MLIR-based UDFs in JIT code-generated DBMS

Tomasz Marzec
tomasz.marzec@epfl.ch

Hamish Nicholson
hamish.nicholson@epfl.ch

Aunn Raza
aunn.raza@epfl.ch

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch

ABSTRACT

This report explores integrating user-defined functions (UDFs) based on Multi-Level Intermediate Representation (MLIR) into a just-in-time (JIT) code-generated DBMS, focusing on Proteus. We introduce a new type of Proteus expression, enabling the invocation of externally defined functions within the DBMS. Our implementation demonstrates improvements in code reusability and optimization potential. Case studies with MLIR-based machine learning (ML) compilers showcase the practical benefits of this integration. This work lays the groundwork for enhancing the integration of machine learning models and complex computations into DBMSs, highlighting future development potential.

KEYWORDS

DBMS, Proteus, MLIR, UDF

1 INTRODUCTION

With the abundance of different frameworks available today, the question of code reusability across these frameworks becomes increasingly pertinent. In the context of database management systems (DBMS), integrating user-defined functions (UDFs) efficiently and seamlessly is crucial for extending functionality and optimizing performance. Traditional approaches often face challenges related to compatibility, performance optimization, and maintainability when dealing with diverse frameworks and libraries.

This project aims to address these challenges by leveraging Multi-Level Intermediate Representation (MLIR), a versatile and extensible architecture designed to represent complex data structures and computations across various abstraction levels. By focusing on the Proteus DBMS, which utilizes just-in-time (JIT) compilation for generating specialized query execution code, we explore the potential of integrating MLIR to enhance UDF reusability and performance.

Proteus, known for its JIT compilation capabilities using LLVM's infrastructure, benefits from the introduction of *ExternExpression*, a new type of Proteus expression that facilitates the invocation of externally defined functions within the DBMS. This approach enables the integration of advanced MLIR-based functionalities without requiring data migration outside the system, thus maintaining efficiency and performance.

As part of our exploration, we experimented with Toy-MLIR, a simple example language provided by the MLIR framework, to understand its capabilities and integration potential. Additionally, we developed a custom MLIR compiler supporting simple dialects to test and validate the integration process. These experiments provided valuable insights into the flexibility and robustness of MLIR when used in conjunction with Proteus.

We further illustrate the practical applications and benefits of this integration through case studies involving MLIR-based machine learning (ML) compilers, such as ONNX-MLIR and Torch-MLIR. These case studies highlight how machine learning models and other complex computations can be seamlessly incorporated into the DBMS, leveraging MLIR's powerful optimization and abstraction capabilities.

In this report, we detail the implementation process, challenges encountered, and solutions developed to enable MLIR-based UDFs in Proteus. We also discuss the broader implications of our findings and propose future directions for enhancing the integration of MLIR into JIT code-generated DBMS environments. Future work will focus on improving type handling, automating the creation of wrappers for external libraries, and further optimizing the interaction between MLIR and Proteus to enhance code reusability and performance across different frameworks.

2 LLVM

LLVM (Low-Level Virtual Machine) [5] is a collection of modular and reusable compiler and toolchain technologies. Originally designed as a set of reusable compiler components, LLVM has evolved into a robust framework for developing compiler frontends and backends. LLVM's intermediate representation (LLVM IR) is central to its design, providing a common, low-level code representation that facilitates advanced optimizations and code generation for a variety of hardware targets.

2.1 MLIR

MLIR (Multi-Level Intermediate Representation) [6] is a subproject of LLVM designed to address some of the limitations of LLVM IR when dealing with problems better modeled at higher- or lower-level abstractions. One example of this is source-level analysis, which is very difficult at LLVM IR level. For this reason, many compilers have developed their own IRs created for tackling issues like library and language-specific optimizations. This is the case for many languages, including Swift, Rust, Julia, and Fortran. See Figure 1.

In a similar fashion, machine learning systems often use "ML graphs" as domain-specific abstractions.

To solve these problems, MLIR provides a flexible, extensible architecture for representing complex data structures and computation patterns across different levels of abstraction. It introduces the concept of dialects, which are customizable extensions that define specific operations, types, and transformations tailored to particular domains or architectures.

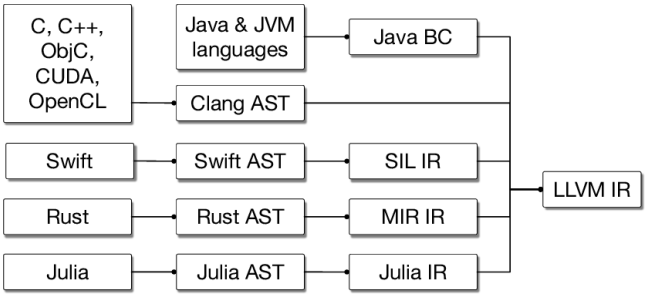

Figure 1: Compilation pipeline with mid-level language IRs [6].

2.2 Co-Compiling and Optimizing with External Functions/Operators

One of the primary motivations for MLIR is to enhance the ability to co-compile and optimize code that interacts with external functions or operators. Traditional compilers often struggle with efficiently integrating and optimizing code that relies on domain-specific libraries or hardware accelerators. MLIR's dialect system allows for the definition of custom operations and types, facilitating seamless integration and optimization of external functionalities within the compiler framework.

To see this better, consider two different MLIR programs, each one of them using custom dialects, and custom transformation passes. It's very easy to create a new MLIR program using dialects from both MLIR programs. Dialect is a very high-level concept in MLIR, and extending an MLIR module to use a new dialect is usually as simple as writing a single line of code. It's the same with transformation passes, as registering a new pass is very straightforward. An example of such merging of multiple dialects is shown later, in Section 5.4.

2.3 Migration from LLVM IR to MLIR

Even though the primary end goal of introducing MLIR is the final lowering to LLVM IR, it also makes sense to talk about "uplifting" LLVM IR to MLIR. This especially makes sense for already existing projects leveraging LLVM IR for execution, that want to enable higher-level abstraction optimizations by using the MLIR architecture.

Migrating an existing engine from LLVM IR to MLIR can be done gradually, which is advantageous for maintaining stability and performance during the transition. It would consist of several phases:

Step 1: Change the codegen to emit operations from the LLVM Dialect instead of LLVM IR.
Step 2: Introduce new dialect
Step 3: Define translation/lowering to a previously defined dialect
Step 4: Add optimization rules
Step 5: Change the codegen to also emit operations from the new dialect
Step 6: Repeat Step 2

For Step 1, the *ConvertFromLLVMIR* functionality available in MLIR might be helpful. It lifts LLVM IR to LLVM Dialect.

This phased approach allows developers to incrementally adopt MLIR's advanced features while continuing to leverage LLVM's mature infrastructure. The ultimate goal is to enable better optimizations through MLIR dialects compared to what is achievable with just LLVM IR. This gradual migration strategy ensures that the benefits of MLIR can be realized without disrupting existing workflows or sacrificing performance in the short term.

3 PROTEUS BACKGROUND

Proteus [4] is a database engine designed for today's heterogeneous environments. Proteus adapts to variable data, hardware and workloads through a combination of GPU acceleration, data virtualization, and adaptive scheduling.

Before Proteus executes a query, it traverses all the operators and generates their implementation. All those code fragments are merged into query engine implementation. Proteus leverages the LLVM infrastructure to generate this code by generating LLVM IR. The choice for Proteus to generate LLVM IR was made for several reasons, some of them being [4]:

- it is strictly-typed and less error-prone than macro-based C++
- it compiles much faster than macro-based C++ code
- LLVM offers rewrite passes such as dead code elimination that optimize the generated IR

This way, Proteus creates a specialized query engine for a query. This code can then be optimized using many sophisticated optimization phases provided by LLVM.

3.1 RelBuilder

The *RelBuilder* class in Proteus is used to build a physical query plan of operators. It's syntactic sugar for building query plans by hand in C++. Here is a simplified example of how it can be used:

```
std::vector<int> v1{4, 5, 7};
std::vector<int> v2{6, 7, 8};

auto builder = RelBuilderFactory{}
  .getBuilder()
  .scan({{/*attrName*/ "a", v1}, {"b", v2}})
  .unpack()
  .filter([](const auto &arg) {
    return expression_t { arg["a"] < 5 };
  })
  .project([](const auto &arg) {
    return expression_t { arg["b"] + 1 };
  });
```

The code in the plan is generated and compiled using the prepare method. This compiled query can then be executed:

```
builder.prepare().execute();
```

In Proteus, two main constructs represent transformations on data: operators and expressions.

3.2 Operators

Operators are the fundamental building blocks of a query execution plan, each representing a specific step in the data processing pipeline. Key operators in Proteus include:

- Scan: reads data from a source, such as a table or a file, and forms the starting point of a query plan.
- Filter: applies a predicate to remove rows that do not satisfy a specified condition, refining the dataset.
- Project: selects specific columns or computes new values from existing columns, transforming the dataset into the desired shape.
- Aggregate: computes summary statistics like sum, average, or count, providing insights from the data.
- Sort: orders rows based on the values of one or more columns, organizing the data.

3.3 Expressions

Expressions (*expression_t*) represent computations and transformations performed on a single row in the query plan. They are commonly used within the *project* operator. They provide a flexible way to define various operations, including:

- Constants: expressions can represent constant values, allowing for fixed values to be used within the query plan. These include numerical values, logical values, strings, and dates.
- Arithmetic Operations: expressions can model basic arithmetic operations such as addition, subtraction, multiplication, and division on data fields. There is also support for modulo operation, and computing max and min values.
- Predicates: expressions can be used to represent conditions or rules that filter data within the query plan. Those include comparisons of numerical values and checking for nulls.

Evaluating expressions in Proteus is accomplished through visitors [4] tasked with generating the corresponding code for each expression:

```
ProteusValue ExpressionGeneratorVisitor::visit(
    const expressions::Int64Constant *e) {
  ProteusValue valWrapper;
  valWrapper.value =
    ConstantInt::get(context->getLLVMContext(),
                     APInt(64, e->getVal()));
  valWrapper.isNull = context->createFalse();
  return valWrapper;
}
```

Here, an *LLVM* value is created from *int64_t* value, and wrapped in a *ProteusValue* struct.

4 EXTERN EXPRESSION/OPERATOR

As part of my project, we wanted to model calling externally defined functions from the Proteus context. As an example, one could want to run the following SQL query with a user-defined function *square*:

```
    SELECT value, square(value) AS squared_value
FROM numbers;
```

We want to be able to model a situation, where the *square* function is arbitrarily complex, and could be defined outside of Proteus. But still, we don't want to introduce the additional complexity of migrating the data outside of Proteus, we want it to happen purely inside the Proteus environment.

We implemented this feature by adding a new kind of Proteus expression, called *ExternExpression* with the following interface:

```
ExternExpression(string name,
                 vector<expression_t> args,
                 ExpressionType* type)
```

The *name* variable refers to the name of the function. The function can take an arbitrary number of arguments, which is modeled with the *vector<expression_t>* type. Also, the type of the expression has to be specified when creating *ExternExpression*, as it can't be automatically deduced when creating the expression (it's runtime/context agnostic).

4.1 Adding new external functions

Currently, all the external targets, which are the sources for external functions, are linked into Proteus's LLVM context in the *Context::registerExternModules* function.

One thing that we had to decide on is the way "external" (user-defined) functions should be made accessible in the Proteus environment. The Proteus's *Context* object, which is a wrapper around LLVM's code generation APIs, provides a 'getModule()' to access the base *LLVMModule* object that LLVM-IR is currently being inserted into. A single query may generate and compile multiple modules, one for each compiled pipeline in the query. In the context of this work, we only use queries with a single pipeline/module. In Proteus, all code is generated in the modules accessed by *Context->getModule()*, so symbols for external functions must be visible within this module in order to be called by Proteus-generated code. Currently, this is done by requiring the libraries implementing those external functions to provide a C++ function that returns a *LLVMModule* with LLVM-IR functions implemented. Those modules are then linked into one owned by the Proteus Context.

4.2 ExternExpression examples

This allows for using the external expressions as such (with simplified syntax):

```
std::vector<int32_t> v1{4, 5, 7};
std::vector<int32_t> v2{6, 7, 8};

auto builder = RelBuilderFactory{}
  .getBuilder()
  .scan({{/*attrName*/ "a", v1}, {"b", v2}})
  .unpack()
  .project([](const auto& arg) {
      return expressions::ExternExpression(
        "subtract",
        {arg["a"], arg["b"]},
        new IntType());
  });
```

Since Proteus's *expression* is at the core of its execution framework, the implementation of *expressions::ExternExpression* is enough to be used in any operator. Here is an example with the *filter* operator:

```
std::vector<int32_t> v1{4, 5, 7};
std::vector<int32_t> v2{6, 7, 8};
```

```
auto builder = RelBuilderFactory{}
  .getBuilder()
  .scan({{/*attrName*/ "a", v1}, {"b", v2}})
  .unpack()
  .filter([](const auto& arg) {
    return expressions::ExternExpression(
      "add", {arg["a"], 1}, new IntType()
    ) < arg["b"];
  });
```

4.3 External Operator

The *ExternExpression* defined above properly describes functions accepting a single tuple as their input. They cannot model functions that repeatedly consume tuples before computing a final result. Some examples of such functions that can't be implemented using tuple-at-a-time functions are average and sum functions.

Also, as part of this project, an experimental and not thread-safe way of handling those functions has been added. As an example, sum and average functions are available as external operators.

```
std::vector<int32_t> v{4, 5, 7};

RelBuilderFactory{}
  .getBuilder()
  .scan({{/*attrName*/ "a", v}})
  .unpack()
  .reduceExtern("average",
    /*resultType*/ new FloatType());
```

Right now it's assumed that external operator functions accept any number of tuples, each containing only one value.

Those functions should have the following interface:

```
func @func_name(
  %input: memref<?xargType>,
  %size: index
) -> resultType
```

It's currently implemented using states. For each function (like sum), a global variable of type *memref<?xargType>* is stored, along with the current length. The functions *func_name_append* and *func_name_compute* are then created (currently written by hand). The function *reduceExtern* is implemented by chaining two project operators, with a value materialization in between them. First, all tuples are appended to the global state by calling *func_name_append* function. Then the materialization of all values is forced. Here is a simplified implementation of this operator:

```
auto appendData = [&](const auto& arg) {
  return expressions::ExternExpression(
    funcName + "_append",
    {arg[0]},
    new IntType());
};

auto computeResult = [&](const auto& arg) {
  return expressions::ExternExpression(
    funcName + "_compute",
    { },
    resultType);
};

// append data, materialize data,
// then filter and compute the result
auto result = project(appendData)
                .pack()
                .unpack()
                .filter([](const auto &arg) {
                  return arg[0] == 0;
                })
                .project(computeResult);
```

Here the *funcName_append* function returns the size of the state before appending. This way, we can filter out non-zero elements, and compute the result only once. *Pack* and *unpack* called in succession force materialization (so all the values are appended before the actual compute is called).

This approach requires some boilerplate code to be written. Ideally, this would be made thread-safe, and the state and 2 helper functions would be auto-generated (perhaps by creating a new MLIR dialect).

4.4 State in Proteus's operators

Stateful operators are supported natively in Proteus. It does so with the concept of pipelines. At any point, Proteus's contexts hold a stack of pipelines. Each pipeline refers to a part of the generated code. They provide functionality for managing the state, and code generation.

Each operator can register a new pipeline and can register open and close functions, that allow for initialization and deinitialization of a state, This is done by using *context->registerOpen* and *context->registerClose* methods.

When an operator is a pipeline breaker, for example, operators that introduce parallelism or operators that must materialize data such as a hash join, it creates a new pipeline. Creating a new pipeline is done inside the *<OperatorName>::produce_* function, which pushes the new pipeline on the stack. Then, any operator generating code in that pipeline can add its own LLVM-IR state variables to the pipeline state, generally also registering callbacks to generate the LLVM-IR necessary to initialize their state. This approach allows maintaining a state local to an operator, with its lifecycle taken care of. This also means that operators that are not pipeline breakers, e.g., filter and project, are oblivious to exactly which pipeline's state they are adding to. Further, Proteus supports initializing/deinitializing state with C++ code by registering callbacks to be called when the pipeline is opened and closed.

This would also be the preferred way to implement external operators. The state, in that case, would need to contain a *memref* object. Since it's not practical to allocate them in LLVM IR, there should be a function implemented in MLIR, that would allocate a new *memref* object when called or possibly call out to c++ code to do so.

5 CASE STUDY

A significant part of this project involved searching for interesting libraries that expose interesting functions via the MLIR infrastructure. A prime motivation for this search was the ability to run machine learning (ML) prediction on data in Proteus.

5.1 "Toy" language

The Toy-MLIR project is a tutorial and provides a hands-on introduction to the MLIR framework within LLVM. It guides through the creation of a compiler for a simple educational language called "Toy." It gives a walkthrough of creating a simple parser and defining an MLIR dialect. It then goes on to show how to implement high-level abstraction transformations like shape inference and inlining. Overall, it's an interesting but simple example of using the MLIR infrastructure to reuse compiler components and expand them.

```
def multiply_transpose(a, b) {
  # Transpose is a built-in
  return transpose(a) * transpose(b);
}

def main() {
  # Toy is a tensor-based language
  var a<2, 3> = [[1, 2, 3], [4, 5, 6]];
  var b<2, 3> = [1, 2, 3, 4, 5, 6];
  var c = multiply_transpose(a, b);
  var d = multiply_transpose(b, a);
  print(d);
}
```

At the end of the compilation process, the MLIR module is lowered into the LLVM module, which then can be linked into Proteus's context. This proved to be a good starting point for calling into externally provided functions from the Proteus's environment. The only problem is that it's not relevant to our goal. As a way to avoid complex type inference, the "Toy" language inlines all the functions to the *main* function. It then removed all non-main functions. The *main* function doesn't take any arguments and doesn't return anything. For this reason, it doesn't fit our concept of external functions as transformations on tuples, since it can only be observed by its side effects.

5.2 ONNX-MLIR

ONNX [1] is an open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models - and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers. ONNX supports many runtimes: python, C++, Javascript, and many more.

ONNX-MLIR [2] project provides compiler technology to transform a valid ONNX graph into code that implements the graph with minimum runtime support. It implements the ONNX standard and is based on the underlying LLVM/MLIR compiler technology. It contributes:

- an ONNX Dialect that can be integrated in other projects,
- a compiler interfaces that lower ONNX graphs into MLIR files/LLVM bytecodes/C & Java libraries,
- an onnx-mlir driver to perform these lowering,
- and a python/C/C++/Java runtime environment.

Here is the workflow for obtaining a model and enabling it to be used from Proteus:

- Getting a pre-trained ONNX model.
- Generating an LLVM IR module using ONNX-MLIR, containing *@main_graph* method. This is the entry point to the model.
- Creating a wrapper function for calling that entry point.

As an example, I implemented a simple linear regression using ONNX with the following formula:

$$Y = X \cdot [0.5, -0.6]^T + 0.4 \tag{1}$$

Where X has the following shape: $[x_1, x_2]$.

Using *onnx-mlir* binary, this model (stored as a ".onnx" file) can be transformed into an MLIR with custom ONNX-MLIR dialects, or pure LLVM IR module.

The generated MLIR module contains a single function responsible for computing (running prediction) the model for several pairs $[x_1, x_2]$.

```
func @main_graph(%arg0: memref<?x2xf32>)
-> memref<?xf32>
```

Since *memref* is a concept from MLIR, ONNX-MLIR also comes with a C++ runtime library, that allows creating and manipulating *memref*s within the C++ code. This is problematic because when this MLIR module is lowered into an LLVM IR module, it has external dependencies to functions provided in the runtime. Because we can write MLIR code, we don't have to use them. To do that, I defined a new MLIR module, that created wrapper functions for calling *main_graph*:

```
func private @main_graph(%arg0: memref<?x?xf32>)
  -> memref<?xf32>

// Creates a memref<1x2xf32> from two f32
func @pack_two(%arg0: f32, %arg1: f32)
  -> memref<?x?xf32> {
  ...
}

// Extracts f32 from memref<1xf32>
func @extract_first_element(%arg0: memref<?xf32>)
  -> f32 {
  ...
}

// Calls the prediction model
func @compute_onnx(%arg0: f32, %arg1: f32)
  -> f32 {
  %args = call @pack_two(%arg0, %arg1)

  %preds = call @main_graph(%args)

  %res = call @extract_first_element(%preds)
```

```
  return %res : f32
}
```

That way, we're independent from the helper C++ runtime functions. Because they still need to be loaded, we compiled them into LLVM IR and loaded them as a module into Proteus's LLVM context.

Overall, ONNX-MLIR is a very good candidate for enabling the use of ML models using the MLIR infrastructure. It creates reusable dialects, which could be made a part of Proteus's language after migrating to LLVM.

5.3 Torch-MLIR

The Torch-MLIR [3] project aims to provide first-class compiler support from the PyTorch ecosystem to the MLIR ecosystem. PyTorch [8] is an open-source machine learning framework that facilitates the seamless transition from research and prototyping to production-level deployment. An overview of Torch-MLIR's architecture can be found in Figure 2.

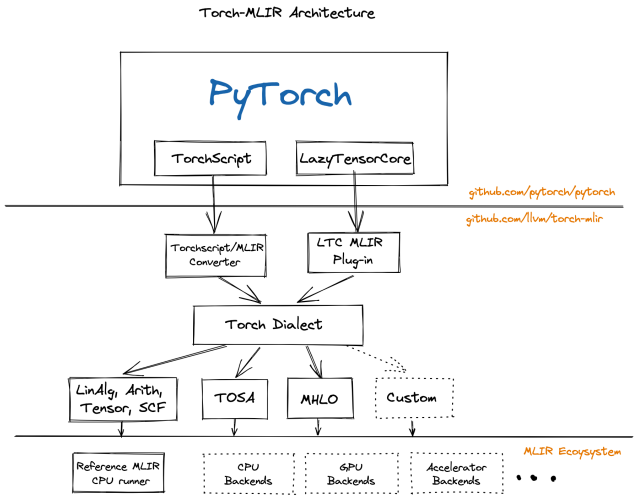

Figure 2: High-level architecture diagram for Torch-MLIR [3]

This project was explored as an alternative to ONNX-MLIR, when it wasn't sure that ONNX-MLIR would work. There were many reasons why we chose not to commit to using it after some exploration:

- It lacks a simple C++ frontend. Even after understanding its Python frontend, a significant amount of work would have to be done to allow the extraction of the underlying LLVM module.
- Version supporting LLVM 14.0.0 is quite outdated and has much worse documentation and examples compared to answer version. Sadly LLVM 14 is the newest LLVM version supported by Proteus as of writing this report.
- It requires a prebuilt *pytorch* package, which is hard to integrate into Proteus build and be built as a dependency from scratch. Pytorch installed via package managers uses libstdc++, whereas Proteus's build system requires LLVM's libc++. This causes a mismatch between symbols.

5.4 MLIR dialects

MLIR allows the integration of multiple dialects, which are essential for engaging with and extending the MLIR ecosystem. Dialects enable the definition of new operations, attributes, and types, each within a unique namespace.

MLIR supports the coexistence of multiple dialects within a single module. Dialects are utilized and produced by various optimization and lowering passes, as MLIR offers a robust framework for converting between and within different dialects.

Some of the built-in dialects provided by MLIR include:

- Affine: Supports affine operations for loop optimization and polyhedral compilation.
- Standard: Contains common operations and types used across multiple domains.
- LLVM: Enables the conversion of MLIR to LLVM IR, facilitating integration with the LLVM ecosystem.
- Tensor: Defines operations and types for tensor computations, core for machine learning workloads.
- SCF (Structured Control Flow): Provides structured control flow operations like loops and conditionals.
- Arith: Supports basic integer and floating point mathematical operations.

Those verbose dialects show the diversity and versatility of MLIR in accommodating multiple high-level concepts at once.

To showcase this, we decided to create a simple MLIR compiler with support for those dialects. We wanted to use functions defined using those dialects as external functions in Proteus. Here is an example of a simple recursive "factorial" function that uses those dialects:

```
func @factorial(%n: i32) -> i32 {
  %c1 = arith.constant 1 : i32
  %cond = arith.cmpi sle, %n, %c1 : i32
  %result = scf.if %cond -> (i32) {
    scf.yield %c1 : i32
  } else {
    %n1 = arith.subi %n, %c1 : i32
    %rec = call @factorial(%n1) : (i32) -> i32
    %fact = arith.muli %n, %rec : i32
    scf.yield %fact : i32
  }
  return %result : i32
}
```

Those functions can be used with the *expressions::ExternExpression* syntax shown before.

6 PERFORMANCE

In this section, we delve into the performance implications of incorporating MLIR-based external functions into a JIT code-generated DBMS, using the Proteus framework as our case study. We specifically focus on two types of external functions: simple arithmetic operations and integrating a GPT2 machine-learning model using the ONNX-MLIR project.

To comprehensively assess the performance implications, we will concentrate on two critical metrics: the execution time of queries and the compilation/optimization time of the query code.

The comparison was quantified using the formula

$$\frac{\text{mean(experiment)} - \text{mean(baseline)}}{\text{mean(baseline)}} * 100\%$$

providing a percentage-based analysis of performance differences. This is shown in the "% Change" column in the tables.

6.1 Simple arithmetic

Incorporating simple arithmetic operations as external MLIR-based functions into Proteus introduces new dimensions to the compilation and optimization processes. Our focus here is to evaluate the impact of compiling MLIR code at runtime and subsequently optimizing the generated LLVM IR functions. Specifically, we aim to measure the overhead introduced by the need to optimize an increased code volume due to integrating these external functions.

The compilation of MLIR code involves translating high-level representations into LLVM IR, a process that could potentially increase the compile time of a query. The subsequent optimization of this LLVM IR is crucial for ensuring efficient execution. However, optimizing a greater amount of LLVM IR code can significantly extend the optimization phase, impacting the overall system performance.

6.1.1 *Experiment setup.*

In our experiment, we evaluated the performance impact of incorporating MLIR-based arithmetic operations by comparing two configurations. The baseline configuration used native Proteus arithmetic operations, executing a sequence of built-in operators (multiplication, subtraction, addition) directly within the query environment on two data arrays. For the experimental setup, the same arithmetic operations were conducted using external functions defined in MLIR, invoked through the *expressions::ExternExpression* interface. This required the runtime compilation of MLIR code and integration of the generated LLVM IR functions. We measured and compared the total execution time and various detailed execution and optimization metrics between these configurations to assess the overhead introduced by the use of external MLIR functions. In this experiment, we compared compilation and execution times averaged over 500 runs of both the MLIR approach and built-in Proteus operators.

The experiment query first loads two vectors (representing columns a and b) containing around 30 million 32-bit integers. Then the following expression is computed for each tuple:

```
(arg["a"] * arg["b"]) + (arg["a"] - arg["b"])
```

The resulting tuples are then reduced, and the maximum value is computed. This is done to remove the impact of printing computed values.

6.1.2 *Results.*

Refer to Table 1 for detailed results. The experiment demonstrates that integrating MLIR incurs non-negligible overheads, especially during the compilation and optimization stages. The overall change in query execution duration was small (0.9%), with the query using plain Proteus operators being faster. However, it is generally expected that using MLIR-compiled functions could lead to performance enhancements, as they are implemented in a higher-level code than Proteus's native operations, which are written in LLVM IR. These extra abstraction levels (introduced as dialects) allow the compiler to use certain complex properties and enable dialect-specific optimization passes.

6.2 GPT2 model

In this section, we compare two methods of integrating machine learning models into the runtime environment of Proteus queries using the ONNX-MLIR project. The ONNX-MLIR project allows processing models in ONNX format and lowering them to LLVM IR using the MLIR framework. It supports the compilation of the model into various formats:

- LLVM bytecode file (.bc),
- Shared library file (.so),
- Object file (.o).

We have tested two of these approaches: bytecode and shared library files. The bytecode file can be linked to the LLVM module corresponding to a compiled query, while the shared library file can be linked during the building of the Proteus project, rather than at runtime. Regardless of the method, the *main_graph* function, serving as the entry point to the model's execution code, is made accessible in the query runtime environment.

Each method has its advantages and disadvantages. Generally, linking the compiled module directly into the query code allows for extensive optimizations, as the entire model execution code becomes part of the query's LLVM module. This integration enables more aggressive optimizations but also introduces more complexity to the query's compilation and optimization processes. Therefore this method is likely to increase the time required for query compilation, whereas it may reduce the actual query runtime.

6.2.1 *Experiment setup.*

For this experiment, we utilized the GPT2 model from [7], which is particularly suitable due to its substantial size. We employed an open-source implementation of the GPT2 Tokenizer from [9]. Additionally, we developed a wrapper for the core GPT2 model to enable the generation of multiple tokens. In this benchmark, the query generated 40 tokens from a sample input sentence and we measured the average compilation and execution times over 15 runs for both approaches.

In this experiment, we compared linking the model in the runtime (from a bytecode file into the query LLVM module) against linking the compiled model into the Proteus binary as a shared library.

6.2.2 *Results.*

Refer to Table 2 for detailed results. The results are as expected. The execution times were slightly increased when using a pre-compiled shared library. On the other hand, the compilation times were massively decreased. This shows a great potential for using pre-compiled libraries when dealing with machine learning models of significant size.

7 CONCLUSION

In this project, we have explored the integration of MLIR-based user-defined functions (UDFs) into a just-in-time (JIT) code-generated database management system (DBMS). Through a detailed analysis and implementation, we have demonstrated the feasibility and advantages of incorporating MLIR into Proteus, a JIT-enabled DBMS.

Our work focused on enabling the use of externally defined functions within the Proteus environment without the need to migrate data outside the system.

We successfully implemented a new kind of Proteus expression, named *ExternExpression*, which allows the invocation of complex functions defined in MLIR from within Proteus. This integration provides the benefits of high-level abstractions offered by MLIR, such as support for multiple dialects, robust optimization frameworks, and seamless interaction with the LLVM ecosystem.

A highlight of our project was the creation of a simple MLIR compiler that supports various MLIR dialects, demonstrating the versatility and power of MLIR in handling complex operations. By integrating ONNX-MLIR, we showed that it is possible to leverage pre-trained machine learning models directly within the DBMS, enhancing its capabilities for data processing and analysis.

Our case studies, including the use of a toy language and Torch-MLIR, highlighted the practical challenges and solutions in adopting MLIR for UDFs. Despite some limitations, such as version compatibility issues and the need for prebuilt packages, our findings indicate significant potential for MLIR to streamline the development and execution of UDFs in a JIT-compiled environment.

Incorporating the findings from our performance analysis, the integration of MLIR-based UDFs in Proteus demonstrates promising enhancements in system performance and extensibility. Our experiments with simple arithmetic operations and a GPT2 machine learning model reveal that while there is an overhead associated with compiling and optimizing MLIR code, the potential for enabling new features in complex query environments is significant. These results align with our project's objective to improve code reusability and optimization capabilities within the Proteus framework.

In conclusion, this project sets a foundation for further exploration and development of MLIR-based UDFs in JIT code-generated DBMS. The integration of MLIR not only enhances the expressiveness and efficiency of UDFs but also opens new avenues for leveraging advanced machine learning models and optimizations within the DBMS framework.

8 FUTURE WORK

8.1 Type visitors

Looking back at the external expression in Section 4, they have the following constructor:

```
ExternExpression(string name,
                 vector<expression_t> args,
                 ExpressionType* type)
```

Here it's required to specify the *type* object representing the return type of a function. This is necessary because there are a lot of places, where Proteus uses the type of an expression for code generation. Sometimes, the type has to be known before the expression is evaluated. One example is code generation for flushing values in *cvs-plugin*.

In Proteus, the type of an expression is assigned when it is created. It is currently implemented with each expression storing a *type* variable, which can represent one of the multiple types supported. E.g. *IntType*, *Int64Type*, *FloatType*. Because of that, the type has to be computed independently of the Proteus's context, and more specifically, independently of the *LLVMModule* stored by it. That's why it's necessary to specify the type of external expression when creating it.

In the Proteus's codebase, there is a draft of the *ExpressionTypeVisitable* interface. This is an analog (but for types) of *ExprVisitorVisitable*, which allows evaluating expressions in specific runtimes. After adding support for evaluating the expression type by "visiting" it, it would be possible to not specify the expression type, and could be evaluated in runtime from Proteus's context instead.

8.2 Defining wrappers for external libraries

Currently, all libraries providing external functions were added manually. For some of those libraries, a wrapper had to be written to expose certain functions to the Proteus's libraries. This was the cast for functions, that were not created to be called as they are. For example, in Section 5.2, a wrapper for *main_graph* had to be handwritten to expose a function with a proper interface.

This leads to writing a lot of repetitive code, enough to cause the developers to make mistakes. Similarly, all implementations of models exposed by ONNX-MLIR would have a very similar wrapper code, where only types and dimensions differ.

For this reason, the creation of this code should be automated. This might be difficult to do for an arbitrary library, but for some specific "supported" libraries there should be a seamless way to register new functions. One specific use case with ONNX-MLIR could be that a user provides a compiled model and all the wrapper code is generated for it automatically.

REFERENCES

[1] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[2] ONNX-MLIR Contributors. 2020. ONNX-MLIR: An open-source compiler infrastructure for ONNX and MLIR. https://github.com/onnx/onnx-mlir.

[3] Torch-MLIR Developers. [n. d.]. Torch-MLIR: An integration of PyTorch and MLIR. https://github.com/llvm/torch-mlir.

[4] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries Over Heterogeneous Data Through Engine Customization. *Proc. VLDB Endow.* 9, 12 (2016), 972–983. https://doi.org/10.14778/2994509.2994516

[5] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[6] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[7] ONNX. 2018. ONNX Model Zoo. https://github.com/onnx/models.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martín Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. 8024–8035.

[9] Kuiyi Wang. 2023. Huggingface Tokenizer in C++. https://github.com/wangkuiyi/huggingface-tokenizer-in-cxx. Accessed: 2024-07-04.

A APPENDICES

Table 1: Performance Metrics Comparison Between Plain and MLIR Operators

Metric	Plain (ms)	MLIR (ms)	% Change
Total Execution Time (T)	24.5	24.7	0.937%
Execution Time (Texecute)	24.5	24.7	0.936%
Execution Time w/ Sync	24.5	24.8	0.937%
MLIR Compiling and Linking	-	4.4	-
Compile and Load (CPU)	3.14	3.19	1.56%
Compile and Load (CPU, waiting - critical - getKernel)	40.3	46.4	15.2%
Compile and Load (CPU, waiting - critical)	0	0	0%
Compile and Load (CPU, waiting)	58.9	65.2	10.6%
Data Loading (Current)	0	0	0%
Module Copying	2.03	2.03	0.0833%
Optimization Avoidable Phase	5.52	5.98	8.29%
Optimization Phase	31.9	34.9	9.42%
Optimization Run Phase	26.9	29.8	11.1%

Table 2: Performance Metrics for Runtime vs. Precompiled Linking of GPT2 Model

Metric	Runtime (sec)	Shared Lib (sec)	% Change
Total Execution Time (T)	145.525	149.931	3.03%
Execution Time (Texecute)	145.525	149.931	3.03%
Execution Time w Sync	145.525	149.931	3.03%
MLIR Compiling and Linking	10.7	0.009	-99.9%
Compile and Load (CPU)	26.325	0.002	-100%
Compile and Load (Critical - getKernel)	70.513	0.136	-99.8%
Compile and Load (Critical)	0.032	0	-100%
Compile and Load (Waiting)	70.542	0.136	-99.8%
Data Loading (Current)	0	0	0%
Module Copying	26.324	0.002	-100%
Optimization Avoidable Phase	0.003	0.003	-14.1%
Optimization Phase	37.947	0.055	-99.9%
Optimization Run Phase	37.944	0.052	-99.9%