**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 1:**

Construct a DSL which can describe a stream processor and be converted into a RustAST

**Proposal 2:**

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

**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) **[need source]**. 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.

**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 in 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.

**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.**