VecLang: A Domain-Specific Language for Vector Operations KNN Search

Abylay Amanbayev [1]

Abstract

This paper presents VecLang, a domain-specific programming language designed to handle vector and matrix operations with a Python-like syntax and LLVM-based compilation. VecLang focuses on enabling efficient computations for tasks such as vector similarity measures, nearest neighbor searches, and matrix-to-matrix distance calculations. Although existing systems are widely used for similarity search, they do not directly provide a matrix-to-matrix distance calculation interface. VecLang addresses this gap by introducing a language construct that supports such computations. We discuss VecLang's design, LLVM integration, and the mathematical formulation of matrix-to-matrix distance. Preliminary implementation results are presented, and future directions, including GPU acceleration and advanced index strategies, are outlined.

1. Introduction

Efficient manipulation of vectors and matrices is integral to a broad range of applications in machine learning, data analysis, and scientific computing. Current libraries, such as FAISS (Douze et al., 2024), ANNOY (ann) as well as vector databases, such as Milvus (Wang et al., 2021), are designed to handle these computationally intensive similarity search operations. These systems provide efficient algorithms for k-NN search, often optimized for GPU acceleration and distributed systems. However, they do not offer built-in matrix-to-matrix distance computation functionality, a feature critical for some advanced analytical workflows.

VecLang (vec) addresses this shortcoming by introducing a specialized command for matrix-to-matrix computations, while maintaining support for common operations and nearest neighbor searches.

VecLang's key contributions are:

- A Python-like syntax tailored for vector and matrix operations.
- Built-in functions for similarity and KNN computations, including `similarity`, `knn_bf`, and `knn_bf_batch` for matrix-to-matrix calculations.
- Integration with LLVM IR (Lattner & Adve, 2004) to produce optimized, low-level code, thus ensuring both performance and platform independence.

2. Language Design

VecLang focuses on providing high-level constructs for defining, manipulating, and analyzing vectors and matrices.

2.1. Data Structures

Vector: A one-dimensional array of real numbers. Matrix: A two-dimensional array of real numbers.

2.2. Operations

2.3. Data Structures

- **Vector**: A 1D array of real numbers.
- **Matrix**: A 2D array of real numbers.

2.4. Operations

Arithmetic:

- Addition ($+$), scalar multiplication ($*$), and dot product ($@$).

Advanced Functions:

- `similarity(v1, v2, method="...")`: Computes similarity (cosine or Euclidean distance) between vectors `v1` and `v2`.
- `knn_bf(v, M, k, method="...")`: Finds the k-nearest neighbors of vector `v` in matrix `M`.
- `knn_bf_batch(M1, M2, method="...")`: Computes matrix-to-matrix distances between `M1` and `M2`, returning both aggregate statistics and mapping results.

2.5. Example Code

```
# Define vectors and matrices
v1 = vector([1, 2, 3])
m1 = matrix([[1, 2, 3], [4, 5, 6]])
m2 = matrix([[7, 8, 9], [10, 11, 12]])

# Compute vector similarity
cos_sim = similarity(v1, \
    vector([4, 5, 6]), method="cosine")

# Perform KNN search
knn_result = knn_bf(v1, m1, 2, \
    method="euclidean")

# Compute matrix-to-matrix distance
batch_result = knn_bf_batch(m1, m2, \
    method="euclidean")
```

3. Mathematical Formulation of Matrix-to-Matrix Distance

Consider two matrices:

$$M_1 \in \mathbb{R}^{m \times n}, \quad M_2 \in \mathbb{R}^{p \times n}.$$

Each row of M_1 and M_2 is a vector in $\mathbb{R}^n$. Let

$$\mathbf{u}_i \text{ be the } i\text{-th row of } M_1, \quad \mathbf{v}_j \text{ be the } j\text{-th row of } M_2.$$

For Euclidean distance, the distance between $\mathbf{u}_i$ and $\mathbf{v}_j$ is:

$$d(\mathbf{u}_i, \mathbf{v}_j) = \sqrt{\sum_{k=1}^{n} (\mathbf{u}_i[k] - \mathbf{v}_j[k])^2}.$$

The result is an $m \times p$ distance matrix D:

$$D[i, j] = d(\mathbf{u}_i, \mathbf{v}_j) \quad \forall i \in \{1, \ldots, m\}, j \in \{1, \ldots, p\}.$$

4. Implementation

- **Lexer**: Converts source code into tokens (identifiers, numbers, operators).
- **Parser**: Transforms tokens into an Abstract Syntax Tree (AST).

We choose an AST representation because it provides a clear, structured, and language-agnostic intermediate form. An AST simplifies subsequent optimization and code generation steps compared to direct translation or other intermediate forms.

4.1. Interpreter and LLVM Integration

- **Interpreter**: Initially, an interpreter can directly evaluate the AST. However, interpretation alone can be slow for large datasets.
- **LLVM Integration**: To achieve performance, we generate LLVM IR from the AST rather than relying solely on interpretation. LLVM was chosen because:
  - It provides a well-documented, stable, and widely adopted Intermediate Representation (IR).
  - Offers rich optimizations (loop unrolling, vectorization) and supports multiple architectures.
  - Allows mixing low-level runtime functions (like `knn_bf` and `knn_bf_batch`) with high-level constructs seamlessly.

4.2. Interpreter and LLVM Integration

Runtime functions implemented in C handle low-level computations. For example:

```
void* knn_bf(void* vector, void* matrix,
    int k, int method);
void* knn_bf_batch(void* matrix1, void*
    matrix2, int method);
double knn_bf_batch_get_min_sum(void*
    batch_result);
void* knn_bf_batch_get_mapping(void*
    batch_result);
```

These are linked with LLVM IR, providing a clean separation between language syntax and performance-critical code.

5. Performance Evaluation:

To illustrate VecLang's IR generation and the potential for optimizations, we show representative LLVM IR snippets.

Vector Addition:

```
%v1 = call i8* @vector_create(double*
    %data1, i32 %size1)
%v2 = call i8* @vector_create(double*
    %data2, i32 %size2)
%v3 = call i8* @vector_add(i8* %v1,
    i8* %v2)
```

KNN Function Call:

```
%knn_result = call i8* @knn_bf(i8*
    %vector, i8* %matrix, i32 %k,
    i32 %method)
```

Batch KNN (Matrix-to-Matrix) Function Call:

```
%batch_result = call i8*
    @knn_bf_batch(i8* %matrix1,
    i8* %matrix2, i32 %method)
%min_sum = call double
    @knn_bf_batch_get_min_sum(i8*
    %batch_result)
%mapping = call i8*
    @knn_bf_batch_get_mapping(i8*
    %batch_result)
```

By generating LLVM IR, VecLang paves the way for high-level optimizations and hardware acceleration. While we have demonstrated successful computation of matrix-to-matrix distances, detailed benchmarking and performance comparisons with other systems are ongoing. Future evaluations will measure speedups and memory usage against established libraries.

6. Conclusion

VecLang enhances existing vector seach and k-NN-toolkits by providing a domain-specific language with native support for matrix-to-matrix distance calculations. By integrating LLVM, VecLang benefits from sophisticated optimizations and portability.

Future Work:

- GPU Acceleration: Employ CUDA or OpenCL to parallelize similarity and KNN computations.
- Index Search Utilization: Integrate advanced indexing methods, such as clustering-based indexes or graph-based approaches (e.g., Hierarchical Navigable Small World (HNSW) graphs) for efficient nearest neighbor queries.
- Computation Reuse: Implement caching and partial computation reuse strategies to improve performance in iterative workloads.

These enhancements will further expand VecLang's applicability and reduce computation times in large-scale data analysis scenarios.

References

Annoy (approximate nearest neighbors oh yeah). https://github.com/pinecone-io.

Veclang: A domain-specific language for vector and matrix operations. https://github.com/aabylay/VecLang.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. The faiss library, 2024. URL https://arxiv.org/abs/2401.08281.

Lattner, C. and Adve, V. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, 2004. doi: 10.1109/CGO.2004.1281665.

Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., Yu, K., Yuan, Y., Zou, Y., Long, J., Cai, Y., Li, Z., Zhang, Z., Mo, Y., Gu, J., Jiang, R., Wei, Y., and Xie, C. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pp. 2614–2627, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457550. URL https://doi.org/10.1145/3448016.3457550.