Hardware aware code generation for Rust

Klas Segeljakt <klasseg@kth.se>

March 9, 2018

Contents

List of Tables

List of Figures

1 Abstract

Continuous Deep Analytics is a new type of analytics with performance requirements which exceed what systems like Spark and Flink can offer. The backbone of these systems is the Java Virtual Machine (JVM). While the JVM is portable, it is a also performance bottleneck due to its garbage collector. A portable runtime with high-performance, capable of running efficiently on heterogeneous architectures, is needed. We have developed a code generator as a framework for this runtime. Instead of running portable but slow Java code, highly optimized Rust code will be generated to different architectures. By generating Rust code, as opposed to C or C++ code which is the common approach, we achieve stronger safety guarantees. The code generator will be used in a five year project by KTH and RISE SICS. In this project a new distributed system will be developed to meet the demands of CDA. Due to time constraints, the generator is currently only able to generate a subset the Rust language. It is however designed to be extensible in respect to adding new constructs. Adding full support for all of Rust's features is left for future work.

2 Acronyms and Keywords

Acronym	Expansion
AST	Abstract Syntax Tree
IR	Intermediate Representation
P-IR	Physical-IR
L-IR	Logical-IR
DSL	Domain Specific Language
GPL	General Purpose Language
CDA	Continuous Deep Analytics
JVM	Java Virtual Machine

3 Introduction

Deep Analytics in the field of data mining is the application of data intensive processing techniques . Data can come from multiple sources in a structured, semi-structured or unstructured format. Continuous Deep Analytics (CDA) is a new form of Deep Analytics where data is both massive, unbound, and live.

This thesis is part of a five year project by KTH and RISE SICS to develop a system capable of CDA. The CDA system must be able to run for long periods of time without interruption. It must also be capable of processing incoming queries in short time windows to support real-time decision making. CDA is aimed towards both the public sector and industry. It will enable new time-sensitive applications such as zero-time defense for cyber-attacks, fleet driving and intelligent assistants. These applications involve machine learning and graph analytics, both of which might require large scale data intensive matrix or tensor computations. Finishing these computations in short periods of time is infeasible without hardware acceleration. The system thereby needs to be able to exploit the available hardware resources to speedup computation.

Hardware acceleration is not easy. Developers must have expertise with multiple APIs and programming models which interface with the drivers, e.g., CUDA, OpenCL, OpenMP and MPI [1]. As interfaces change, developers need to update their code. Moreover, machines in distributed systems can have various hardware configurations. For example, when scaling out, one may choose to add new machines with better hardware. This becomes an issue as code for one architecture might not be portable to others.

In consequence, many distributed systems use hardware virtualization to abstract the physical hardware details away from the user [1]. Spark and Flink realize hardware virtualization through the Java Virtual Machine (JVM) [2]. While the JVM is highly optimized and portable, it is not well suited for interfacing with the GPU **???**. It also has a big runtime overhead, partially from garbage collection. Evaluation by **???** has shown that a laptop running handwritten low-level code can outperform a 128 core native Spark cluster in PageRank. The evaluation measured 20 PageRank iterations for medium sized graphs of ~105M nodes and ~3.5B edges .

The CDA system will try to obtain both portability and performance at the same time by using code generation. Instead of writing different code for different hardware, one can write code which is generated for different hardware. At the front-end, the user will describe the desired behavior of the program in a high-level declarative language. Then, at the back-end, a code generator will generate corresponding low-level code, tailored to a set of specific hardware configurations.

The code generator is written as a library in a host language. The host language will act as a meta-language which translates an IR provided by the front-end into a target language with the corresponding behavior. The target language code is then compiled and deployed to different hardware configurations.

3.1 Problem

C and C++ are a commonly used as the target language for code generation. While both compile to fast machine code, neither provide strong safety guarantees. Double free errors, null pointer dereferences and segmentation faults are recurrent errors [4]. The CDA code generator will therefore instead generate Rust code. Rust is a recent programming language which achieves both safety and performance through a special memory policy. To our knowledge, no Rust code generation framework exists yet for Scala. Thereby, the problem is to implement this framework.

The code generator is composed of two parts. First, it must be able to provide a data structure for expressing the Rust language in Scala. The representation will be in the form of an Abstract Syntax Tree (AST). Then, it should be able to output the corresponding Rust code and compile it to an executable. This thesis focuses on the former part, while the latter is covered in a thesis by Oscar Bjuhr.

The framework's Rust AST should be able to express a majority of Rust's language constructs. As CDA developers will use the framework to implement IR nodes, it is important to provide static semantic checking of AST nodes, e.g., type checking. The framework interface should be high level and declarative with minimal boilerplate code. Finally, since Rust is a rapidly evolving language, the Rust AST must be extensible in respect to adding new constructs.

The problem statement for this thesis can be defined as: "How do you implement a framework for generating Rust code in Scala which is both robust, declarative, extensible, and statically safe?".

Rust is a recent programming language with stronger safety guarantees.

The front end of systems such as Spark and Flink consists of a query interface. The interface provides a set of transformers and actions. Users can also define custom user defined functions (UDFs).

For a given query, the code generator must both optimize each stage of the query and the query plan as a whole. Another issue is User Defined Functions (UDFs). UDFs are essentially black boxes whose functionality might not be known at compile time. The task of optimizing these is a monumental challenge, and will be left out for future work. Another topic of interest is whether code also could be generated for the network layer. Most modern day switches are programmable, and could allow for further optimizations **???**.

A possible approach to mitigate the performance loss, while still maintaining portability is to use a domain specific language Domain Specific Language (DSL) [5]. DSLs are minimalistic languages, tailor-suited to a certain domain. They bring domain-specific optimizations which general-purpose languages such as Java are unable to provide. DSLs can optimize further by generating code in a low level language, and compiling it to binary code which naturally runs faster than byte code. C and C++ are commonly used as the target low-level language. We regard Rust as a candidate as it provides safe and efficient memory management through ownership.

Previous work by **???** has shown that Spark's performance can be improved to be much faster through the use of DSLs. There is no equivalent solution yet for Flink, and this thesis aims to address this. The problem can be summarized as the following problem statement: How can Apache Flink's performance be improved through a Rust DSL?

The program generator is written in a meta-language, and generates hardware-sensitive code for a target-language.

The code generator is written programmed with a DSL. DSLs are minimalistic languages, tailor-suited to a certain problem domain.

3.2 Purpose

The purpose of this thesis is to provide a useful resource for developers of distributed systems. It is specifically aimed at those who seek to boost the performance of distributed systems which are built on top of the JVM.

3.3 Goal/Contributions

The goal of this thesis is to explore the area of DSLs with respect to Rust and Scala. The following deliverables are expected:

- A Rust DSL written in Scala.
- An evaluation of Apache Flink's performance before and after integrating the Rust DSL.

3.4 Benefits, Ethics and Sustainability

Flink is being used by large companies such as Alibaba, Ericsson, Huawei, King, LINE, Netflix, Uber, and Zalando. As performance is often a crucial metric, these companies may see benefits in their business from the outcome of this thesis. As an example, Alibaba uses Flink for optimizing search rankings in realtime. By improving Flink's performance, it may allow for more complex and data intensive search rank optimizations. This will in consequence be beneficial for the customer whom will have an easier time finding their product.

Low level code is able to utilize hardware more efficiently than high-level code. Thus, better performance in this case also implicates less waste of resources. As a result the power usage goes down, which is healthy and sustainable for the environment.

In terms of ethics, it is crucial that the code generator does not generate buggy code which could compromise security. Keeping the code generation logic decoupled from the client code is also important. Without this consideration, an attacker would be able to generate malicious code to stage an attack with ease.

3.5 Related Work

3.5.1 Flare

Our approach to optimizing Flink draws inspiration from Flare which is a back-end to Spark [6]. Flare bypasses Spark's inefficient abstraction layers by 1. Compiling queries to native code, 2. Replacing parts of the Spark runtime, and by 3. Extending the scope of optimizations and code generation to UDFs. Flare is built on top of Delite, a compiler framework for high performance DSLs, and LMS, a generative programming technique. When applying Flare, Spark's query performance improves significantly and becomes equivalent to HyPer, which is one of the fastest SQL engines.

3.5.2 Weld

Weld is an interface between different data-intensive libraries. Re-using code is a central part of software development. The field of distributed systems has a large eco-system of open-source software. As an example, libraries such as NumPy and Tensorflow are being used together in data intensive analytic applications. One of the most common optimizations in distributed processing is lazy evaluation. Through lazy evaluation, data of the distributed computation is materialized only when the computation is finished. This speeds up performance by reducing the data movement overhead.

Libraries are naturally modular: they take input from main memory, process it, and write it back. As a side effect, successive calls to functions of different libraries might require materialization of intermediate results, and hinder lazy evaluation.

Weld aims to solve this problem by bridging the gap between libraries through a common IR. The libraries submit their computations in IR code to a lazily-evaluated runtime API. The runtime dynamically compiles IR code fragments and applies optimizations such as loop tiling, loop fusion, vectorization and common subexpression elimination. The IR is minimalistic with only two

abstractions: builders and loops. Builders are able to construct and materialize data, without knowledge of the underlying hardware. Loops consume a set of builders, apply an operation, and produce a new set of builders. By optimizing data movement, Weld is able to speedup programs using Spark SQL, NumPy, Pandas and Tensorflow by at least 2.5x.

3.5.3 ScyllaDB

NoSQL is a new breed of high performance data management systems. They store data in more flexible formats than the regular tabular format found in SQL systems. As a result, they are both able to store more data, and are able to read and write it faster. One of the leading NoSQL data stores is Cassandra, developed by Facebook. Cassandra is written in Java and provides a highly customizable and decentralized architecture. Following Cassandra's success came ScyllaDB. ScyllaDB is an open-source re-write of Cassandra into C++ code with focus on utilization of multi-core architectures, and abolishing the JVM overhead. Most of Cassandra's logic is retained in ScyllaDB. One notable difference is their caching mechanisms. Caching reduces the disk seeks on read operations. This helps decrease the I/O usage which can be a major bottleneck in distributed storage systems. Unlike Cassandra's cache, ScyllaDB's cache is dynamic. ScyllaDB will allocate all available memory to its cache and dynamically evict entries if memory is required for other tasks. This would be less feasible in Cassandra where memory is managed by the garbage collector. In evaluation, ScyllaDB's caching strategy improved the reading performance by less cache misses, but also had a negative impact on write performance.

3.5.4 Voodoo

Voodoo is a declarative intermediate algebra which abstracts away details of the underlying hardware. It is able to express advanced programming techniques such as cache conscious processing in few lines of code. The output is highly optimized OpenCL code. It could be classified as an external DSL, as it has its own compiler. It has been applied as a backend to MonetDB, which is a high-performance query processing engine, with good results.

Code generation is complex. Different hardware architectures have different ways of achieving performance. Moreover, the performance of a program depends on the input data, e.g., for making accurate branch predictions. As a result, code generators need to encode knowledge both about hardware and data to achieve good performance. In reality, most code generators are designed to generate code solely for a specific target hardware. Voodoo solves this through providing an IR which is portable to many different hardware targets. It is expressive in that it can be tuned to capture hardware-specific optimizations of the target architecture, e.g., data structure layouts and parallelism strategies. Some additional defining characteristics of the Voodoo language are that it is vector oriented, declarative, minimal, deterministic and explicit. Vector oriented implicates that data is stored in the form of vectors, which conform to common parallelism patterns. By being declarative, Voodoo describes the dataflow, rather than its complex underlying logic. It is minimal in that it consists of non-redundant stateless operators. It is deterministic, i.e., it has no control-flow statements, since this is expensive when running SIMD unit parallelism. By being explicit, the behavior of a Voodoo program for a given architecture becomes transparent to the front end developer.

Voodoo is able to obtain high parallel performance on multiple platforms through a concept named controlled folding. Controlled folding folds a sequence of values into a set of partitions using an operator. The mapping between value and partition is stored in a control vector. High performance is achieved by executing sequences of operators in parallel. Voodoo provides a rich set of operators for controlled folding which are implemented in OpenCL. Different implementations for the operators can be provided depending on the backend.

When generating code, Voodoo assigns an Extent and Intent value to each code fragment. Extent is the degree of parallelism while Intent is the number of sequential iterations per parallel work-unit. These factors are derived from the control vectors and optimize the performance of the generated program.

3.6 Delimitations

- ***What was intentionally left out?***
- ***References.***

Only Rust will be used as the target language for code generation. It would be interesting to compare its performance to C and C++, but this is out of scope for this thesis. Another delimitation is regarding the problem of optimizing UDFs. Existing solutions to this problem will be described in the background. The task of inventing a new solution, specialized for Flink, will be left for future studies.

3.7 Conventions used in this paper

Italic Used when a new term is introduced

`Monospace` Used for displaying code

`Bold Monospace` Used for displaying commands typed by the user

4 Background

The following sections describe the Rust programming language, Domain Specific Languages and the Scala language.

4.1 The Rust Programming Language

For many years, C has been used as the main-goto low-level system development language. While C is very optimized, it is also unsafe. Mistakes in pointer aliasing, pointer arithmetic and typecasting can be hard to detect, even for advanced software verification tools. Meanwhile, high level languages like Java solve the safety issues through a runtime which manages memory with garbage collection. This safety comes at a cost since garbage collection incurs a big overhead.

Rust is a relatively new low-level programming language. It achieves both performance and safety through a memory management policy based on ownership. In addition, Rust provides many zero-cost abstractions, e.g., pattern matching, generics, traits, and type inference. Packages, e.g., binaries and libraries, in Rust are referred to as crates. Cargo is a crate manager for Rust which can download, compile, and publish crates. Rust has a big ecosystem of open-source crates which can be browsed on `https://crates.io`. Since Rust's original release, it has seen multiple major revisions. Some dropped features include a typestate system , and a runtime system with green threaded-abstractions .

4.1.1 Ownership

In Rust, when a variable is bound to an object, it takes ownership of that object. Ownership can be transferred to a new variable, which in consequence breaks the original binding. Variables can however temporarily borrow ownership of an object without breaking the binding. An object can be either mutably borrowed by at most one variable, or immutably borrowed by any number of variables. Hence, objects cannot be mutably and immutably borrowed at the same time.

By restricting aliasing, ownership solves many safety issues found in other low level languages, such as double-free errors, i.e., freeing the same memory address twice. Moreover, Rust effectively eliminates the risk of data-races as ownership applies across threads.

4.1.2 Lifetimes

Objects are dropped, i.e., de-allocated, when their owner variable's lifetime ends. The lifetime of a variable in Rust is currently determined by the Lifetime is

4.1.3 Mutable aliasing

Ownership can in some cases be too restrictive. There are in general two ways to achieve mutable aliasing. The first way is to use a reference counter (`Rc<T>`) together with interior mutability (`RefCell<T>`). The reference counter, i.e., smart pointer, allows an object to be immutably owned by multiple variables simultaneously. An object's reference counter is incremented whenever a new ownership binding is made, and decremented when one is released. If the counter reaches zero, the object is de-allocated. Interior mutability lets an object be mutated even when there exists immutable references to it. It works by wrapping an object in a `RefCell`. Variables with a mutable or immutable reference to the `RefCell` can then mutably borrow the wrapped object. By combining reference counting with interior mutability (`Rc<RefCell<T>>`), multiple variables can own the `RefCell` immutably, and are able to borrow the object inside mutably.

The other way of achieving mutability is through unsafe blocks. Unsafe blocks are blocks of code wherein raw pointers can be dereferenced. Raw pointers are pointers without any safety guarantees. Multiple raw pointers can point to the same memory, and the memory they point to might not be allocated. Code inside of unsafe blocks have the potential to cause segmentation faults or other undefined behavior and should be used with caution.

4.1.4 Use cases

Upfront, ownership might appear to be more of a restriction than a benefit compared to other memory models. Other languages might be better for prototyping. Albeit the restrictiveness, ownership can solve complex security concerns such as Software Fault Isolation (SFI) and Static Information Control (IFC).

Software fault isolation (SFI) enforces safe boundaries between software modules. A module should not be able to access the another's data without permission. As an example, C can violate SFI since a pointer in one module could potentially access another module's heap space. In contrast, Rust's ownership policy ensures that an object in memory is naturally accessible by only one pointer. Ownership, and information, can securely be transferred between modules without violating SFI.

Static information control (IFC) enforces confidentiality by tracing information routes of private data. This becomes very complex in languages such as C where aliasing can explode the number of information routes. Meanwhile, IFC is easier in Rust due to its aliasing restrictions.

4.2 RustBelt

Rust is a new programming language designed to overcome the tradeoff between the safety of high-level languages and control of low-level languages. The core design behind Rust is ownership. Previous approaches to ownership were either too restrictive or too expressive. Rust's ownership discipline is that an object cannot be mutated while being aliased. This eliminates the risk for data races. In cases where this model is too restrictive, Rust provides unsafe operations for mutating aliased state. Safety of these operations cannot be ensured by the Rust compiler.

While Rust is safe without using unsafe operations, it is questionable how safe its libraries are. Many Rust libraries, including the standard library, makes use of unsafe operations. RustBelt is an extension to Rust which verifies the soundness of code that uses unsafe operations. It builds a semantic model of the language which is then verified against typing rules. A well-typed program should not express any undefined behavior.

4.3 Domain Specific Languages (DSL)

Domain Specific Languages (DSLs) are small languages suited to interfacing with a specific domain. Domain contexts can be implemented directly in general purpose languages (GPLs) without the use of DSLs, but this would require many more lines of code.

DSLs can either be external or embedded. External DSLs exist in their own ecosystem, with their own compiler, debugger, and editor. Building and maintaining these tools require copious amounts of work. In contrast, embedded DSLs are are embedded within a host GPL. Embedded DSL's take less time to develop, but are also restricted in their expressiveness by the host GPL's syntax.

Embedded DSLs can either have a shallow or deep embedding. Shallow embedding implicates that the DSL is executed eagerly without constructing any form of intermediate representation. Deep embedding means that the DSL creates an intermediate abstract syntax tree (AST) which can be optimized before being evaluated.

There is a tradeoff between deep and shallow embeddings. Deep embeddings which represent a program as an AST can be compiled and optimized. The AST can grow large and complex to accommodate for new language features. This results in a DSL which is difficult to program. Shallow embeddings which directly translate from constructs to semantics are easier to implement. Although, by not having an AST, the DSL code cannot be optimized and verified.

Their tradeoff referred to as the *expression problem*. Deep embeddings are flexible in adding new interpretations, but inflexible in adding new constructs. Adding a new construct requires updating all existing interpretations. It is the opposite for shallow embeddings. Because the semantic domain is fixed in shallow embeddings, the representation dictates what operations can be performed. Thereby, there is essentially only one interpretation. Adding a new interpretation would therefore require updating all existing constructs.

It is possible to combine the two embedding styles and keep most of their strengths. This hybrid style is referred to as a deep embedding with shallow derived constructs. Derived constructs

Another approach is the Finally Tagless technique. It uses an interface to abstract over all interpretations. The interface can be implemented to create different concrete interpretations. New constructs and interpretations can be added with ease. Constructs are added by supplementing the interface with new methods. Interpretations are added by creating new instances of the interface.

4.4 The Scala Programming Language

4.4.1 Macros

Scala version 2.10 introduced macros. Macros enable compile-time metaprogramming. They have many uses, including boilerplate code generation, language virtualization and programming of embedded DSLs. Macros in Scala are invoked during compilation and are provided with a context of the program. The context can be accessed with an API, providing methods for parsing, type-checking and error reporting. This enables macros to generate context-dependent code. Scala provides multiple types of macros: def macros, dynamic macros, string interpolation macros, implicit macros, type macros and macro annotations.

4.5 Code generation

- Generate binary directly, or generate Rust and compile?

[1] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun, "Language virtualization for heterogeneous parallel computing," in *ACM sigplan notices*, 2010, vol. 45, pp. 835–847.

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, and others, "Apache spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[4] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the rust programming language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 66, 2017.

[5] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Parallel architectures and compilation techniques (pact), 2011 international conference on*, 2011, pp. 89–100.

[6] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf, "Flare: Native compilation for heterogeneous workloads in apache spark," *arXiv preprint arXiv:1703.08219*, 2017.