**Main Idea:**

dsl to scala ir (not part of project), ir to rust

front end: IR which can represent rust

back-end: take ast convert to rust binaries

**Proposal:**

Take RustAST and compose to rust code. Use rust compiler (optionally Cargo) to generate binaries for specific architectures. Dynamically link runtime tools, buffers etc.

**Possible solution:**

Compose a task into three parts:

* Task driver – binary, coordinates and redeploys task
* Runtime tools, e.g. implementation of data-transport in the system - sockets only have to implement std::io::Read and std::io::Write traits, returned as trait object <T: Read, Write> Box<T> - dynamically linked and used for initiation or reconfiguration of worker – **may become problem with socket type defined in dynamically linked library? May incur performance penalties for each call to socket due to dynamic lookup.**
* Task worker – actual task, can be sent as a dylib and dynamically linked by the driver. This would enable the system’s driver dynamically changing the role of workers during run time.

**Possible Extension:** Stream pipeline which can dynamically change similar to Ziria, where specific components can issue changes in the processor pipeline. And/or like Flink, software updates from client changes the pipeline. Or key-space can be dynamically readjusted to inhabit more intensely used parts of key-space with more workers. Update nodes directly after state is saved to stable storage as Flink, using watermarks and epochs.

**Possible solution:**

Physical plan consists of three roles:

* Input
* Task – stateful & stateless
* Sink – inherently stateful

Task are always local in Lime Java terminology, which means that they do not access any global fields. They can be run independently of any other task and there is no risk of race-conditions. There are two kinds of tasks, stateful and stateless. Stateless works as a pure function where the output is derived only from the input to the function. A stateful task does have a persistent state which can affect output or be affected by input to the task.

Sinks are inherently stateful, makes no sense in altering the values in the stream and then just discard.

Stateful tasks is hard to parallelize, only one physical worker should run a specific stateful task to avoid race-conditions and the need for synchronization. A stateful task may be distributed by data-parallelization, e.g. Flink shards stateful instances and make them data-parallel by splitting the key-space. But no two stateful instance’s key-space overlap. A stateful task which cannot be decomposed into data-parallel shards should therefore be made as effective as possible, making the required execution time to handle one input as short as possible.

Stateless task can be distributed quite arbitrarily on the other hand. Since stateless task only act as a data-transformer, there could be several workers performing the same stateless task without any need of synchronization between them. This can enable scale-out performance. Another way to improve performance would be to merge connected stateless tasks to reduce the impact of network latency on the overall performance.

**Spark:**

*Source: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*

Spark is an expressive distributed computing framework running on the Java virtual-machine (VM). Spark aims to improved performance by avoiding moving data eagerly to external stable storages. Other popular distributed computing frameworks prior to Spark moved intermediate result to an external stable storage, as a distributed database, to guarantee fault tolerance. This impedes their execution and decrease performance significantly. Spark introduced Resilient Distributed Datasets (RDDs) as an alternative. RDDs are abstract execution plans on data stored in stable storage. RDDs are lazily evaluated. No intermediate results are calculated before a result value from the expression the RDD represents is requested. The only exception to this is if the user explicitly instructs the intermediate result to be persisted to stable storage. At the time when the result value is requested, Spark will optimize the execution plan and distribute the workload over the cluster. The effects Spark had on performance was an improvement of great magnitude, up to 20x compared to Hadoop.

**Spark SQL:**

*Source: Spark SQL: Relational Data Processing in Spark*

Application analyzing Big Data tend to utilize a mixture of procedural algorithms and relational queries. Using Sparks RDDs or other distributed computing frameworks, as Hadoop, for relational queries can be tedious and complex. It requires a lot of manual optimization to match the performance of frameworks specialize for relational queries. To tackle this problem, Spark introduced the DataFrame concept along with an optimizer specifically developed for query optimization called Catalyst. DataFrames represents a distributed set of rows with a uniform schema. These rows can be manipulated using the DataFrame API, which consists of a set of lazily evaluated relational operators, similar to RDD transformations. A DataFrame can be transformed to an RDD of row objects. This bridge the gap between relational queries and procedural algorithms.

**Flare:**

*Source: Flare: Native Compilation for Heterogeneous Workloads in Apache Spark*

Sparks performance can suffer radically due to the intent to be expressive whilst keeping the distribution “under the hood”. Flare is an attempt to fix a part of the performance issue, focusing on DataFrames. Operations on a DataFrame will originally be executed on the Java VM. Running operations in a VM is inherently slower than native-code. Therefore, Flare transforms SQL queries on DataFrames to native code and improves performance significantly. Sparks query performance with Flare matched the performance of the best SQL engines without affecting Sparks expansive expressiveness. **(CDA is an attempt to make a complete system compile to native code, or at least as much as possible. Performance can probably be significantly improved doing this.)**

**Flink:**

*Sources: State Management in Apache Flink*

*Apache flink: Stream and batch processing in a single engine*

*https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/runtime.html*

Flink is an open source system focused on continuous distributed computing. Flink’s system will be scheduled once and long running. Therefore, it needs to be adaptive and fault tolerant. To achieve this, Flink employs a dynamic cluster architecture which is composed of three processes: the client, the Job Manager and the Task Managers.

The client will compile and optimize the logical pipeline before sending it to the Job Manager. The Job Manager is responsible for coordination, physical deployment and fault tolerance. The system can even handle Job Manger failures using leader election in ZooKeeper. The Task Managers are workers, using a single Java VM to execute tasks assigned to it.

The data-transmissions are performed in two ways, pipelined or blocking data exchange. Pipeline is the preferred method, since it enables concurrent execution of producer and consumer. When using pipeline, the consumer will continuously transmit partial results to the consumer during its execution. This reduce the memory requirements for buffers and also allows backpropagation of pressure, i.e. the producer will be notified if they produce results to slow.

During block data exchange, the producer will not send any intermediate results until all it’s input has been consumed. This separates the execution of producer and consumer into stages, and has a higher probability to spill to secondary storage due to larger memory requirements for buffers.

**Out-of-Order Processing**

*Source: Out-of-Order Processing: A New Architecture for High-Performance Stream Systems*

Causal ordering of all messages as a property in a stream processor seems theoretically desirable. It gives tasks the ability to incorporate ordering into the evaluation of input. Most importantly it makes it easier to track systemwide progress, but practical implementation of causal ordering of messages have large impediments. Streaming systems which enforce causal message order across all messages in the stream, called in-order processing (IOP), do so at a sever performance penalty. Firstly, IOP requires strict synchronization between streams in the pipeline and stragglers will have large effects on the performance. The stream’s performance will be proportional to the slowest worker, the stragglers. Secondly, IOP do not scale well in a distributed setting where different executing nodes have a large variance in computational power. Fast nodes will be stalled to wait for stragglers and large buffers will be needed whilst waiting for input from stragglers to buffer incoming input which arrive “too early”.

The problem with IOP, and also the basis for the solution, is the idea that causal ordering is required to guarantee stream progress. It seems hard to determine if a message got lost if messages are not guaranteed to arrive in the same order they were put into the stream. The “lost” message may arrive arbitrary late due to stragglers and network latency.

A solution to this problem is to enforce causal order of messages with respect to a special set of messages in the pipeline, called watermarks.

Watermark are simultaneously put into each input to the system. Each node in the graph will temporarily paus a specific input stream if a watermark arrives in it. When all input streams are paused, the node will forward the watermark to each of its output streams and reopen its input streams. When all input streams have sent a watermark it is guaranteed that all other messages sent prior the watermark has arrived and been processed by the node. Therefore, the stream is guaranteed to make progress and not loose messages. This technique is called out-of-order processing (OOP). OOP is more scalable than its counterpart IOP since it only requires occasional synchronization of streams. Most of the time OOP allow streams to flow according the performance capability of the producers.

**State Management in Flink:**

*Source: State Management in Apache Flink*

Flink utilize OOP to manage system states and achieve fault tolerance. Before sending watermarks to all output streams and opening up the node’s input streams, the node will save its current state to persistent storage. The states of all nodes for a specific low watermark, called epoch in Flink, will be a consistent snapshot of the system for that epoch. (**Describe consistent snapshot according to Lamport.)** This means that the system can guarantee correctness when it is re-allocated and resumed from the state stored in persistent memory.

Flink takes advantage of its data-parallel structure when creating snapshots and re-allocating from a previous snapshot. The state of a stateful data-parallel logical operator is split into partitions, each corresponding to a unique key-space which does not overlap with any other key-space. Together, all partitions covers the complete key-space for the logical operator. Each parallel physical instance of the logical operator will be assigned a set of these partitions. Upon re-allocation, the physical instance will only have to read the partitions assigned to it from storage. This is done to balance between unnecessary data-flow to and from the persistent storage and performance impediments due to lookup requirements for small key-spaces. The extremes are either that all key-value pairs for the logical instance would be sent to the physical worker, which then would filter out the ones which is part of its key-space. The other extreme would be that all key-value pair are stored independently in the persistent storage. The persistent storage would then have to iterate over all key-value pairs for the logical operator and extract the ones corresponding to the worker’s key-space before sending them.

**Weld:**

*Source: Weld Rethinking the Interface Between Data-Intensive Libraries*

Performance of data-intensive computations suffer heavily from data movement due to usage of different libraries within the same program. This is caused by the fact that compilers cannot optimize across library boundaries. Instead all intermediate results are forced to be calculated and stored in memory before passing control to an external library. Weld has shown that huge speedups in execution time can be achieved by addressing this problem. Weld implement a cross-library optimizer to enhance performance of data-intensive computations on bounded data. To enable the cross-library optimizations, each library must supply an IR representation of their program using Welds’ IR representation. This IR can be generated after the source code has been optimized to achieve domain specific optimizations, such as rearrangements of linear algebra operations. Weld then constructs a complete IR of the program containing all function calls across libraries. Then Weld optimizes the whole IR tree. The final resulting IR tree is then used to compile to hardware specific code.

Weld’s compile time, including optimization of the IR, ranged from 62-256ms. The mean compile time was 126ms and median 117ms. Since the target users will apply weld to programs analyzing huge datasets this compile time seems acceptable. The compilation time would be amortized constant time if Weld were applied to a long running stream processor.

Weld does not support asynchronous computations **(interesting in CDA?)**.

**Deep Embedding of DSLs:**

*Source: Modular Domain Specific Languages and Tools*

General-Purpose Languages are great tools for programmers who’s work spans over several application domains. But a General-Purpose Language might imped the development process for user which has limited experience with software development and who are specialized in a specific application domain. A more precise and narrow language may prove beneficial for such developers. Such languages are called domain specific language (DSL). A DSL is specifically developed for a target application domain. A good DSL should make the development process fast and efficient whilst still being correct and not limit the expressiveness within the application domain.

To develop such a DSL can become costly if it is done from scratch. To avoid this, embedded DSL takes advantage of the host language, the language the DSL interpreter is developed in. Much of the host language’s syntax, semantics, development tools and other related artifacts is reused by the DSL. This will reduce development cost of the DSL significantly and make it less error prone. Focus during the development of the DSL can instead be shifted from syntax to the semantics of the domain specific parts of the DSL. These semantics should be clear and capture the intuition behind the domain concepts to simplify the formal process of proving the correctness of a program.

**Lightweight Modular Staging:**

*Source: Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs*

If it is possible to represent the expressions of the DSLs as a data-structures, e.g. represent power expression as a power(base, power) structure, the embedded DSL can be reduced to a library for the host language. This technique is called lightweight modular staging (LMS). LMS tries to enclose domain specific expressions and stage them by representing the computations as a data-structure. The evaluation of the expression is postponed when using staging. This allows staged expressions of generic types. The compiler can then optimize the expressions when concrete types are supplied to the program if they are supplied prior to compile time. Else the staged expressions will have to be dynamically evaluated without static optimizations by the compiler. Hence, LMS acts as a program generator, generating runnable code from staged code.

Based on the staged representation of the code, domain specific optimizations may be applied. As an example, mul(var(label), con(1)) can be rewritten as var(label). This can be achieved using multi-methods for the staged representations, where pattern matching can be used to catch specific cases where domain specific optimizations may be applied. When using multi-methods for optimization there are three main problems which the developer of the DSL has to look out for: “separate type checking/compilation, ensure nonambiguity, and ensuring exhaustiveness”. *Using Rust traits, generic types, and trait objects will ensure type checking by the compiler and rust runtime checker. Unambiguity will be achieved by the order which traits are implemented, always falling back on a higher level implementation if the type for a specific trait implementation is not satisfied. Exhaustiveness is ensured since the trait will fall back to requiring definition of the trait by the user if no specific trait implementation for the types exists.*

Finally, to generate code each staged data-structure has to implement the Compile trait. Compile will recursively be called for all staged data-structures and generate code in the target language.

**LMS for CDA in Rust:**

**Trait TimeOrder which require function to set and get timestamp of generic type T. T must implement trait std::cmp::PartialOrd (gives <, <=, >=, >, ==).**

**Trait Window containing list of generic type Message, which has to implement TimeOrder. Window also has to implement TimeOrder. Generic type T for Window and Message has to be the same. List may be possible to extend to generic type as well.**

**Traits can be transformed to trait objects if type of items in the window can be known at compile time. Therefore, this would be transformed to dynamic staging but opens up for runtime errors and no static optimization by the compiler.**

**Optimization can for example be Map(f(T): T, Map(g(T): T, Input: T)) => Map(f(g(T)): T, Input: T)**

**Not sure if optimization can be done in Rust, may have to be done in Scala, and omit optimized but still staged code.**

**Language Virtualization:**

*Source: Language Virtualization for Heterogeneous Parallel Computing*

*Check sources [8,22]*

Modern software development, especially in the Big Data domain, become more focused on scale-out performance. This means that instead of improving the capacity of a single computer, scale-up, the program will run on a large set of possibly distributed computers. Coordination in a parallel distributed setting is non-trivial and error prone. Therefore, most developers turn to abstract models for parallel programming where the distribution is implemented implicitly by the framework **(references to Spark/Hadoop etc)**. This will work quite good on homogenous cluster, e.g. Flink and Spark which run on java VMs, but it will be inefficient on a heterogenous cluster.

To fully utilize the potential of such a cluster the code would need to be specifically tailored for the underlying hardware of each individual node. This is a real problem since vendors of accelerators, e.g. GPUs, tend to update and release new APIs which make more efficient use of their hardware. If an application used these APIs directly it would not be portable to new hardware. To make such an application portable to new hardware would require substantial continuous efforts by the application developer as long as new hardware and APIs are released.

Language Virtualization is a method to alleviate the application developer of the problem of continuous portability updates whilst optimizing the usage of the underlying heterogenous cluster. The DSL will first be transformed to an intermediate representation (IR). Aggressive domain specific optimizations can be applied to the IR. The resulting optimized IR will then be transformed to target code then compiled to native code.

The Liszt DSL **[reference]** was converted to an IR, optimized and then turned to either CUDA, nVidias API for GPUs, or MPI, message passing interface. This enabled Liszt to target a heterogenous cluster of GPUs and CPUs and better utilized the underlying hardware components.

**DSLs for distributed computing**

*Source: Lime: a Java-Compatible and Synthesizable Language for Heterogeneous Architectures*

Lime is another implemented language virtualization method. It introduces a DSL within Java with a set of language constructs which limits the usage of global fields or none final static fields and encourage usage of none mutable types. The constructs which enable these restrictions on methods are two keywords, local and global. A local method can only invoke other local methods while global methods may invoke either local or global methods. Global methods may not override or implement local methods but local can override and implement global functions. The local methods have the restriction that they may only access the instance’s local fields, but can be part of a stateful instance. The local method can use the instance’s local fields though Java’s “this” construct, with the restriction that it is not a static field. The two constructs enables Lime to represent a stream pipeline into a set of autonomous tasks, where each task is a set of local methods. Hence, these tasks are guaranteed to not be in need of any synchronization. These tasks can therefore be dynamically allocated to hardware components in the cluster. No distributed coordination aside from input-output pipelining need to be emplaced between these tasks. To further decrease network traffic, connected tasks in the pipeline can be merged and assigned to a single node in the cluster. This also enables the compiler to distribute the workload more evenly over the cluster. A running graph may be dynamically extended with new task. Added task will automatically be connected and started. Explicit loops in the dataflow-graph are illegal, such a graph will throw an exception when run. Feedback is instead possible through messaging techniques between tasks.

**(Does Lime allow user to develop applications which target specific hardware? E.g. tensorflow on GPU.)**

**Ziria:**

The implementation of wireless protocol is none trivial due to the high throughput demands. Data in the rate of Gigabits/second have to be processed by a wireless router. Most protocols have therefore been implemented without much hardware abstractions. This enhances performance, since the programs are tailormade for the underlying hardware, but it makes the software development process more complex and error prone. Ziria is an DSL which has been implemented to enhance development and maintenance of wireless systems. It consists of two main components: stream transformer and stream computer.

The transformer takes an input stream, transform the data, and passes the transformed data to an output stream. The computer is similar, but besides consuming the input and emitting output, it produces another value called control value. The workflow of a stream computer is as follows, it continuously consumes input and produces output which is passed downstream. After a while the computer halts and emits the control value. The workflow of the pipeline is not delayed by the computers since they continuously emit output when calculating the control value. The control value can be metadata about the input stream. Ziria can use this to reconfigure the whole pipeline downstream. An example from the paper is a payload header decoder, where the computer emits a control value corresponding to which decoder should be used for the payload body. The rest of the downstream pipeline can then dynamically be initialized as the required decoder for the current input in the stream. **This might be interesting for CDA, where the stream processing pipeline may change depending on the input data.**