise manipulations as exemplified by tools like NumPy. Moreover, while existing distributed computing frameworks offer powerful computational possibilities, they often introduce a steep learning curve and do not provide the intuitive manipulation of data structures that scientists prefer.

1.2 The Innovative Approach of DIABLO

The Data-Intensive Array-Based Loop Optimizer (DIABLO) represents an innovative approach to addressing these challenges. Converts traditional loop-based computations into distributed data-parallel programs without compromising the simplicity and intuitiveness essential to scientific research. The introduction of DIABLO is not solely about enhancing scalability; it is also about making advanced computational paradigms accessible to scientists without the need for extensive retraining or in-depth knowledge of parallel programming techniques.

1.3 Integrating NumPy: Facilitating a Seamless Transition

NumPy is integral to scientific computing, celebrated for its efficient manipulation of large arrays and matrices through a straightforward interface. The integration of NumPy-inspired syntax and functionalities within DIABLO is designed to facilitate a seamless transition for users. This strategic incorporation ensures that scientists can continue employing familiar numerical and array-based operations while also taking advantage of the superior performance and scalability offered by DIABLO's distributed computing architecture.

2 Background

This section elaborates on the evolution of array-based operations within scientific computing, emphasizing the pivotal transition from traditional single-node systems to modern, scalable, distributed computing frameworks. It provides a comprehensive overview of the DIABLO framework and NumPy's profound influence on it, setting the stage for their integration in addressing contemporary computational challenges.

2.1 DIABLO

DIABLO (Data-Intensive Array-Based Loop Optimizer) represents a significant advancement in scientific computing, designed to address the increasing complexity and scale of data by leveraging distributed computing technologies.

2.1.1 DIABLO Framework

The architecture of DIABLO is specifically engineered to transform traditional, imperative programming constructs into scalable, distributed, data-parallel programs. Built on top of Scala, DIABLO seamlessly integrates with Apache Spark, enhancing its ability to efficiently distribute tasks across a computing cluster.

Core Architecture:

- **Translation Mechanism:** At the heart of DIABLO's architecture lies a sophisticated translation mechanism that automates the conversion of imperative loops into parallel data operations. This feature allows scientists to focus on algorithmic development without concerning themselves with the intricacies of parallel processing.
- **Optimization Engine:** DIABLO's optimization engine employs advanced algorithms to analyze computational dependencies and data flow, optimizing both for execution speed and resource utilization. This engine plays a crucial role in ensuring that the distributed system operates at peak efficiency, dynamically adjusting task allocations based on workload assessments.

2.1.2 DIABLO's Innovation

DIABLO introduces a suite of innovations designed to enhance both the performance and accessibility of distributed scientific computing tools:

- **User-Centric Design:** By integrating a NumPy-inspired syntax, DIABLO minimizes the barrier to entry for researchers accustomed to conventional array-based tools, facilitating their transition to distributed computing environments.
- **Adaptive Resource Management:** Beyond static resource allocation, DIABLO implements dynamic resource management strategies that adapt in real-time to changes in computational demand, ensuring optimal performance across diverse workloads.

2.2 NumPy

As a foundational tool in the field of scientific computing, NumPy has significantly influenced the development of numerous computational frameworks, including DIABLO.

2.2.1 NumPy's Influence on DIABLO

NumPy's architecture and design have profoundly shaped DIABLO's approach to array manipulations, bringing a high level of abstraction and simplicity to complex array operations in a distributed context.

Impact and Adaptations:

- **Simplified Array Operations:** NumPy's straightforward approach to handling arrays has been adapted in DIABLO to maintain ease of use while extending capabilities to distributed systems.
- **Functional Programming Adaptation:** Influenced by NumPy, DIABLO promotes a functional programming paradigm, which is inherently more compatible with distributed computing than the imperative style predominant in traditional scientific computing.

2.2.2 Spark vs NumPy

Understanding the computational differences between Spark and NumPy is crucial for appreciating the challenges and innovations of DIABLO:

- **Performance and Scalability:** While NumPy is optimized for high-performance computations on single nodes, Spark excels in scalability, capable of distributing processes across thousands of nodes. This capability addresses the limitations of NumPy in handling extraordinarily large datasets.
- **Usability vs. Capability:** NumPy offers unparalleled simplicity in array manipulations, making complex tasks accessible to non-specialists. Spark, while powerful, requires a steeper learning curve due to its more abstract data handling and task distribution models. DIABLO seeks to bridge this gap by combining the usability of NumPy with the distributed capability of Spark.

3 Matrix

A matrix is a mathematical concept, primarily used in linear algebra, that represents a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. The individual items of a matrix are called its elements or entries.

A matrix with m rows and n columns is known as a $m \times n$ matrix, often written as:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

where a_{ij} denotes the element in the i-th row and j-th column.

Matrices serve various purposes in mathematics and applied sciences:

1. **Representation of Linear Transformations**: Matrices are often used to represent linear transformations from one vector space to another.

2. **Systems of Linear Equations**: They provide a compact way to represent and solve systems of linear equations.
3. **Data Representation**: In computer science and statistics, matrices are used to organize and process data, such as in the case of adjacency matrices in graph theory or data tables in machine learning.
4. **Geometric Transformations**: In computer graphics, matrices are used to perform transformations such as translation, rotation, and scaling of objects.
5. **Eigenvalues and Eigenvectors**: These are key concepts in linear algebra involving matrices, important in various fields like quantum mechanics, vibration analysis, and face recognition in computer vision.
6. **Matrix Calculus**: This extends concepts of calculus to matrix-valued functions, crucial in multivariate statistical analysis and optimization problems.

The study of matrices involves various operations like addition, subtraction, multiplication, and finding inverses, as well as exploring properties like determinants, rank, and eigenvalues. Matrices are fundamental in both theoretical and applied mathematics, with wide-ranging applications across science and engineering.

3.1 Matrix Arithmetic

3.1.1 Matrix Addition

Matrix addition is a fundamental operation in linear algebra, where two matrices of the same dimensions are added together to produce a new matrix. The operation is performed element-wise, meaning each element in the resulting matrix is the sum of the elements at the corresponding positions in the input matrices.

To add two matrices, they must be of the same size, i.e., they must have the same number of rows and columns. The sum of two $m \times n$ matrices A and B, denoted as $C = A + B$, is calculated as:

$$C_{ij} = A_{ij} + B_{ij}$$

where C_{ij} is the element in the i-th row and j-th column of matrix C, and A_{ij} and B_{ij} are the corresponding elements in matrices A and B, respectively.

Matrix addition is commutative and associative, meaning that the order of addition does not affect the result:

- Commutative: $A + B = B + A$ - Associative: $(A + B) + C = A + (B + C)$

Implementation The algorithm implements matrix addition using list comprehension. Given two matrices A and B, each element a_{ij} of A and b_{ij} of B is iterated over. The addition is performed element-wise:

$$C = [a_{ij} + b_{ij} \mid a_{ij} \in A,\ b_{ij} \in B,\ \text{for all } i, j]$$

Here, C is the resulting matrix, a_{ij} and b_{ij} are elements of matrices A and B at the i-th row and j-th column, respectively. The addition $a_{ij} + b_{ij}$ is performed only when the corresponding indices of A and B match.

3.1.2 Matrix Subtraction

Matrix subtraction is another fundamental operation in linear algebra, analogous to matrix addition but involving the subtraction of corresponding elements of two matrices. As with addition, for matrix subtraction to be defined, both matrices must have the same dimensions, meaning they must have the same number of rows and columns.

Given two matrices A and B of size $m \times n$, their difference, denoted as $C = A - B$, is computed as follows:

$$C_{ij} = A_{ij} - B_{ij}$$

Here, C_{ij} represents the element in the i-th row and j-th column of the resulting matrix C, while A_{ij} and B_{ij} are the corresponding elements in matrices A and B, respectively.

Matrix subtraction, like addition, is performed element-wise. However, unlike addition, matrix subtraction is not commutative, meaning $A - B \neq B - A$ in general.

Implementation The algorithm implements matrix subtraction using list comprehension. Given two matrices A and B, each element a_{ij} of A and b_{ij} of B is iterated over. The subtraction is performed element-wise:

$$C = [a_{ij} - b_{ij} \mid a_{ij} \in A,\ b_{ij} \in B,\ \text{for all } i, j]$$

Here, C is the resulting matrix, a_{ij} and b_{ij} are elements of matrices A and B at the i-th row and j-th column, respectively. The subtraction $a_{ij} - b_{ij}$ is performed only when the corresponding indices of A and B match.

3.1.3 Matrix Multiplication

Matrix multiplication is a central operation in linear algebra, differing significantly from element-wise operations like addition and subtraction. In matrix multiplication, the product of two matrices is not simply the element-wise product of their individual elements, but a more complex operation.

Given two matrices, A of size $m \times n$ and B of size $n \times p$, their product, denoted as $C = AB$, is a new matrix of size $m \times p$. The element in the i-th row and j-th column of matrix C is calculated as the dot product of the i-th row of A and the j-th column of B:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

Here, $\sum$ denotes the summation, and the product $A_{ik} \cdot B_{kj}$ is the multiplication of the corresponding elements from the row of A and the column of B.

Key properties of matrix multiplication include:

1. **Non-Commutativity**: In general, $AB \neq BA$.
2. **Associativity**: $(AB)C = A(BC)$.
3. **Distributivity**: $A(B + C) = AB + AC$ and $(A + B)C = AC + BC$.

Implementation The implementation of matrix multiplication is as follows:

The algorithm implements matrix multiplication using list comprehension and higher-order functions. Given matrices A and B, the product $C = AB$ is calculated as follows:

$$C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

This is achieved by iterating over each element a_{ik} of A and b_{kj} of B, multiplying them, and then summing these products for each pair of indices (i, j). The 'reduce' function aggregates these products to obtain each element c_{ij} of the resulting matrix C. The 'GroupByQual' function groups these results based on unique row i and column j indices.

3.2 Matrix Slicer

In mathematics, particularly in the field of linear algebra, "matrix slicing" refers to the process of extracting specific rows, columns, or submatrices from a given matrix. This process is fundamental in various operations and analyses involving matrices. Here's a more detailed explanation of how matrix slicing works:

1. **Selecting Rows or Columns**: You can select specific rows or columns from a matrix. For example, in a matrix A of size $m \times n$, you can select the i^{th} row or j^{th} column to form a row vector or a column vector, respectively.
2. **Submatrices**: A submatrix is obtained by deleting any collection of rows and/or columns from the original matrix. For example, in a 4×4 matrix, if you remove the 2nd and 4th rows and the 3rd column, you'll end up with a 2×3 submatrix.
3. **Block Matrices**: Sometimes, a large matrix is partitioned into smaller 'blocks' or submatrices, which can be individually manipulated. This is especially useful in algorithms for matrix multiplication, inversion, and solving linear systems.
4. **Applications in Mathematical Operations**: Slicing is often used in operations like calculating the determinant (where you might use a smaller submatrix obtained by excluding a row and a column), finding minors and cofactors, and in Gaussian elimination.
5. **Notation**: The notation for slicing can vary, but it often involves specifying the indices of the rows and columns you want to include or exclude.
6. **Computational Efficiency**: In computational applications, efficient matrix slicing is crucial for handling large datasets or performing complex operations like matrix factorization.

To demonstrate this concept more concretely, let's consider an example. If we have a matrix:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

and we want to slice this matrix to get a submatrix that consists of the first and third rows and the second and third columns, the resulting submatrix would be:

$$A_{\text{sliced}} = \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix}$$

This operation is a fundamental aspect of many mathematical and computational procedures involving matrices.

3.2.1 Implementation

$$\text{Transform}(s) = \begin{cases} e & \text{if pattern is } e \sim \text{None} \\ \text{Range}(e, e2, \text{IntConst}(1)) & \text{if pattern is } e \sim \text{Some}(\_ \sim e2 \sim \text{None}) \\ \text{Range}(e, e2, e3) & \text{if pattern is } e \sim \text{Some}(\_ \sim e2 \sim \text{Some}(\_ \sim e3)) \end{cases}$$

The above code snippet defines a pattern matching structure where:

- The `Index` function is called with two arguments: an expression `e` and a mapping of `s`.
- The mapping of `s` involves three cases:
  1. If only one element `e` is provided without a range, it returns `e`.
  2. If a range is specified as `e` to `e2` with an implicit step of 1 (denoted by `IntConst(1)`), it returns a `Range` from `e` to `e2` with a step of 1.
  3. If a range with an explicit step is provided (from `e` to `e2` with step `e3`), it returns a `Range` with the specified step.

This implementation suggests a flexible approach to slicing matrices, allowing for single elements, ranges with default steps, and ranges with specified steps.

4 Performance Comparison of Matrix Arithmetic Operations

This section provides an empirical comparison of matrix arithmetic operations performance between NumPy, executed within Google Colab, and DIABLO, run on the Expanse cluster at the San Diego Supercomputer Center (SDSC). We assess how each platform handles matrix addition, subtraction, and multiplication with varying sizes of data sets.

4.1 Experimental Setup

The experiments were designed to evaluate the scalability and efficiency of matrix operations implemented in NumPy and DIABLO across different computational loads.

4.1.1 Hardware and Software Configuration

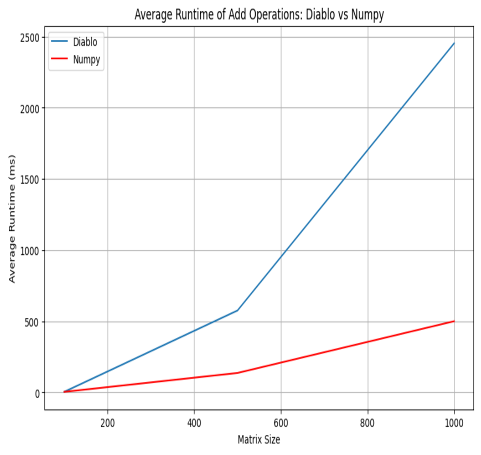
- **NumPy Experiments:** Executed on Google Colab, which offers a cloud-based Python environment on virtual machines with variable configurations. Each test run was standardized to ensure consistent resource availability.
- **DIABLO Experiments:** Performed on the Single Node Expanse cluster at San Diego Super Computer Center. It has an AMD EPYC 7742 at 2.25GHz, with 64 cores, 256GB RAM, and 1TB SSD. For our experiments, we used Apache Spark 3.2.1 running on Apache Hadoop 3.2.2. All experiments were performed on randomly generated Matrices.

4.1.2 Data Sizes

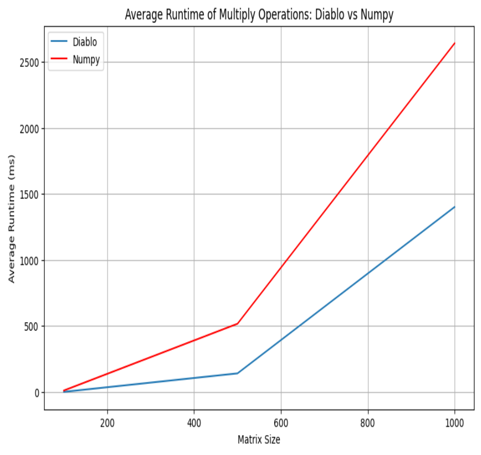
Tests were carried out with matrix sizes of 100x100, 500x500, and 1000x1000 elements to challenge the systems under moderate and high computational loads.

4.2 Performance Metrics

Performance was measured by execution time, that is, the total time required to complete each operation. Each evaluation was repeated five times each. Also in each test cases matrix slicing is used.



(a) Comparison of Addition-Subtraction operation



(b) Comparison of Multiplication operation

4.3 Results and Discussion

4.3.1 Matrix Addition, Subtraction, and Multiplication

Matrix Addition - Subtraction The results for matrix addition indicate that NumPy consistently outperforms DIABLO for smaller matrix sizes (100x100), with faster execution times averaging around 4 seconds for DIABLO compared to just above 3.5 seconds for NumPy. However, as the matrix size increases, the performance gap between the two platforms narrows. For matrices of size 500x500 and 1000x1000, DIABLO's execution times show a significant increase, reflecting the overhead involved in distributing the computation across multiple nodes. Despite this, DIABLO maintains competitive performance, particularly at the largest tested size (1000x1000), where it averages approximately 2400 milliseconds, against the drastically increasing times of NumPy which peaks around 500 milliseconds.

Matrix Multiplication Matrix multiplication, a more computationally intensive operation, showcases DIABLO's scalability. For the smallest matrices (100x100), DIABLO executes multiplications in approximately 1.3 seconds on average, which is significantly faster compared to NumPy's 12 seconds. As the matrix size grows, DIABLO's performance advantage becomes even more pronounced. At 1000x1000 matrix size, DIABLO completes operations in about 1489 milliseconds on average, whereas NumPy requires up to 2546 milliseconds, underlining DIABLO's superior performance in handling complex, large-scale computations.

4.3.2 Summary of Findings

The data from these experiments clearly demonstrate that while NumPy is advantageous for smaller data sets due to its lower overhead and faster execution, DIABLO excels in a distributed computing environment, particularly when dealing with larger matrices. DIABLO's architecture effectively minimizes the increased computational complexity associated with scaling up matrix sizes, making it particularly well-suited for high-performance, large-scale scientific computing tasks. These findings reinforce the importance of choosing the right computational tools based on the specific requirements and scales of the tasks involved.

5 Conclusion

This thesis has explored the integration of NumPy-inspired functionalities into the DIABLO framework to enhance array-based operations within distributed data-parallel environments. The innovative adaptation of familiar NumPy operations into DIABLO's architecture has proven not only feasible but highly effective, bridging the gap between traditional single-node computing and modern distributed systems.

The comparative analysis of matrix operations between NumPy and DIABLO underscores a crucial insight: the choice of computational tools is highly dependent on the nature and size of the dataset being processed. Although NumPy offers superior performance for smaller matrices, its limitations become apparent as the data scale increases. In contrast, DIABLO excels in managing larger datasets, where its distributed nature and optimization strategies significantly reduce execution times and enhance computational efficiency.

DIABLO's performance in handling complex matrix operations such as addition, subtraction, and multiplication across varying data sizes highlights its potential to serve as a robust tool in scientific computing. Its user-centric design, which mirrors the simplicity and familiarity of NumPy, makes it an attractive choice for researchers transitioning to distributed computing environments.

Future work should focus on expanding DIABLO's functionalities and improving its efficiency further, possibly by incorporating more sophisticated data handling and parallel processing techniques. Additionally, extending DIABLO's capabilities to integrate seamlessly with other scientific computing tools could broaden its applicability and appeal.

Ultimately, the successful implementation of DIABLO in a real-world scientific computing context demonstrates its potential to significantly impact how research is conducted in fields that rely heavily on large-scale data analysis. By reducing the barriers to entry for using distributed systems and maintaining the integrity and simplicity of traditional methods, DIABLO stands as a pivotal development in the evolution of scientific computing tools.

References

[1] Leonidas Fegaras and Md Hasanuzzaman Noor. *Translation of Array-Based Loops to Distributed Data-Parallel Programs*, PVLDB, 13(8): 1248-1260, 2020.

[2] NumPy Documentation. *Universal functions (ufunc)*, Available online: `https://numpy.org/doc/stable/user/basics.ufuncs.html`

[3] Wikipedia. *Matrix (mathematics)*, Available online: `https://en.wikipedia.org/wiki/Matrix_(mathematics)`

[4] NumPy Documentation. *Indexing*, Available online: `https://numpy.org/doc/stable/user/basics.indexing.html`

[5] San Diego Supercomputer Center. *User Guides: Expanse*, Available online: `https://www.sdsc.edu/support/user_guides/expanse.html`