MASTER'S THESIS | LUND UNIVERSITY 2018

# High-performance signal processing for digital AESA-radar

Alexander Olsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-41

# High-performance signal processing for digital AESA-radar

Alexander Olsson
dat12aol@student.lu.se

September 24, 2018

Master's thesis work carried out at Saab AB.

Supervisors: Jörn Janneck, Jorn.Janneck@cs.lth.se
Anders Åhlander, Anders.Ahlander@saabgroup.com

Examiner: Flavius Gruian, Flavius.Gruian@cs.lth.se

## Abstract

The demands on signal processing in digital AESA(Active Electonically Scanned Array)-systems is rising rapidly due to an increase in both digitization and number of antenna elements. The computational performance and data throughput requirements for the next generations signal processing in digital AESA-systems will be massive. Additionally it must be possible to handle the high complexity of new advanced applications. It is not only tough demands on performance, but also on how to improve the engineering efficiency. There is thus a need for a high-performance signal processing architecture with a stable and engineering efficient API.

Stream processing is a decades old computing paradigm that is just now beginning to mature. Big data, autonomous driving and machine learning are all driving the development of low-latency, high-throughput stream processing frameworks that work in real time. Data and computations are abstracted to streams that flow from node to node where operations are performed.

This project aims to explore the possibility of utilizing this emerging computational trend to meet the signal processing demands of the future. It will examine several stream processing frameworks, among them Apache's Flink, the open source C++ template library RaftLib and Google's TensorFlow.

The two main candidates, Flink and RaftLib are used to implement several use cases to test certain requirements. The results show that both frameworks parallelize tasks well, with RaftLib performing parallel matrix multiplication operations nearly as fast as a custom written thread-pool style benchmark. Flink is faster still, beating the benchmark in execution time. Flink also performs well compared to RaftLib when stream latency is examined. Furthermore, both frameworks contribute towards engineering efficiency, Flink having a richer API, as well as a larger organization and more active community than RaftLib.



**Keywords**: Signal processing, stream processing, high-complexity, high-throughput, low-latency, Flink, RaftLib, TensorFlow

# Acknowledgements

This project was suggested and supported by Saab Surveillance, Airborn. Chalmers University of Technology (CTH) and Lund University (LTH) accepted and supported the project.

First of all I would like to thank Henrik Möller from Chalmers University of Technology, for a successful collaboration which made the project both more fun and lighter. I would like to thank our supervisors, Jörn Janneck at LTH, and Vincenzo Gulisano at CTH for helpful input and advice during the project. A big thank you to Anders Åhlander, our supervisor at Saab which have helped me from the start, by teaching out the basics of signal processing and giving me daily input and essential guidance. I would also like to thank the members of the reference group. The bi-weekly meetings with the group was very helpful and pointed us towards the right direction. I especially thank Håkan Carlsson for finding and setting up our testing environment, and to Per Ericsson for help and ideas regarding our use cases. A final special thank you to Jonas Lindgren, for your support and enthusiasm, and for efficiently shielding us from all the things administrative.

# Contents

# Division of work

This master thesis was conducted at Saab Surveillance, Airborn, together with Henrik Möller from Chalmers University of Technology. Due to being from different universities, two separate reports have been written to fulfill each schools requirements for a master degree. Since a common report was written at Saab, that thesis have been rewritten. This report is written to not overlap Henrik's report more than 40% of the content. However, since the original report was written together similarities of the reports' structure and content will arise. During the thesis, both have striven to divide all work, including report writing, equally.

## Division of work - report writing

All images, except for Figure 1.1, in the report have been created by Alexander. All tables, except Table 4.1, have been created by Henrik. Though both have been part of criticism and feedback in both images and tables.

The abstract was mostly written by Henrik, with modifications from Alexander.

The first part of the *Introduction* has been rewritten in both reports, to be as unique and personal as possible. The subsection covering *Background & contribution* is mostly written by Alexander and the *Stream processing* subsection is written by Henrik. The *AESA signal processing* subsection is written by Alexander.

The *Related work* section is completely written by Alexander.

The *Approach* section was developed quite evenly. The subsections written fully or modified by Alexander are: the "introduction" area of the *Approach*, Section 3.2, Section 3.3 & Section 3.5.2. In Section 3.6 the parts describing how the use cases was implemented in the benchmark, RaftLib and Flink is written by Alexander. The subsections that have been written by Henrik and not modified are: Section 3.1, Section 3.6.1 & Section 3.6.1. The rest subsections of the *Approach* have been written and modified by both.

In the *Stream processing frameworks* section, Henrik have written the section about RaftLib and the first paragraph on TensorFlow. Alexander have written the Flink section and the section covering rejected frameworks, (apart from the first paragraph covering TensorFlow).

The *Results* section is mostly written by Henrik part from the subsection concerning stability, which is written by Alexander. The *Discussion*, *Conclusions* and *Future work* sections have been rewritten in this report to more reflect the conclusions of Alexander.

# Division of work - development and implementation

Most of the work has been discussed and completed together, with some exceptions. The installation of RaftLib and Flink was done by Henrik. The installation and testing of Wallaroo was done by Alexander, and the installation and testing of TensorFlow was done together, but with more focus from Henrik's side. The testing of RaftLib and Flink was done together.

The metrics were produced together in the start of the project, but updated during the project. The use cases and requirements were also developed together and modified through out the project. The implementation of the benchmark was mostly done by Alexander, and the implementation of all the use cases was done together or split evenly among us. In the beginning of the project both searched for different ways to handle signal processing chains, and together a decision was made to look more into complete frameworks within stream processing.

# Chapter 1
# Introduction

The complexity for the next generation of AESA (Active Electronically Scanned Array) sensor systems is increasing rapidly, together with the demand for high throughput and low latency. The future AESA sensor systems will have thousands of individual antenna elements, signals from each element can by, interference patterns, be directed in different angles and frequencies. Clusters of these antenna elements can be joined together, resulting in several separate radar arrays creating multiple parallel signal processing chains.

Digital signal processing algorithms requires lots of mathematical computations to be performed quickly and repeatedly over multiple samples of data. If the latency and throughput are not good enough, the information gathered will be obsolete. For example the computational complexity of the STAP (Space-Time Adaptive Processing) algorithm is provisioned to the number of antenna elements cubed.

Input data received from the AESA-radar arrives in real time and needs to be processed and parallelized as soon as possible among the available cores in a computer cluster. There are several approaches to deal with the parallelization process among cores, e.g. by implementing everything manually using POSIX threads or by using OpenMP. This thesis examines complete frameworks within a computer paradigm called stream processing. These frameworks aid with the parallelization and strafe towards increasing engineering efficiency.

Stream processing is a computer programming paradigm developed to simplify parallel computing [4]. This is done by reducing programmers' direct interaction with allocation, synchronization and communication between the units in a computer cluster and make it more automatic. Stream processing is a very old procedure that has grown in popularity in the last years. A key feature of stream processing is that it does not wait for input to be "complete". As soon as it can do something with the data it starts processing it [2].

The focus for this thesis is how complete frameworks within stream processing can contribute towards both performance and engineering efficiency when processing digital signal processing chains.

The sections in Chapter 1 describes stream processing, AESA signal processing as well as the frameworks that were chosen to proceed with testing and evaluation, it also mentions the frameworks that did not meet set requirements for continued studies. Related work is presented in Chapter 2 while the approach of this study is explained in Chapter 3. Chapter 4 and Chapter 5 presents the results and discussion, respectively, of the tested frameworks. A conclusion of the study and ideas for possible future work are presented in Chapter 6 and Chapter 7.

## 1.1 Background & contribution

At first, the objective was to study different combinations of signal processing architectures. I.e. test and evaluate different setups of parallel hardware, programming languages, resource and communication models, etc., to see what different setups in architecture would be more optimal in processing digital signal processing chains, also to get a better perception on the next step within digital signal processing. The different combinations and choices of setup would be decided by doing an extensive literature study of viable options.

This objective was too broad, which resulted in narrowing the scope to focus more on complete frameworks which could be used to realize signal processing chains. These frameworks have several automated steps for the parallelization which should ease the workload and increase engineering efficiency. The frameworks under consideration are part of a computer paradigm called stream processing.

The scientific assignment became to see how stream processing frameworks could handle the typical signal processing chains of the AESA-radar: the high-complexity, high-throughput and low-latency. As well as to see how well the frameworks contribute towards engineering efficiency, e.g. how they simplify the parallelization and load balancing process over available cores. Are the streaming frameworks more engineer efficient than traditional solutions? Meaning, is it easier to use and develop applications within the frameworks than in a traditional language such as C++. Also, how well do the streaming frameworks support the ”x-abilities” for engineering efficiency, i.e. scalability, usability, sustainability and portability.

The best path forward will be defined not only in terms of current research but also taking future platform development and research efforts into account. This is a necessity since the lifespan of a typical radar application is 15-20 years, compared with the lifespan of technology where 15-20 years translates to several generations [1].

This thesis will investigate how complete frameworks within stream processing can contribute towards both performance and engineering efficiency requirements within the next-

generation signal processing development for digital AESA-radars.

## 1.2 Stream processing

Stream processing is a computing paradigm that has been developed since at least 1974 [13], then under then name data flow procedure language. It has since been known as dataflow programming and event stream processing. In stream processing an application is seen as compute kernels, or operations, connected by streams of data. The streams are links between kernels, and are implemented as first-in-first-out (FIFO) queues [6]. Data is processed in compute kernels as soon as it is made available over any incoming streams. Figure 1.1 shows a simple multiplication application that multiplies pairs of numbers from two incoming streams to produce a stream of resulting products.

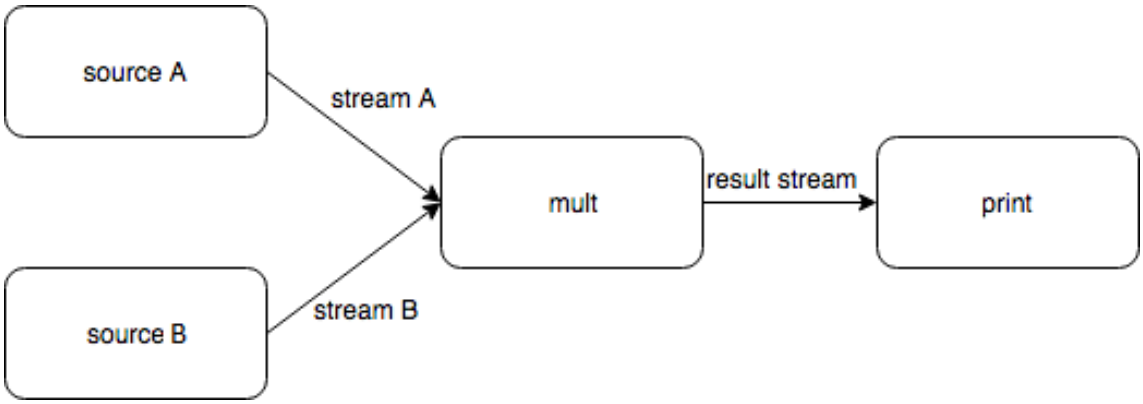



**Figure 1.1:** Simple streaming multiplication application.

A prominent feature of stream processing is the lack of kept state between streams[6]. In other words compute kernels compartmentalize states.

### 1.2.1 Moving data

With the speed of modern CPUs, the most costly part of most applications is not computation, but moving data [6]. This is a problem when parallelizing any task and using stream processing does not inherently solve this problem. The application in Figure 1 would most likely incur a heavy overhead since it is moving many small data units between a relatively large number of compute kernels, i.e. for this specific example it would probably better to merge the last two kernels together for increased performance. Decomposing the streaming graph and increasing the size of transmitted data often decrease the execution time of streaming applications [6].

The problem of moving data means that efficient queues are of outmost importance for any streaming framework. An important factor of any queue is size; if too small bottlenecks are created, then to big system resources are wasted. Furthermore the demand placed on individual queues can vary over the lifetime of an application. Therefore dynamic optimization of queues is important aspects of modern streaming frameworks [6].

### 1.2.2 Hybrid computing

A feature of stream processing is the ability to support so called hybrid computing. Hybrid computing, or heterogeneous computing, means that the capabilities of different computational units, such as CPUs, GPUs, FPGAs etc., can be leveraged within the same framework, with minimal effort from the developer [6]. GPUs and FPGAs two types of accelerators, special purpose processors designed to increase the performance and speed up compute-intensive sections of applications. The FPGAs are highly customizable and GPUs provide massive parallel execution resources and high memory brandwidth [12].

If a framework is aware of the available hardware, and a compute kernel implementation is created that is able to run on a specific computational unit, the framework can schedule the implementation to execute on that unit, increasing performance drastically.

## 1.3 AESA signal processing

An AESA-radar consist of several "antenna elements" or "cells", which can work as their own identical radar but can also be grouped together into a cluster of several cells. These can be directed in different directions to keep track of several orientations in parallel. Every antenna element produces input as streams of complex samples to the signal processing step at regular intervals, normally 10-20 ms in a so called integration interval (INTI) [14]. The input data are organized as "cubes", taking three dimensions in consideration, i.e. the AESA antenna element, pulse and range bin. A pulse is being sent every Pulse Repetition Interval (PRI) to get an echo from a reflected object, and the range bin is representing the range between the radar and reflective object [14]. This indata is processed in a number of steps:

- Digital Beam Forming (DBF)
- Doppler Filter Bank (DFB)
- Pulse Compression including an envelope detection (PC)
- Constant False Alarm Ratio (CFAR)
- Integrator (INT)

An illustration of a simplified radar signal processing chain can be viewed in Figure 1.2.

Several receiver beams are created simultaneously from the DBF step. The main operation type for this step is vector by matrix multiplication [14]. The next step, DFB, gives an estimation of the target's speed relative to the radar, as well as giving improved signal-to-noise ratio due to coherent integration of indata. The pulse bins in the data set are transformed into Doppler channels. The main operation type here is Fast Fourier Transform (FFT) [14]. The goal of the pulse compression is to collect all received energy from one target into a single range bin. This received echo is first digitally passed through a FIR filter, to then calculate the absolute values of the digital samples. The data is real after this step. The main operation type for this step is FIR filtering, all steps before have handled

complex data, after PC the data is real. The CFAR processing is done to minimize the number of false targets while also maintaining a best possible sensitivity. The main operation type is multiplications and additions of real data. The last step, the integrator, is an n-tap unsigned integrator that integrates channel orientated video. Each Doppler channel, range bin and antenna element shall be integrated over a number of pulses [14].

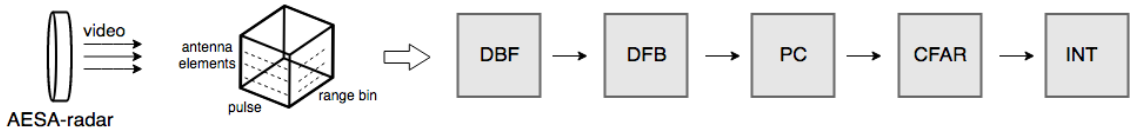



**Figure 1.2:** A simplified radar signal processing chain used to illustrate the calculations of interest.

## 1.4 Stream processing frameworks

### 1.4.1 RaftLib

RaftLib is a stream processing framework written as a C++ template library [10]. This means it can natively interface with legacy C++ code and fully utilize the C++ language. Everything that can be done in C++ can be done in RaftLib. What RaftLib aims to introduce is the stream processing paradigm, with automatic parallelization and dynamic queue optimization through low overhead real-time monitoring.

RaftLib was first presented in 2015 by Beard, Li and Chamberlain at the Washington University in St. Louis [10]. Since then, Beard has been the primary developer.

#### RaftLib and stream processing

RaftLib builds stream processing graphs by linking *compute kernels* with *ports* [10]. RaftLib compute kernels are C++ classes that extend the raft::kernel class, and ports (input and/or output) are defined in the constructor of each kernel class. Each such kernel also implements a run-time method where stream operations are performed. The graph is constructed in the main method, where declared kernels are linked by connecting ports using link operators [9]. These operators describe how two or more ports are connected, i.e. dynamically, static pipeline, static split/join, and so on. Table 1.1 lists these operators.

A RaftLib stream (output-input port pair) is implemented as a first in-first out (FIFO) queue [10]. That is to say, data sent over a single stream is guaranteed to arrive in order. No such guarantees are made for data sent over several streams.

The size of the FIFO queues can have drastic effect on performance. Performance drops drastically when they are smaller than 80 kB [10]. For sizes larger than 8 MB performance also slowly decreases.

Each compute kernel keeps state internally, but state cannot be kept between kernels. This is a prominent feature of stream processing and enables much simplified parallelization [10].

| Name | Symbol | Description |
|---|---|---|
| Basic Link Operator | a >> b | Links two compute kernels. a and b are assumed to have a single output and input port respectively, and the ports are of the same type. |
| Chain Link Operator | a >> b >> c | Extension of basic link operator. Can be extended N times. |
| Named Port Link Operator | a["out"] >>b["in"] | Used when kernels have more than one input and output port. Explicitly links two ports of the same type. |
| Dual Named Link Operator Chain | a["out"] >> b["in"]["out"] >> c["in"] | Two sets of brackets can also be used to explicitly link specific ports in a larger chain. |
| Static Split-Join Operator | a <= b >= c | The split operator <= maps N output ports of a to N duplicates of b. Conversely, the join operator >= maps N input ports of c to the N duplicates of b. |
| Static Split-Join to Kernel Set | a<= raft::kset(b,c) >= d | If the port types are not the same, a kernel set can be used to statically split and join kernels. |
| Out-of-Order Stream Modifier | a >> raft::order::out >> b | Informs the run-time that the order of data on the link is unimportant. The run-time is then free to duplicate a and b. |

**Table 1.1:** Kernel link modifiers in RaftLib

## Moving data

In RaftLib there are many options for moving data between compute kernels. The process involves two steps: writing data on output ports and reading data on input ports. Once a kernel writes data, it is gone and released to the downstream kernel.

There are two basic types of send/receive methods in RaftLib: zero copy methods and non-zero copy methods [7]. Zero copy means that data is not moved from the kernel context into the application context[37]. This eliminates unnecessary context switching and

shuffling of data from buffers controlled by the kernel to buffers controlled by the application. However, this is not always preferable since the number of instructions required might be higher [7]. For smaller transfers, such as simple types or data that fits within a single cache line, the RaftLib copy methods can be faster.

The zero copy methods are *allocate()* and *allocate_s()* [7]. allocate() returns a reference to a writeable position at the tail of the FIFO. allocate() can also be called on an object type, in which case the function will call the constructor and initialize any internal data types and structures. Once something is written a call to send() must be made to release it to the FIFO. A *deallocate()* function exists if a user needs to deallocate the memory. allocate_s() returns an object instead of a reference. The user must dereference the object to access the memory. There is no need to call send() since the object is released to the FIFO as it exits its scope.

The copy methods are *push()* and *insert()* [7]. push() makes a copy of any object it is called on using the object constructor, and passes it to the FIFO along with a signal that is guaranteed to arrive at the same time as the object. To transfer a C++ container, insert() can be used to transfer the range of contained items downstream. It uses C++ iterator semantics and takes an iterator to the start of the range and an iterator to the end of the range as parameters. If the range is greater than the available space in the FIFO it will block and add items as space is made available.

On the other end there are also copy and zero copy methods to retrieve data from the stream [7]. The zero copy methods are *peek()* and *peek_range()*. peek() returns a reference to the head of the queue. Since the stream automatically optimizes itself a subsequent call to *unpeek()* can be made to inform the run-time that the reference is no longer used, and to keep the memory in a valid state. This means that RaftLib does not release the memory location. Thus any subsequent call to peek will return a reference to the same object. This can be used to keep state within a kernel. When data on the input port has been peeked, *recycle()* can be called to free memory on the queue. It can also be used before calling peek() in order skip items altogether. peek_range() is equivalent to peek() but used to access a range of items from the stream. This range is often termed window in stream processing.

The copy return methods are *pop()* and *pop_range()* [7]. Once pop() is called the memory at the head of the queue belongs to the user and no further steps are required. Analogous to peek_range(), pop_range() pops a range of items and places them in a std::vector, passed by the user as a parameter. The fifo must be pre-allocated to the correct size. As in std::vector x( size ).

## Adaptability

RaftLib claims to be an “online auto-tuned streaming system”, utilizing an array of techniques to adaptively optimize itself during run-time [6]. When optimizing queues it uses models that suit certain environments, and it utilizes the machine learning mechanisms Support Vector Machine (SVM) and Artificial Neural Networks (ANN) to detect patterns

in usage environment and pick the correct model cite [6]. It also has low-overhead instrumentation that dynamically monitors queue occupancy, non-blocking service rate, utilization etc.

Moreover, RaftLib can also automatically parallelize streams set by the user as "out-of-order" streams [6]. The framework will analyze the graph, detect these streams and insert split and reduce link operators where needed. Round-robin and least-utilized strategies are used to direct data to suitable kernels.

## Hybrid computing

RaftLib uses the Portable Hardware Locality (hwloc) library to create a hierarchical topology abstraction of the host system [3, 33]. This includes other system attributes such as layout of memory, I/O devices and sockets. Depending on the implementation specifics of the compute kernels, RaftLib claims to be able to schedule threads on appropriate computational units.

## Example application

A simple example application can be found in Appendix B. In this example there are three compute kernel classes: a producer, processor and consumer. The producer reads pre-generated integer matrices from memory and sends it to the processor compute kernel. The processor performs a matrix multiplication on this matrix. The processor then sends the result to the consumer.

The producer and consumer are implemented as *parallel_k kernels*, meaning the number of output or input ports can be set by parameters (nr_ports in this example). These ports are declared in the class constructor, and are connected to data to be received or sent in the kernels run-time methods, named *run()*. The processor kernel consist of two constructors, one class constructor declaring one of each ports(input/output ports), as well as a copy constructor. The processor kernel uses the CLONE() macro, which makes it possible for the run-time to create as many processor kernels as there are input and output ports on the connected producer and consumer kernels. The CLONE() macro inserts C++ code within each compilation unit so the kernel can be cloned by the run-time without knowing specifically what the original arguments were, or even have a specific type information about it [8].

In the main method the matrices are pre-generated and stored in file, to be identical each repetitive run. The three classes are then declared, and the graph is declared and added to a raft::map container function. The graph is constructed using the link operators listed in Table 1.1, specifically the static split and static join operators. The producer is connected to the processor using static split. Conversely, the producer is connected to the consumer using static join. When the container function is executed by running m.exe() in the example, the graph is created, the compute kernels are instantiated and their run-time functions are executed.

### 1.4.2 Flink

#### General

Apache Flink, created by the Apache Software Foundation[20], is an open-source stream processing framework using Java & Scala as development languages and run in JVM (Java Virtual Machine). It is used for distributed and high-performing data streaming applications. Flink is quite massive and builds on the philosophy that many types of data processing applications, e.g. real-time analytics, continuous data pipelines, historic data processing (batch), and iterative algorithms (machine learning, graph analysis) can be expressed and executed as pipelined fault-tolerant dataflows [11].

At basic level, a program in Flink is made up of three steps: a data source, a transformation step and a data sink. An image on the simple pipeline can be seen in Figure 1.3.


**Figure 1.3:** Image describing a basic stream program in Flink, where the arrows are the intermediate streams and the blocks are the operators.

#### System architecture

Flink is divided into three levels, each level is focused on a certain development area, i.e. streaming/batch applications [18].

The first being a lower level of abstraction that offers stateful streaming. This lower level abstraction is usually not needed by most applications since these applications would mostly program against the Core API's, i.e. the DataStream API and the DataSet API [18]. The Datastream API handles bounded/unbounded streams while the DataSet API handles bounded data sets. These API's offers the building blocks for data processing, i.e. various forms of user-specified transformations, joins, aggregations etc.

One level above the Core API's is the Table API [18], not as expressive as the Core APIs even though it's extensible by user-defined functions. The Table API is a declarative DSL(Domain Specific Language) centered on tables, and follows the extended relational model. Tables have a schema attached which is similar to tables in relational databases and that API consists of similar operations.

According to [18] it's easy for programs to mix tables and the DataStreams/DataSets and convert back and forth.

The top level of abstraction in Flink is SQL. It represents programs as SQL query expressions, which is similar to the Table API both in semantics and expressiveness [18]. SQL queries can be executed over tables defined in the Table API. The focused API in this thesis is the DataStream API.

The process model in Flink comprises three types of processes: the client, the JobManager and at least one TaskManager [11]. The transformation from program code to the actual dataflow graph is done by the client. This dataflow graph is submitted to the JobManager. Under this transformation phase, the data types (schema) of the data exchanged between operators is examined and creates serializers and other type/schema specific code. DataSet programs do an additional cost-based query optimization phase.

The JobManager coordinates the distributed execution of the dataflow, while the actual data processing takes place in the TaskManages. A TaskManager executes one or more operators that create streams, and keeps the JobManager updated on their status [11].

## Streaming dataflows

All programs in Flink, regardless of API the program is written in [11], compile down to a dataflow graph representation. The dataflow graph is executed by Flink's run-time engine, the common layer under the DataSet- and DataStream APIs.

The core abstraction for data-exchange between operators is done by intermediate data streams. They represent a logical handle to the data that is produced by an operator and consumed by at least another operator downstream. The intermediate streams are logical in the sense that the data they point to may or may not be materialized on disk [11]. The pipelined intermediate data streams transports data between simultaneously running producers and consumers resulting in pipelined execution. It's in order to avoid materialization that Flink uses pipelined streams for continuous streaming programs [11]. The transformations in Flink can be represented by nodes and the streams by edges in a graph. A stream is an infinite flow of some recorded data. A transformation takes one or more streams as input, modifies or does something with it and produces one or more output streams. An overview of a streaming dataflow can be seen in Figure 1.3.

When running distributed execution, Flink chains operator subtasks together into tasks [19]. Each task is processed by one thread. Chaining the subtasks together into tasks reduces the overhead of thread-to-thread handover and buffering, overall throughput increases as well as decreasing latency, a useful optimization.

Programs are inherently parallel and distributed within Flink. A stream can be parallelized by having more than one stream partition during run-time, and each operator by having more than one operator subtask [18]. Each operator subtask are independent of one another and will execute in different threads or on different machines. So the parallelism of a particular operator is the number of operator subtasks. The parallelism of a stream is always that of its producing operator. In a program, the level of parallelism of different

operators can vary.

There are two ways for streams to transport data between two operators, either in one-to-one pattern or in a redistributing pattern [18]. The one-to-one streams maintain the partitioning and the order of the elements, i.e. the receiving operator gets the elements in the same order as the sending operator. The redistributing streams partitions the data over several target operator subtasks but the parallelism introduces non-deterministic ordering in which the results for different subtasks of an operator arrive.

### Windows

Since aggregating events is not the same in batch and stream processing Flink uses so called windows when processing unbounded and infinite data sets [18]. Windows scope up a set of incoming data and processes it with a given function, e.g. counts, sums, etc. A window scope can be set in different ways, for example by using time, i.e. gather all data every fifth second, or by element count, i.e. every tenth element [38]. There are different types of windows, tumbling windows, sliding windows, session windows and global windows.

### State backends

Flink provides different ways of memory management depending on the core API used [30]. The memory management for the DataSet API is based on a research project which had the goal to combine the best technologies of MapReduce-based systems and parallel database systems. A good description for this can be found at [39]. Since this thesis is more about the streaming environment the states used in the streaming API will be presented.

The DataStream API uses different so called state backends instead of the memory management for batch processing. There are different state backends which decide how data is being stored in a program [19, 22]. States in the DataStream API can be hold in various forms; windows gather or aggregates until they are triggered, transformation functions may use a key/value state interface and may also implement a CheckpointedFunction interface to make their local variables fault-tolerant. When "checkpointing", state are persisted upon checkpoints to guard against data loss and recover consistently. It is the different state backends that decide how the state is represented internally, and how and where it is persisted upon checkpoints.

The default state backend is the MemoryStateBackend, which is used if nothing else is configured. The other state backends are FsStateBackend and RocksDBStateBackend [22].

The MemoryStateBackend stores data internally on the Java heap [22]. Key/value state and window operators hold hash tables that store the different values, triggers, etc. According to the documentation on state backends of Flink, it is encouraged to use the MemoryStateBackend on local development and debugging, and on jobs that hold little state, e.g. jobs that consist only of record-at-a-time functions (Map, FlatMap, Filter, etc.) [22].

The FsStateBackend holds in-flight-data in the TaskManager's memory, and is configured with a file system URL (type, address and path) [22]. Upon checkpointing it writes the state-snapshots to files in the configured filesystem and directory. The JobManager stores minimal metadata in its memory. The FsStateBackend is encouraged for jobs with larger state, long windows and large key/value states, as well as all high-availability setups [22].

The RocksDBStateBackend is configured with the same file system URL as the FsStateBackend [22]. It holds in-flight-data in a RocksDB database [34] that is (per default) stored in the data directories of the TaskManager. The whole RocksDB database will be checkpointed into the configured file system and directory upon checkpointing, with minimal metadata stored in the memory of the JobManager. This state backend is encouraged for jobs with a very large state, long windows and large key/value states, and all high-available setups [22]. The maximum throughput of this state backend will be lower than the others. However, since the amount of state that can be kept is only limited to the available disk space, allows storing very large states compared to the FsStateBackend which stores them in memory.

## Dynamic Scaling

From Flink 1.2.0, Flink supports dynamic scaling to some extent [29]. It supports changing the parallelism of a streaming job by using savpoints. This is done by rebooting and then restoring the job from a savepoint with a different parallelism than the original. In the *StreamExecutionEnvironment* users can set a new per-job configuration parameter called "max parallelism". It determines the upper limit for the parallelism.

In Flink 1.5 release announcement [25], it is mentioned that improvements have been done which support dynamic resource allocation and dynamic releases on YARN [15] and Mesos [21] schedulers for better resource utilization, failure recovery, and also dynamic scaling.

## Example program

An example program in Flink can be seen in Appendix C, which is a simple matrix multiplication performed over a streaming environment. It contains a main-method that starts with getting the *StreamExecutionEnvironment*, the context in which the program is executed. It provides methods to control the job execution, such as deciding parallelism. The second thing that is initiated is a *DataStream* that receives matrices from a user-defined source, multiplies the matrices with themselves using the *Map* function provided by the Flink API. Then sends the matrices downstream to a sink, which prints out the result based on the *Matrix* object's *toString()* function. A parallelism of four is set on the stream in this case, meaning four "matrixStreams" will be processing matrices in parallel.

The source is constructed by implementing Flink's *SourceFunction* overriding a *run()* and a *cancel()* method. In this case the run-method first reads matrices from a file and then stores them in an *ArrayList*. After this, it iterates over the *ArrayList* containing the ma-

trices and sends them downstream and automatically stops when the end of the array is reached. The cancel-method is not used in this example.

### 1.4.3 Rejected candidates

Several interesting frameworks have been inspected to see which might suit for processing signal processing chains. Due to time limitations some frameworks have been studied in a hastier manner to get an overview of the functionality to evaluate, whether to keep testing it or not. Others have been tested more thoroughly to later be discarded due to missing key functionality.

#### TensorFlow

Of special interest is the Google machine learning framework TensorFlow [23]. It creates a dataflow graph and maps computations onto different hardware in a heterogeneous manner. These operations are often in the form of linear algebra computations on matrix generalizations called tensors. Tensors are typed matrix generalizations that can express matrices of arbitrary dimensions. TensorFlow operations are abstractions of different computations such as matrix multiply, or add. They can be made polymorphic over different tensor element types through the use of operator attributes, which must be provided or inferred as the graph is constructed. In TensorFlow, kernels are implementations of operations that run on specific types of computational units. In other words, kernels give TensorFlow heterogenous computing capabilities.

TensorFlow as a framework differs from RaftLib and Flink in the way that it is not originally a framework for stream processing, but for machine learning. The feeling we got from TensorFlow when it came to the "automatic" parallelization was that it does not support it, i.e. it had to be implemented in a more manual way. Even though it seemed that TensorFlow could support streams to some extent it was discarded as a candidate framework.

#### WallarooLabs' Wallaroo

Wallaroo is a framework developed by WallarooLabs Inc. [26] and uses Python or Go as development languages. This framework seemed interesting for many reasons. By implementing in a very high-level language like Python it could increase engineering efficiency, and according to the description of Wallaroo it is possible to scale without adding extra or changing code. The perception received from testing and running simple programs was that this scaling was handled by issuing terminal commands.

We decided to leave Wallaroo, and move on with Flink and RaftLib for mainly three reasons. First of we thought that how the construction of the topology was a little to complex and not so smooth. Secondly, the terminal commands were very long, and a little confusing at first glance, and lastly, a lack of time. We would not have the time to test more than two frameworks, and we thought of RaftLib and Flink as better test candidates.

Wallaroo can however be an interesting framework to look more into in the future.

## Apache Storm & Spark

These frameworks would be interesting to test but were discarded as test candidates due to two reasons. One being that Flink the latest stream processing framework from Apache, is based on both Spark [16] and Storm [17]. Flink is also backwards compatible with Storm [35]. Another reason was that in [28] they compare all three against each other, showing that Flink outperforms both of the other two.

## TrillDSP

TrillDSP is a framework based on Microsofts Trill which is developed to handle digital signal processing chains. Due to only finding an article [31] and a paper [32] of the framework, and not a Github repository or guidelines on how to run it, it was discarded as a test candidate.

# Chapter 2
# Related work

## 2.1 Engineer efficient framework solutions

A previous study on how to improve engineering efficiency, and lower the development cost when working with advanced applications within parallel signal processing, was done by Anders Åhlander et al. in [1]. It is also discussed how the "x-abilities" like sustainability and usability is of importance to the development tool. I.e. the application used for development must have a lifetime of several years while also being flexible, to be able to handle various application implementations, testing and deployment. It is also said that the development application should accommodate several technology cycles, preferably be hardware independent.

In [1] they also mention that by using more engineer efficient development tools, platforms in this paper, results in several benefitting qualities, being:

- The possibility to take advantage of the rapid technology development due to shortening the development time of the application.
- The availability of multiple implementation options for any given application.
- Scalability in terms of problem size as well as technology development.

Here a platform called GEPARD is described and used to illustrate a good, engineering efficient solution for advanced signal processing. The platform is tested by running two different types of signal processing applications, one compute-intensive STAP and one data-intensive SAR, (synthetic-aperture radar).

### Comparison of stream processing frameworks

A study which compares Big Data Stream Processing frameworks has been done by Ziya Karakaya et al. in [28]. The frameworks they compare are the three latest Apache stream

processing frameworks; Spark, Storm and Flink.

They compare the performance of frameworks as well as their scalability and resource usage towards varying number of cluster sizes. The comparison involved optimizing all the frameworks to their ideal performance using Yahoo Streaming Benchmark.

Summarized, Flink outperforms both Spark and Storm under equal constraints. Spark could however be optimized to provide a higher throughput than Flink with the cost of higher latency. The results from this comparison had an compelling impact when choosing candidate frameworks in Section 1.4.

## 2.2 Programmable stream processors and stream processing languages

To be able to achieve the required computation rates of tens to hundreds of billions computations per second in e.g. signal processing, current media processors use special-purpose architectures tailored to one specific application. These processors requires a significant design effort and are hard to change when algorithms and applications evolve. A paper about programmable stream processors and stream architecture is mentioned and described in [27] by William J. Dally et al. This is a very interesting field within the stream processing paradigm which can be another option to frameworks on how to increase performance and flexibility within signal processing.

According to [27] there is a demand for flexibility within media processing, which therefore motivates the use of programmable processors. There are however very large-scale integration constraints which limit the performance of traditional programmable architectures. Within modern VLSI technology, it is not the computations that are expensive, but the delivery of instructions and data to the ALUs that is the bigger cost. Special purposed media processors are successful because media applications have abundant parallelism. This in turn, enables thousands of computations to be performed in parallel, requiring minimal global communication and storage enabling data to pass directly from one ALU to the next. A stream architecture exploits this locality and concurrency by dividing the communication and storage structures into three parts to be able to support many ALUs efficiently.

- Operands for arithmetic operations are stored in local register files (LRFs), near the ALU.
- Streams of data capture coarse-grained locality and stored in a stream register file (SRF), which efficiently transfer data to and from the LRFs between major computations.
- Global data is stored off-chip only when it is necessary.

These three explicit levels of storage form a data bandwidth hierarchy with the LRFs granting an order of magnitude more bandwidth than the SRF, and the SRF itself provide an

order of magnitude more bandwidth than off-chip storage.

As mentioned in [27], by exploiting the locality inherent in media-processing applications, the hierarchy above stores the data at the appropriate level, enabling hundreds of ALUs to operate close to their peak rate. Stream architecture can support, in a power efficient manner, large number of ALUs. The modern high-performance processors and digital signal processors rely on global storage and communication structures to deliver data to the ALUs. These structures consume both more power and take up more space per ALU than a stream processor.

Another interesting field related to stream processing is that to implement stream processing applications on FPGAs. A DSL, (domain-specific language) suited for implementation of stream processing applications on FPGAs is introduced by Jocelyn Sérot et al. in [36]. The language is called CAPH and relies upon the actor/dataflow model of computation. In the paper they describe the implementation of a preliminary version of the compiler of a simple real-time motion detection application on a FPGA-based smart camera platform.

As mentioned in earlier sections, stream processing applications, i.e. acting on the fly, requires high computing power. This is especially true when it comes to real-time image processing. The computing power is in the range of billions of operations per second. Which is often still beyond the capacity of GPPs, (general purpose processors). What is good with these applications is that most demanding tasks exhibit parallelism, and this makes them good candidates for FPGAs. A negative aspect of FPGAs is that programming FPGAs is mostly a hardware-orientated activity, relying on dedicated HDLs (Hardware Description Languages). These languages are designed for hardware designers, and are thereby unfamiliar to the programmers outside this field.

In [36] they believe that a DSL can provide pragmatic solution to the big gap between the programming model (as viewed by the programmer) and the execution model (as implemented on the target hardware) and thereby obtaining an acceptable trade-off between abstraction and efficiency requirements.

They explain issues raised by FPGA-programming and give a brief survey over some existing solutions and describe why these are not satisfactory. They also describe why the general dataflow/actor orientated model of computation can be a possible solution to these mentioned issues.

The paper show that efficient implementations of rather complex applications can be obtained with a language whose abstraction level is significantly higher than that of traditional HDL languages. Similar to this project, which studies complete frameworks to make the development phase easier, this paper works towards increasing efficiency by using higher level languages.

Similar with previous, Hendarmawan et al. in [24] mentions the high development costs of FPGAs. They try to simplify hardware accelerator development and how it can be implemented with low cost while still having a high performance, this can be compared with

our project on how to increase engineer efficiency with frameworks and thereby reduce development costs. An efficient way to accelerate regular expression matching on FPGAs for streaming processing is shown in the paper.

The approach taken was combined with a heterogeneous Computing Orientated Development Environment (hCODE) Framework for easy Hardware and Software Integration. This framework is from their previous work [40]. The experimental results show that the implemented hardware accelerator is faster than software implementation of regular expression for different sizes of data stream at low cost.

# Chapter 3

# Approach

Once we decided to focus on complete stream processing frameworks our original idea was first to perform a literature study over different frameworks, to see which ones seemed suitable for continued examination. After the literature study, we would follow up with an implementation, testing and result measuring phase. However, this strategy was discarded quite quickly since it proved difficult and time consuming for us to grasp their potential just by reading. We used a more iterative approach instead, where both reading about the frameworks' functionality and implementing simple test cases was performed to give a better understanding of their capabilities.

Every second week, we had a meeting with a reference group assigned by Saab, consisting of signal processing and computing experts. These meetings helped keeping the project on the correct course, aided us in the development with suitable requirements and use cases for the frameworks.

To be able to get relevant results, requirements on the frameworks were decided by us, with the help from our supervisors from Saab and the universities. These requirements, see Section 3.4, set the base for metrics, see Section 3.5, which would be used to evaluate and compare the final framework candidates. The requirements consider performance, engineering efficiency as well as a prediction of the frameworks' future. Also, to be able the get these results, different test use cases, see Section 3.6, were developed. E.g. use cases measuring performance were decided early in the project and other were developed during the project. We developed a benchmark, see Section 3.3, implemented in C++ using POSIX threads to show how the frameworks related to a more traditional solution. This benchmark was only used in the use case comparing performance. There are two major types of input data used in the use cases, see Section 3.2.

## 3.1 Testing environment

In the beginning of the project, development and testing was conducted on a laptop with two cores with hyper threading, resulting in a total of four virtual cores. Later, around half-time, all measurements and results was taken from a server with 16 cores. Custom BIOS settings and a table showing the setup from the lscpu command in Linux can be viewed in Appendix A.

## 3.2 Input data

To test the frameworks limits, and to simulate a pipeline similar to a signal processing system in a radar, we set up several use cases, see Section 3.6. The input data that was chosen in the use cases are of two types: two dimensional vectors, matrices, and three dimensional vectors, cubes, both containing integers.

Initially the plan was to start with an easier operation that was similar to an operation in a signal processing system, e.g. a matrix multiplication. Then during the project, develop the input data and operations as the pipeline grew. I.e. instead of using integers, switch to using complex numbers and add a Fast Fourier Transform (FFT) operation and iterate over data cubes.

As the project went on, original ideas had to be adjusted to meet the most important criteria. This resulted in minimizing the pipeline which was to represent a signal processing system, as well as continuing using integers and not move forward with implementing FFT.

To ensure fair comparisons, for each use case data was generated and saved to disk. The frameworks and benchmark then read data from these identical sources into memory before starting the measurements.

## 3.3 Benchmark

To be able to measure and compare the frameworks with regard to their throughput we implemented a benchmark. This benchmark is set up to see how big the frameworks' slowdown (see Section 3.5.1) is compared with a more traditional implementation.

The benchmark is implemented in C++ using POSIX threads, and uses worker threads in a thread pool to perform a operations over some input data stored in memory. The workers take the first item in the buffer, perform the operation and move on to the next available data. When all input data is computed, the program terminates. The code can be found in Appendix D. This program is translated over to the frameworks and is described more in the use case in Section 3.6.1.

## 3.4 Requirements

To be able to evaluate the frameworks, we set up requirements by discussing with our supervisors from Saab and the universities, on what the frameworks should be able to fulfill. These requirements was based on important factors from Saab, i.e. the development process and the processing of signal processing chains.

These are the requirements that a suitable framework for digital signal processing should be able to meet:

- Performance: The frameworks should not have a significant "slowdown" compared with the benchmark in terms of execution time.
- Scalability: The framework should support scaling. When adding additional cores or increasing the cluster, the system's performance should increase. It should also utilize the the available cores effectively.
- Data granularity: The framework should be effective at sending data of different sizes.
- Streaming: The frameworks can handle infinite data sets during a longer time.
- Load balancing: The frameworks distribute the computational load across available cores.
- Stability: Equal operations on data of equal size should have equivalent latency. I.e. the latency should not fluctuate, it should advocate towards an even pace.
- Sustainability: The framework should have continued development for the foreseeable future. This is measured by our perception which is based on the developers, community and experience within stream processing.
- Elasticity: The framework should dynamically distribute resources at run-time. E.g. a program consisting of different compute intensive steps, should be able to distribute the available cores so that an even flow of data is met through the program.
- Engineer efficient: It should be easy to implement and modify functionality, i.e. the framework should promote engineering efficiency.

## 3.5 Metrics

Metrics in performance and engineering efficiency were created to be able to measure and evaluate the requirements.

### 3.5.1 Performance metrics

- Average execution time: Time measured in seconds for how long it takes to execute a specific use case program. Used to evaluate the performance, scaling and data granularity requirements in Section 3.4.
- Slowdown: The performance cost of the automatic parallelization of the frameworks, as compared to the benchmark. Measured as a percentage of increase in execution time, i.e. $SD = \frac{R_F}{R_B}$. Where $SD$ being slowdown, $R_F$ being run-time for the framework, $R_B$ being the run-time for the benchmark. Used to evaluate the frameworks' performance as per the requirement in Section 3.4.
- Scaling speedup: $S = \frac{T_1}{T_N}$ where $N$ is the number of cores, $T_1$ is execution time for one core and $T_N$ being the execution time for $N$ cores. Used to evaluate the frameworks scalability as per the requirement in Section 3.4.
- Scaling efficiency: $E = \frac{T_1}{N \cdot T_N}$. Used to evaluate the frameworks scalability as per the requirement in Section 3.4.
- Throughput: Number of data cubes arriving at the sink per ten seconds. Used to evaluate how the frameworks handle streaming, i.e. infinite data sets, as per requirement in Section 3.4.
- Load balancing: Percentage of time spent in idle for all cores, measured by Linux sysstat. Used to measure how well the frameworks distribute the computational load over available cores, as per requirement in Section 3.4.
- Latency: The time it takes for an object to traverse the pipeline. Used to evaluate the frameworks stability as per requirement in Section 3.4.

### 3.5.2 Engineering efficiency metrics

Since engineering efficiency is a metric hard to measure, a plan was to create a questionnaire for developers at Saab. The questionnaire would cover functionality from the studied frameworks and thoughts by the developers on what they thought would improve their efficiency. Due to time limitations this questionnaire was never created.

Instead the engineering efficiency was measured by our own thoughts of the frameworks, and how we experienced them to solve the "x-abilities", doing our best to be impartial. The result is based on the following questions and used to evaluate the requirement set in Section 3.4:

- Learnability: How easy was it to learn what needed to be learned to implement the use cases?
- Usability: How easy and efficient was it to implement new use cases?
- Understandability: How easy is it to successfully explain a written program to a person with equal understanding of the framework?

- Lines of code

The result is presented in a table consisting of the questions above, as well as a row rating sustainability and the setup process. These questions are rated on a scale of 1 to 10, where 10 is the best. Lines of code will be compared between the frameworks and the benchmark, using the performance use case for comparison.

## 3.6 Use cases

This section describes the use cases of the project, and the structure of the data. These use cases are implemented with the frameworks and are used to evaluate them. For all use cases, unless explicitly stated, input data is read from the same file in both frameworks and the benchmark. An average run-time over 100 runs is used unless explicitly stated to get a more stable perception of the performance. These use cases test the behaviour of the frameworks, to see if they are suitable for further testing with more realistic signal processing chains for radar applications.

When compiling C++ code for the benchmark and RaftLib, the O3 flag was set to ensure good optimization.

### 3.6.1 Performance

To measure how the frameworks performances compare to each other, and how big their slowdown is, a performance use case was constructed.

The performance use case performed matrix multiplications in parallel and measured the time it took to perform 40,000 matrix multiplications in the frameworks and the benchmark. The matrices were of dimension 100x100. Before starting the measurement, a fixed set of pre-generated matrices were read from file and stored in memory. After this, a starting time was recorded and the matrices were distributed over the available worker threads in the pipeline. The second step was a matrix multiplication, which multiplied the matrices with themselves. When all matrices were multiplied a stop time was recorded and the execution time was calculated according to the metric in Section 3.5.1. The variance over all 100 runs was also calculated. An image on the implementation of the performance use case can be viewed in Figure 3.1.

The benchmark implemented this by pre-generating all the matrices and store them in a vector. The number of worker threads that were started depended on an input parameter. Once all worker threads were started, they iteratively pulled a matrix from the vector and performed the matrix multiplication operation until all matrices were computed. The index variable for the vector is controlled by a mutex-lock to control the read operation of the matrices in the vector. This to prevent a matrix being read and processed multiple times in the use case.

In RaftLib this was implemented with two kernels, linking them together by the $a <= b$

command, meaning that the processor kernel (b) creates as many clones of itself as the amount of output ports created by the producer kernel (a). For the processor to be able to create clones of itself, the producer kernel needs to implement the *raft::parallel_k*. The processor is a normal *raft::kernel* using the *CLONE()* macro. The producer takes the matrices from the pre-generated vector iteratively and sends them downstream to each output port. Each individual processor kernel receive the sent matrix on their input port and perform the matrix multiplication on it.

The Flink implementation of the use case consists of two classes, one main class for the actual use case, and one source-class handling the pre-generation of matrices. The source-class distributes and sends the matrices downstream to the created streams. For this use case one *DataStream* is created, the parallelism for the stream is decided by an input parameter. The created stream uses Flink *Map* function to perform the matrix multiplication.

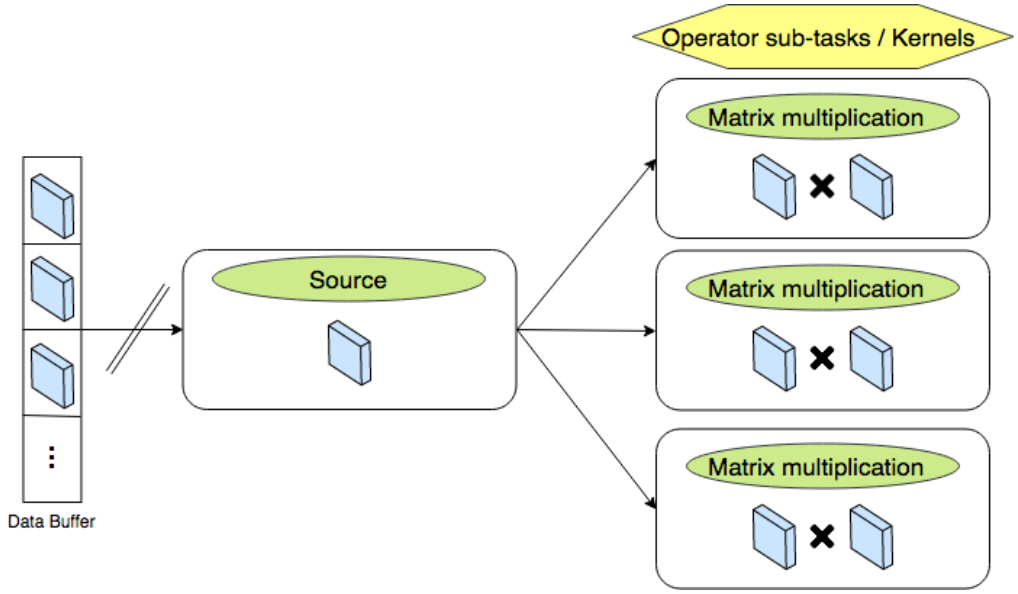



**Figure 3.1:** Illustration of the performance use case.

## Auto scaling

The scaling use case tested how well additional computational resources add to the performance of a simple parallel application. This was achieved by running the performance use case with 1 to 16 worker threads, a maximum of 16 cores was used to hardware limitations. The scaling use case was evaluated in both frameworks and the benchmark. Both the scaling speedup and scaling efficiency is calculated for each framework and the benchmark according to the metric in Section 3.5.1.

## Resource saturation

The resource saturation use case tested how the frameworks' performance is affected when the degree of parallelism reaches the number of available cores by the hardware. This use case have the same implementation as the performance use case, with a modification to the

cores available. E.g. in the first run a parallelism of 8 is used with only 8 cores available by the hardware, resulting in the set parallelism reaching the "roof". Then the second run all 16 cores are available by the hardware but the set parallelism is still 8. I.e. in the first case, half of the cores were turned off in BIOS, while in the second all cores were active. This use case uses the average execution time metric in Section 3.5.1 to measure the difference in execution time between the two runs.

### 3.6.2 Data granularity

A use case to study the frameworks' granularity while sending data was set up to find the frameworks' break point in size of data packets being sent. This was constructed by looking on how much extra overhead was added in throughput by sending sets of two-dimensional matrices of different dimension sizes, but with the same total amount data, the matrices were also filled with only ones to make them identical. The total number of integers sent was always $2^{20} = 1,048,576$, but they were sent at differing data granularities as shown in Table 3.1. I.e. the first iteration sends only sixteen matrices with a dimension of 256x256, next iteration sends sixty-four matrices but with a dimension of 128x128. An illustration of the use case can be viewed in Figure 3.2. An average was taken from 100 runs, and 16 parallel streams were used.

| #Matrices | Single dimension size | #Integers |
|---|---|---|
| $2^4 = 16$ | $2^8 = 256$ | $2^{4+2*8} = 2^{20}$ |
| $2^6 = 64$ | $2^7 = 128$ | $2^{6+2*7} = 2^{20}$ |
| $2^8 = 256$ | $2^6 = 64$ | $2^{8+2*6} = 2^{20}$ |
| $2^{10} = 1024$ | $2^5 = 32$ | $2^{10+2*5} = 2^{20}$ |
| $2^{12} = 4096$ | $2^4 = 16$ | $2^{12+2*4} = 2^{20}$ |
| $2^{14} = 16384$ | $2^3 = 8$ | $2^{14+2*3} = 2^{20}$ |

**Table 3.1:** The total number of integers is always $2^{20}$, but sent in different configurations as presented here and in Figure 3.2

The data granularity use case contains two main parts. First it pulls a matrix from the vector containing the pre-generated matrices, and send it downstream. The second step is to perform a matrix multiplication with itself, then move on to the next available matrix and repeat. When all matrices are computed, the run-time is printed. The data granularity use case uses the average execution time metric from Section 3.5.1 to measure the differences between all the runs.

### 3.6.3 Streaming throughput

This use case was created to verify that the frameworks can run on an infinite data set and support streaming. We continuously iterate over a pre-generated data set for ten minutes. It runs 16 parallel streams and sends three-dimensional cubes with a dimension of 30x30x30. The throughput was recorded every ten seconds as per the metric in Section

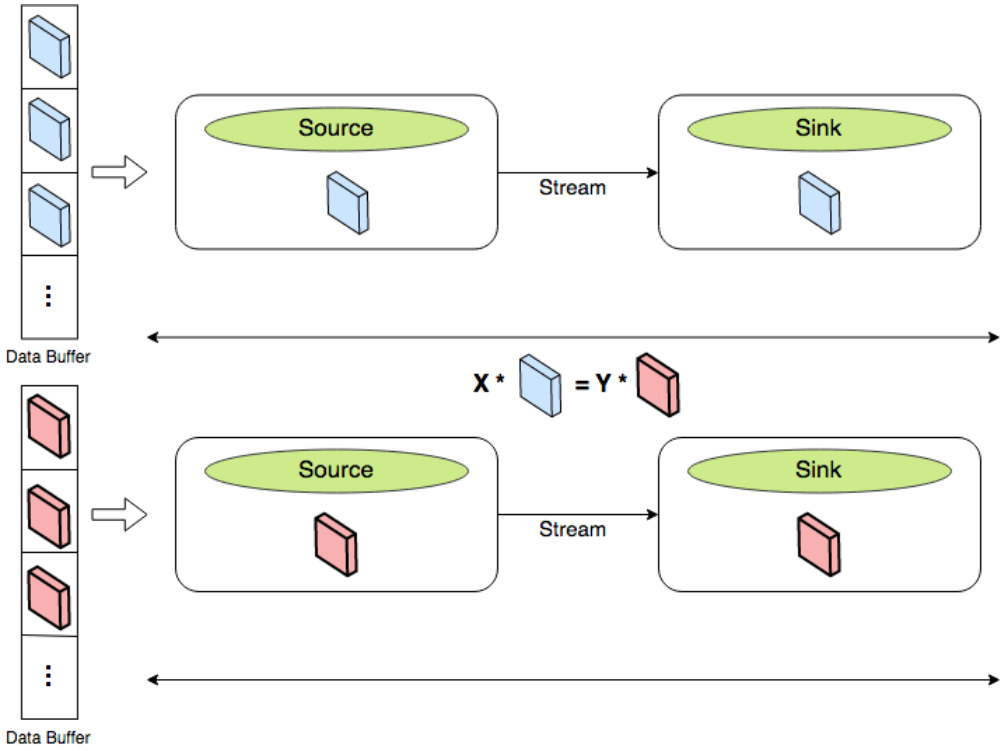



**Figure 3.2:** Illustration of the granularity use case.

3.5.1 and is presented in a graph to illustrate how it varies over time. For this use case only one sample run is presented. An illustration of the streaming use case set up can be seen in Figure 3.3.

The streaming use case is very similar to the performance use case setup. The difference is that the input is cubes and not matrices as well as containing a sink.

In RaftLib three kernel steps were created, a producer and a processor kernel implementing the same setup as in the performance use case in Section 3.6.1, and a consumer kernel which also is a *raft::parallel_k*. The consumer kernel has as many input ports as the number of processor kernels cloned. The kernels are linked together by the $a <= b >= c$ command. Where <= splits the stream, and >= merge it back together. The producer and processor works in similar way to the performance use case, but instead of simply drop the data in the processor, it is forwarded downstream to the consumer after the matrix multiplication. The consumer keeps track of the throughput, and every ten seconds stores the current throughput in a string. After 10 minutes, the string is stored to file and the program shuts down.

The Flink implementation consist of three DataStreams, a source stream distributing the pre-generated cubes, an operation stream performing the matrix multiplication and a sink tracking the throughput every ten seconds. Every ten seconds the current throughput is forwarded and printed on the console. When the program terminates all the throughput printouts are stored to file.

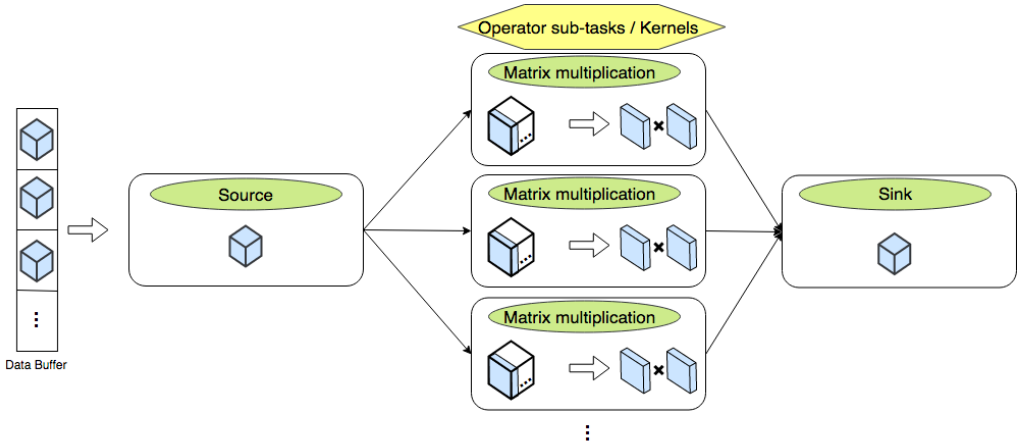



**Figure 3.3:** Illustration of the streaming throughput use case.

### 3.6.4 Load balancing

To test how the frameworks dealt with load balancing over cores and how many percent each core was idle during run-time, a stream topology of two operation blocks with cube distribution over the cores was set up. This stream pipeline was run simultaneously with Linux's sysstat using the "sar" command, to present how the system's cores was used.

The stream throughput use case without the sink was used to achieve the desired pipeline. The operation blocks used was a producer/source for the pre-generated cubes, which kept loop during the execution time, and a processing step doing the matrix-multiplication. The graph parallelism was downgraded from 16 to 8, to ensure there was unused capacity in the system. The test was run over five hours. Results was gathered from a single five-hour run.

### 3.6.5 Stability

A stability use case was implemented to see how the latency fluctuates when increasing the size of data sent over a single stream, i.e. without any parallelism. This expected fluctuation is illustrated in a simple figure and can be viewed in Figure 3.4. The stability is measured with the latency metric from Section 3.5.1.

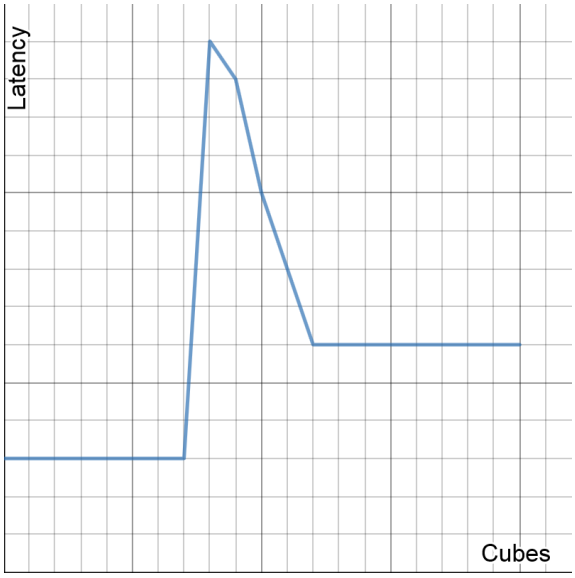



**Figure 3.4:** Expected latency fluctuation during the transitioning of the cube size in the stability use case.

The use case was tested by first sending a set of cubes with a dimension size of 32x32x32, then send another set of larger cubes with a dimension size of 64x64x64. Each set containing 1000 cubes, i.e. a total of 2000 cubes per use case was sent. Two operations were performed on each cube: recording a starting timestamp, and recording a receiving timestamp at the other end of the stream, by subtracting the latter timestamp with the first the latency was calculated. The use case was also run an additional time with the same set up except for the dimension, the latter time a single dimension size of 8, respectively 16 was used. An illustration of the use case can be viewed in Figure 3.5.

In RaftLib two vectors was created, keeping track of the start and end times for each cube. Two normal *raft::kernel* was used, a producer and a receiver. The producer iteratively pulls a cube from the pre-generated vector, and sets a start time to the corresponding index in the start-time vector, and immidietly send the cube downstream. The cube is then received by the receiver and a stop time is set on the corresponding index in the stop-time vector. When all cubes are processed the latency for each cube is calculated and stored to file by taking the end times subtracted by the start times for each index.

A similar process was performed in Flink. Flink used two DataStreams, one to set the start time and one to set the stop time, when the stop time is set the latency is calculated by a *FlatMap* function. The latencies for each cubes are then stored to file.

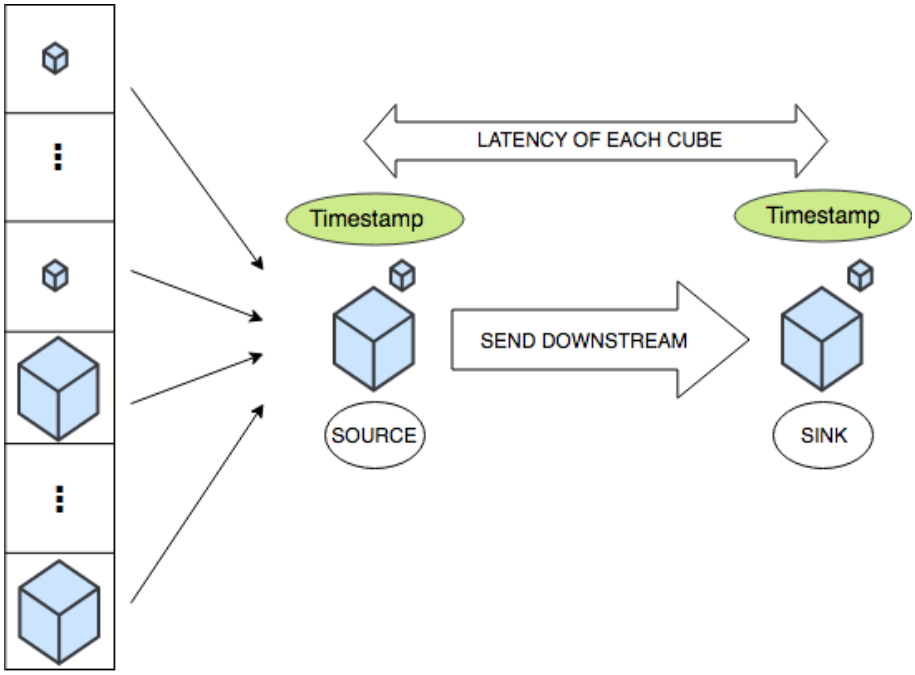



**Figure 3.5:** An illustration of the stability use case, 1000 smaller and 1000 larger cubes are sent, and latencies are measured for each cube.

# Chapter 4

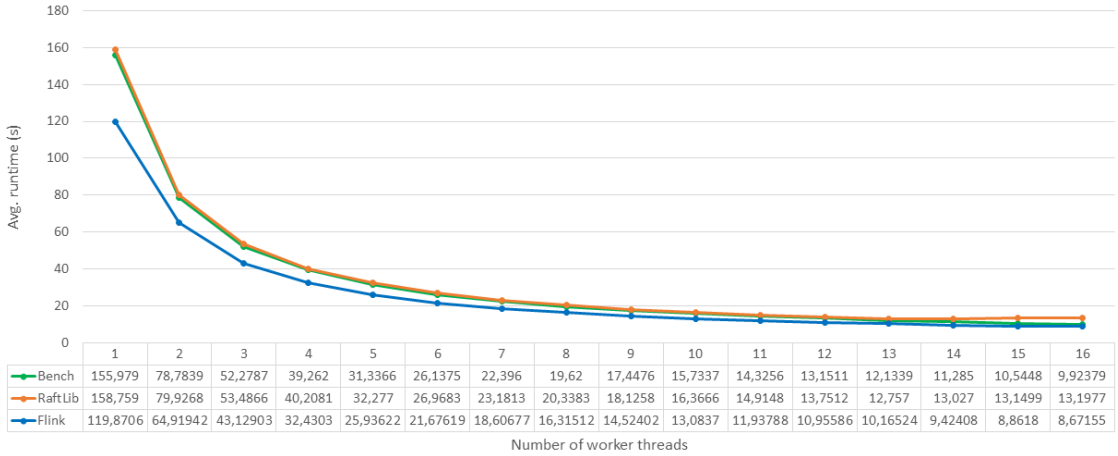
# Results

All the results have been collected by running the use cases on the computer described in Section 3.1. The input data used in the different use cases was identical for all frameworks and the benchmark within the same use case. Between use cases the input data could vary, and be more specified for a particular use case.

## 4.1 Performance

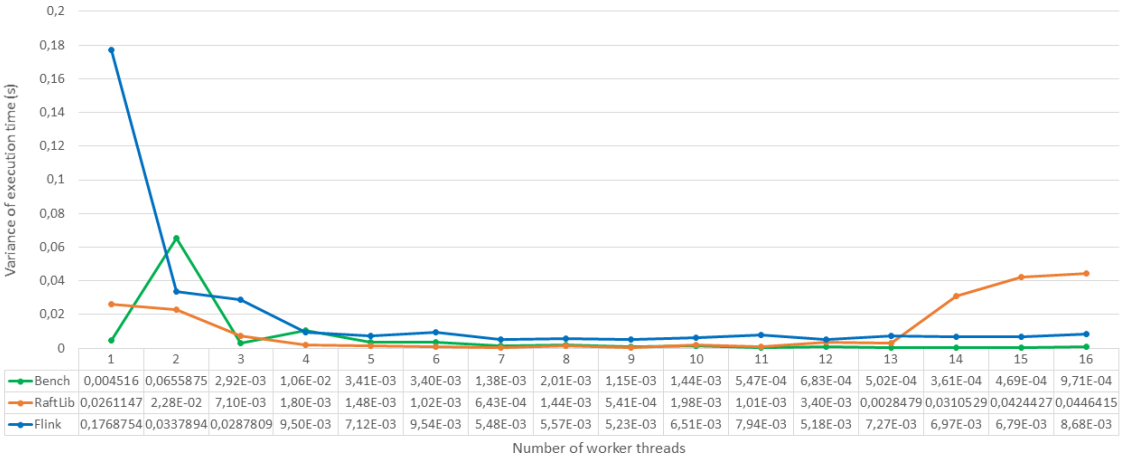
A graph showing the results for the average run-time at different number of worker threads, when computing over 40000 matrices can be seen in Figure 4.1. Figure 4.2 displays the variance of the same runs as seen in Figure 4.1 for the benchmark and each framework. At first glance the results look promising for the frameworks, both frameworks keeps up relatively good with the benchmark in terms of execution time and their corresponding variances. However, Flink presents a very interesting result, it being faster then the benchmark. Another interesting factor is that RaftLib's execution time and variance increase when using more than 13 worker threads.

As seen in Figure 4.3, RaftLib's slowdown towards the benchmark has its lowest slowdown at 1.41% using two worker threads. This number rises slightly with each added parallel compute kernel until kernel 13-14, where the slowdown jumps from 4.88% to 13.37%, probably due to resource saturation. Flinks best "slowdown" towards the benchmark was -30.12% with no parallelism, and -21.36% with a parallelism of two, and after that increasing similar to RaftLib in a linear fashion. The term "slowdown" is based on our expectation that neither of the frameworks would beat the benchmark in performance, this turned to be wrong in Flink's case.




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bench | 155,979 | 78,7839 | 52,2787 | 39,262 | 31,3366 | 26,1375 | 22,396 | 19,62 | 17,4476 | 15,7337 | 14,3256 | 13,1511 | 12,1339 | 11,285 | 10,5448 | 9,92379 |
| RaftLib | 158,759 | 79,9268 | 53,4866 | 40,2081 | 32,277 | 26,9683 | 23,1813 | 20,3383 | 18,1258 | 16,3666 | 14,9148 | 13,7512 | 12,757 | 13,027 | 13,1499 | 13,1977 |
| Flink | 119,8706 | 64,91942 | 43,12903 | 32,4303 | 25,93622 | 21,67619 | 18,60677 | 16,31512 | 14,52402 | 13,0837 | 11,93788 | 10,95586 | 10,16524 | 9,42408 | 8,8618 | 8,67155 |

**Figure 4.1:** Execution times for performing 40000 matrix multiplications, at different levels of parallelism.

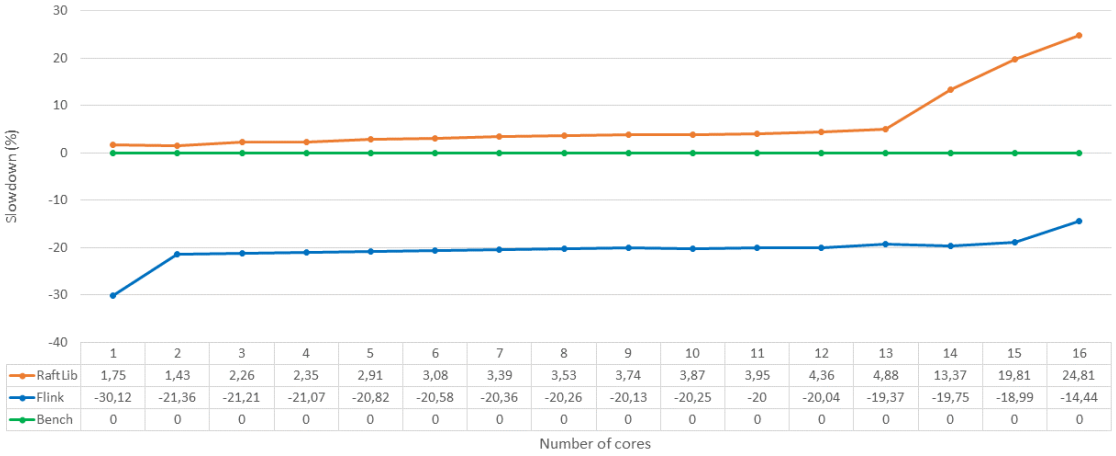



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bench | 0,004516 | 0,0655875 | 2,92E-03 | 1,06E-02 | 3,41E-03 | 3,40E-03 | 1,38E-03 | 2,01E-03 | 1,15E-03 | 1,44E-03 | 5,47E-04 | 6,83E-04 | 5,02E-04 | 3,61E-04 | 4,69E-04 | 9,71E-04 |
| RaftLib | 0,0261147 | 2,28E-02 | 7,10E-03 | 1,80E-03 | 1,48E-03 | 1,02E-03 | 6,43E-04 | 1,44E-03 | 5,41E-04 | 1,98E-03 | 1,01E-03 | 3,40E-03 | 0,0028479 | 0,0310529 | 0,0424427 | 0,0446415 |
| Flink | 0,1768754 | 0,0337894 | 0,0287809 | 9,50E-03 | 7,12E-03 | 9,54E-03 | 5,48E-03 | 5,57E-03 | 5,23E-03 | 6,51E-03 | 7,94E-03 | 5,18E-03 | 7,27E-03 | 6,97E-03 | 6,79E-03 | 8,68E-03 |

**Figure 4.2:** Variance from 100 runs of the results presented in Figure 4.1.

## Scaling

The scaling- speedup and efficiency is used to show how well the frameworks and benchmark perform compared with themselves when the parallelism increases. The scaling speedup metric was 12.03 for RaftLib at max parallelism of 16, and reached a maximum of 12.44 with thirteen parallel compute kernels. Flink had a maximum speedup of 13.82 while utilizing all 16 cores. The benchmark had a speedup of 15.71 at 16 cores. The scaling efficiency at parallelism 16 was 0.98 for the benchmark, 0.75 for RaftLib and 0.86 for Flink. This shows that even though the frameworks does not have the same speedup and efficiency in terms of scaling as the benchmark. They still have good results when looking on how the developer do not need to think too much about the parallelization process, since this is done by the framework.




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RaftLib | 1,75 | 1,43 | 2,26 | 2,35 | 2,91 | 3,08 | 3,39 | 3,53 | 3,74 | 3,87 | 3,95 | 4,36 | 4,88 | 13,37 | 19,81 | 24,81 |
| Flink | -30,12 | -21,36 | -21,21 | -21,07 | -20,82 | -20,58 | -20,36 | -20,26 | -20,13 | -20,25 | -20 | -20,04 | -19,37 | -19,75 | -18,99 | -14,44 |
| Bench | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figure 4.3:** Framework slowdowns as normalized by the benchmark of the results presented in Section 4.1.

## Resource saturation

The benchmark performs well, as expected, as the available resources are saturated, probably since the only extra thread who is not a worker thread is the main thread, which sleeps until all worker threads are done with their tasks. While RaftLib performs significantly worse, probably due to, apart from having worker threads, use an extra producer thread and supervisory threads. Flink also drops slightly in performance, but very little compared to RaftLib. Similar to RaftLib, Flink also uses supervisory threads, but according to the result, more effectively. The slowdown between the 8/16 and 8/8 setups are 1% for the benchmark, 6.9% for Flink and 31.3% for RaftLib. Figure 4.4 details these results.

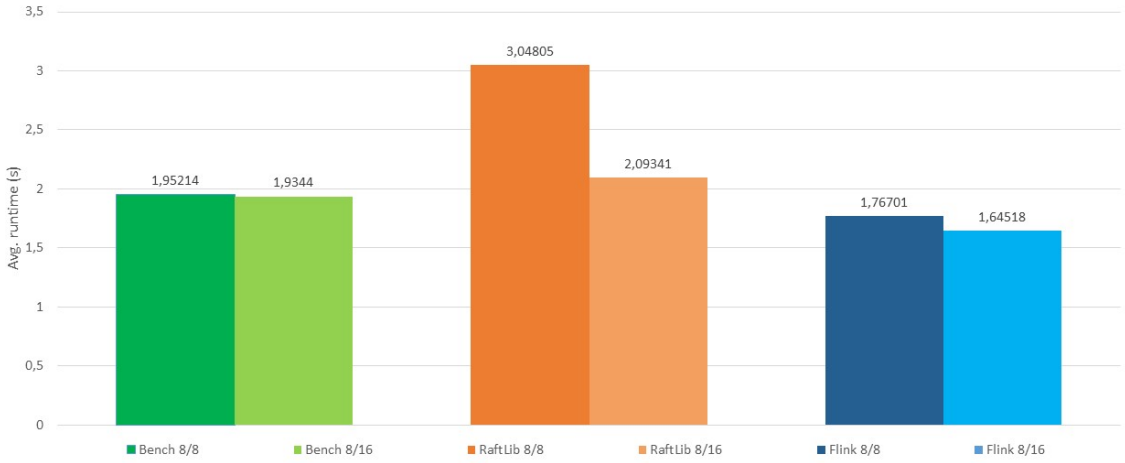
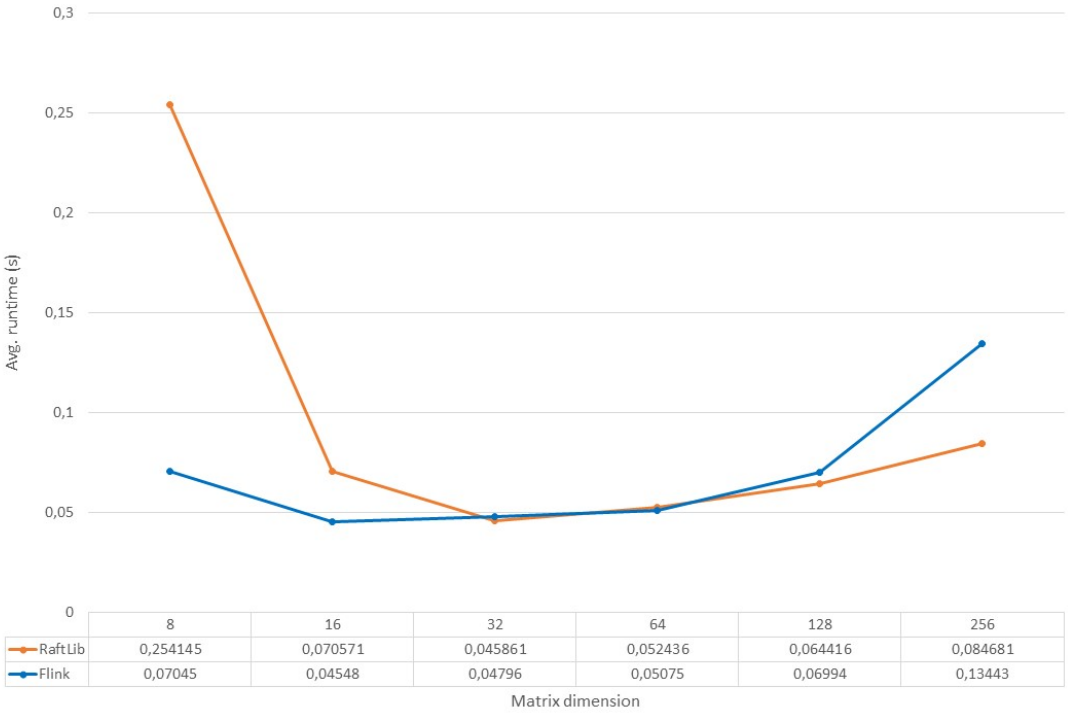



**Figure 4.4:** Histogram showing the run-time using 8/8 available cores versus running with 8/16 available cores. I.e. performance as the available computational resources are saturated.

## 4.2 Data granularity

The granularity use case was run with different setups as seen in Table 3.1. The result for each case's run-time was the average of 100 runs and can be seen in Figure 4.5.




| | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| RaftLib | 0,254145 | 0,070571 | 0,045861 | 0,052436 | 0,064416 | 0,084681 |
| Flink | 0,07045 | 0,04548 | 0,04796 | 0,05075 | 0,06994 | 0,13443 |

**Figure 4.5:** Average run-time of 100 runs when changing the matrix dimension and the number of matrices sent. Result from the data granularity use case.

As presented by Figure 4.5 above, RaftLib is way worse than Flink when handling many small data objects, but more effective when handling larger data objects. Flink is more consistent and performs well for most tested data sizes, but looses performance when the size of the data becomes to big.

## 4.3 Streaming throughput

Figure 4.6 shows the results for the streaming throughput use case on 16 worker threads on 16 cores. Figure 4.7 shows the same use case but only 8 worker threads running on 16 cores. The y axis shows the throughput in cubes over intervals of ten seconds. The x axis shows time in minutes. The reason why two result graphs are presented is that the first run, Figure 4.6, was measured before the results from the resource saturation use case was done. So therefore another run, Figure 4.7, was done considering the resource saturation results. Surprisingly, RaftLib beat Flink in throughput in the second run. This was not expected since Flink always performed better in earlier use cases. The reason why is explained by the data granularity use case, and will be more addressed more in the discussion section below.

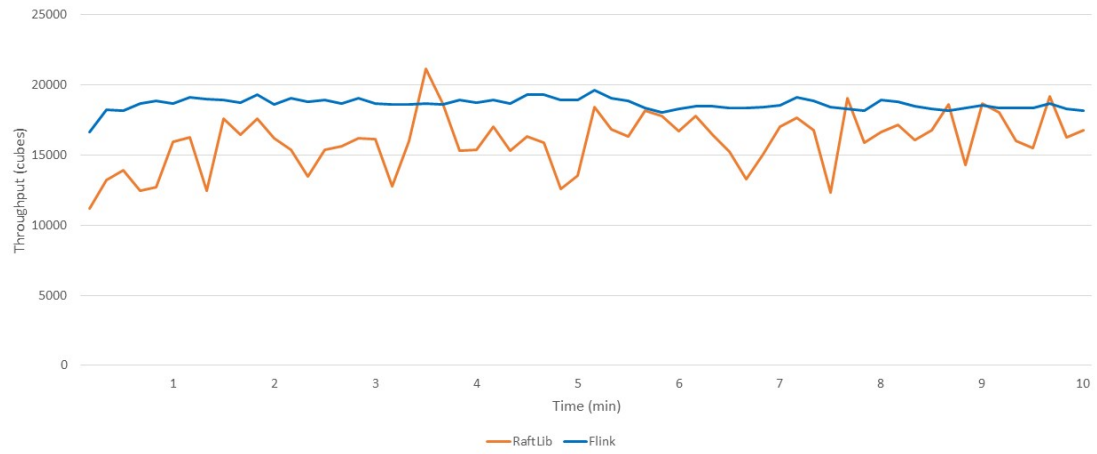



**Figure 4.6:** Result from the streaming throughput use case. Throughput every ten seconds for a duration of ten minutes. 16 worker threads on 16 cores.

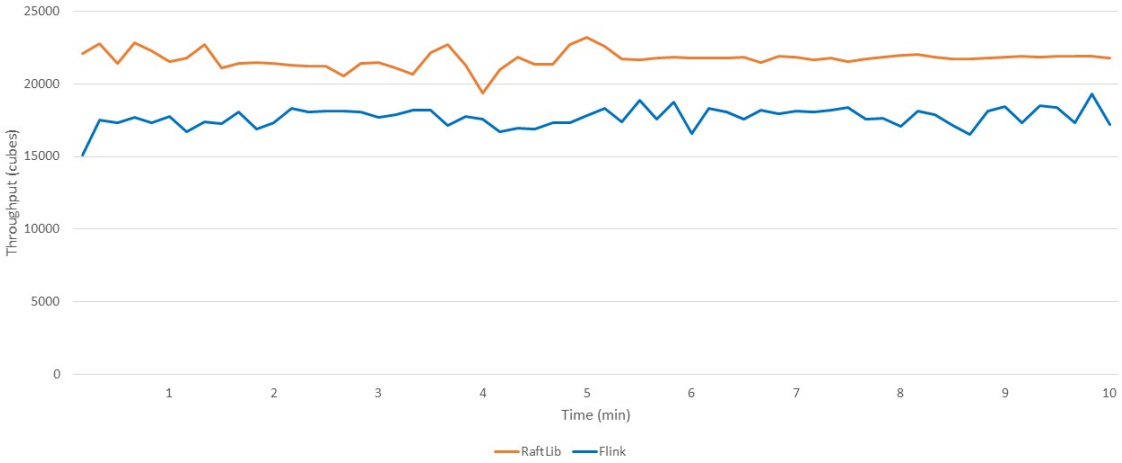



**Figure 4.7:** Result from the streaming throughput use case. Throughput every ten seconds for a duration of ten minutes. 8 worker threads on 16 cores.

## 4.4 Load Balancing

The load balancing use case has a parallelism of 8, with 16 active cores in the test computer to measure the load balancing over all available cores. The result for the frameworks can be seen in Figure 4.8. The result was collected using the sysstat application in Linux. RaftLib does not seem to have any form of load distribution over the cores, since both core 0 and 15 in Figure 4.8 are active 90% of the time while core 7 has an idleness of 97%. Flink shows more promise, but after a five hour run, one could believe a better load distribution would be accomplished. The largest difference between core idleness for Flink is 41.6%.

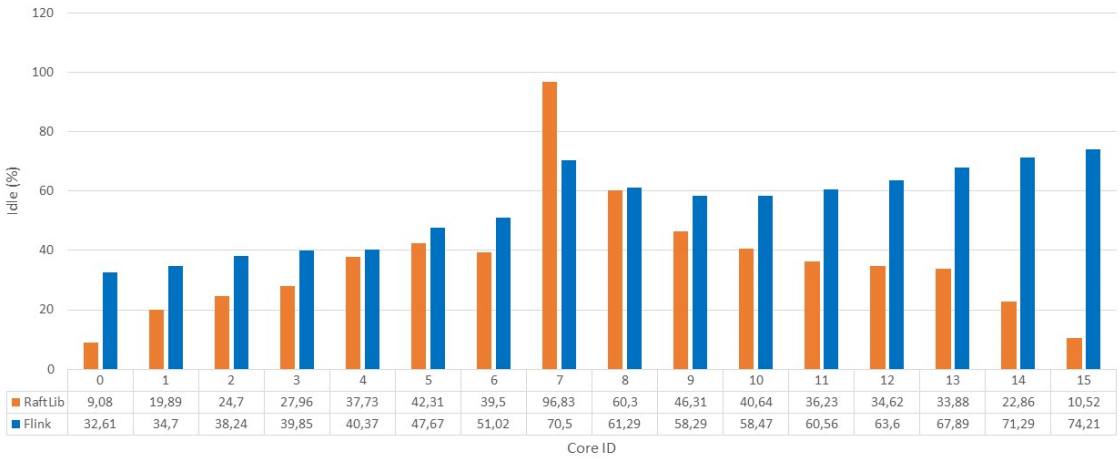



**Figure 4.8:** Image displaying the load balance for each core in Flink and RaftLib running the stream implementation, see Figure 3.3, for a duration of 5 hours.

## 4.5 Stability

Each cube's latency in order, for the different frameworks is recorded and presented in graphs below. In Figure 4.9 both RaftLib and Flink is presented, as can be seen RaftLib have much higher latency than Flink. It is actually 60-70 times larger than Flink's latency, this is quite surprising, since RaftLib had better throughput than Flink when running with larger data objects in the streaming use case.

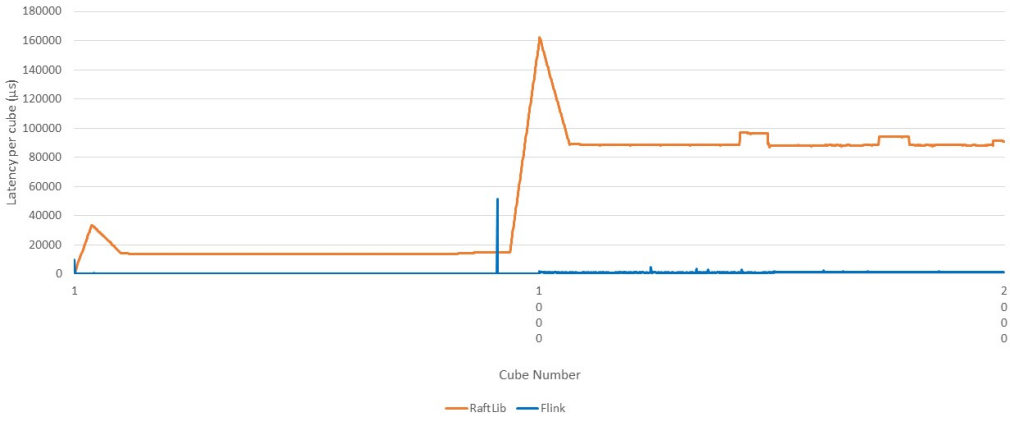



**Figure 4.9:** Latencies of each cube for RaftLib and Flink in the stability use case.

Figure 4.10 displays all values of Flink by itself. An edited version of this graph can be viewed in Figure 4.11, the highest latency outlier has been edited out to give the graph a more readable scale. For more part, Flink have a stable and quite low latency, but time to time an occasional cube have an immense latency compared with the rest. This happens more often with the larger cubes, again confirming that Flink performs better with smaller data objects.

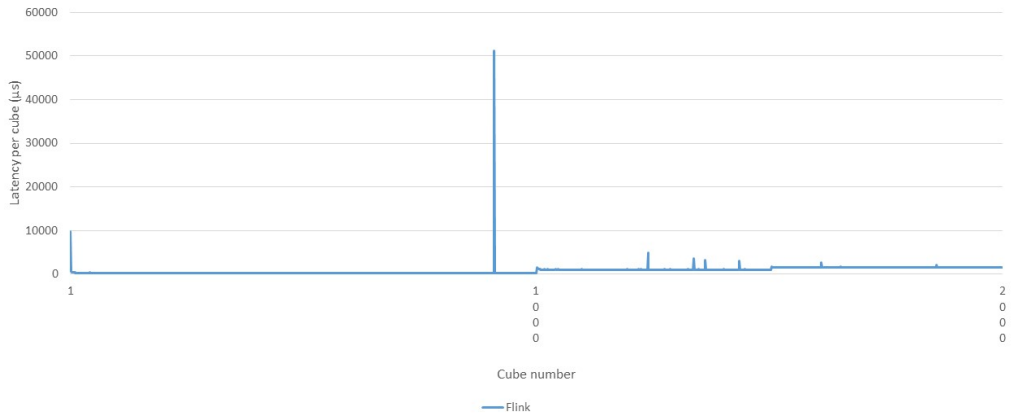



**Figure 4.10:** Latencies of each cube for Flink in the stability use case.

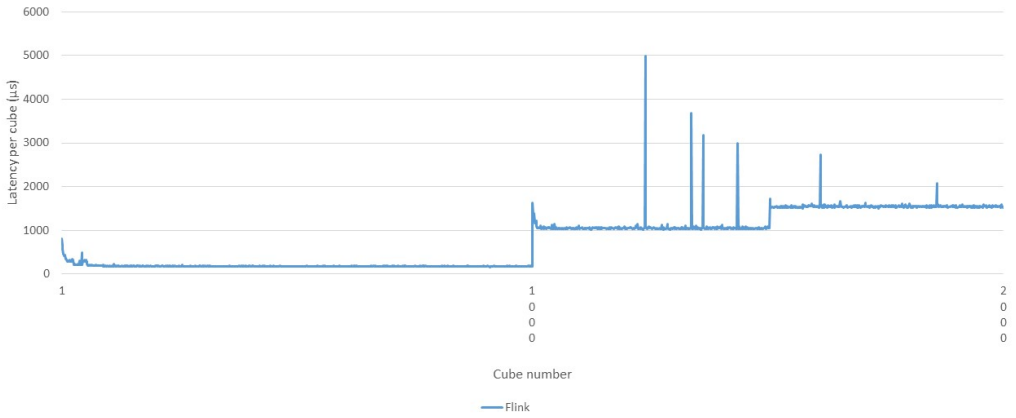



**Figure 4.11:** Latencies of each cube for Flink in the stability use case with the value at position 910 edited to make the graph more clear.

In Figure 4.12 a graph shows the interval [940, 1060] to present a more clear view of the transition from small to larger cubes for Flink. From this its seen that Flink have a quite fast transitioning time of ten cubes before its back to an even and stable pace.

A graph presenting RaftLib's values by themselves can be seen in Figure 4.13, and in Figure 4.14, a more zoomed in interval [930, 1070] is displayed. Even though RaftLib has very high latency, it has an even pace, however, its much slower than Flink on transitioning back to a more even pace after the increased size of the data cubes. The transitioning even starts before it has even occurred, this should not be able to happen when processing data in real-time. However, since the data is stored in pre-generated vectors for our use cases, RaftLib might have been able to "cheat" and try to optimize.

Since this use case was run with larger cubes, 32x32x32 cubes transitioning into 64x64x64 sized cubes, another run was done with smaller sized cubes as well. This due to both frameworks performed better with data objects not having to big size. The results for the latter

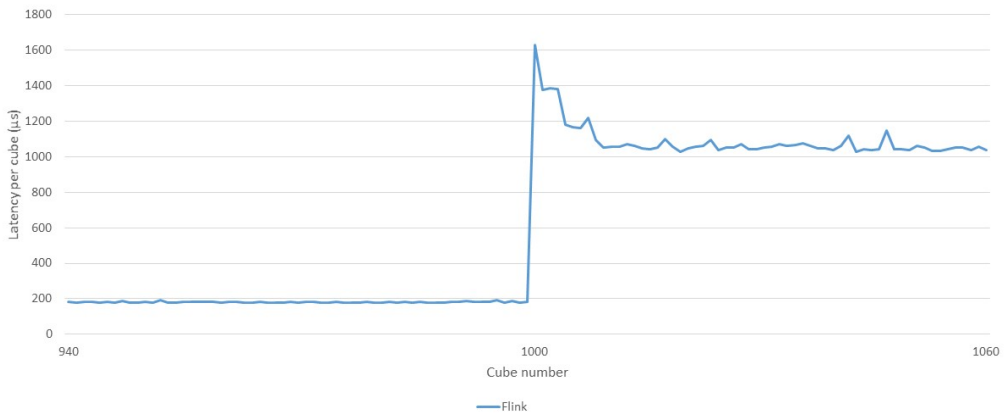



**Figure 4.12:** Latencies of cubes for Flink in an interval [940, 1060] in the stability use case.

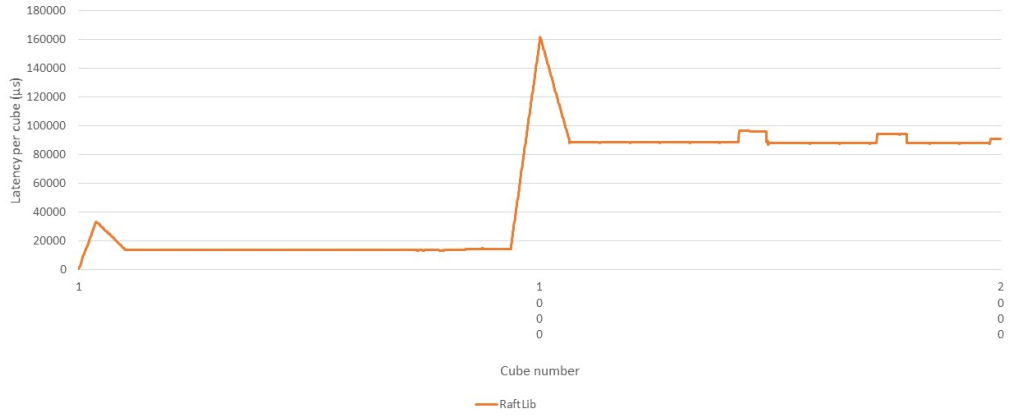



**Figure 4.13:** Latencies of each cube for RaftLib in the stability use case.

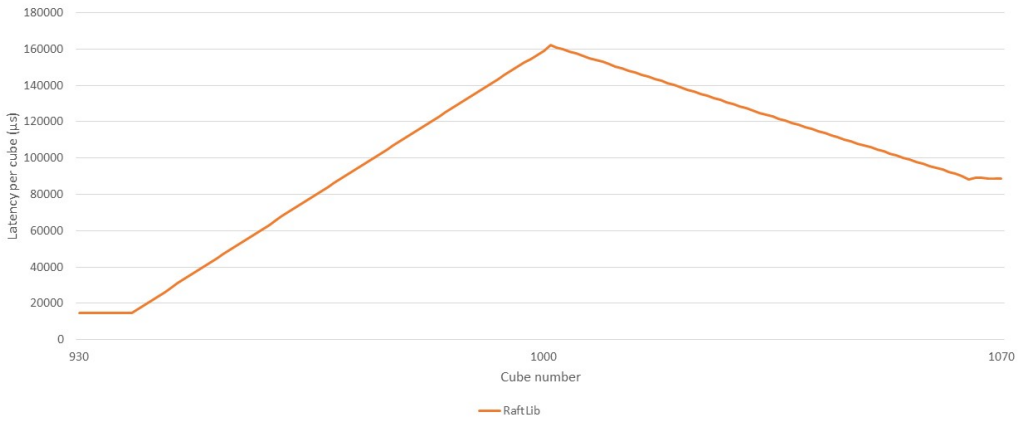



**Figure 4.14:** Latencies of cubes for RaftLib in an interval [930, 1070] in the stability use case.

stability use case run, with smaller sized cubes, can be seen in Figure 4.15 for Flink and in Figure 4.16 for RaftLib. Flink showing similar results as before, but with a bit more abnormal cube latencies. RaftLib still having around 50-60 times higher latencies than Flink, and being more instable than before.

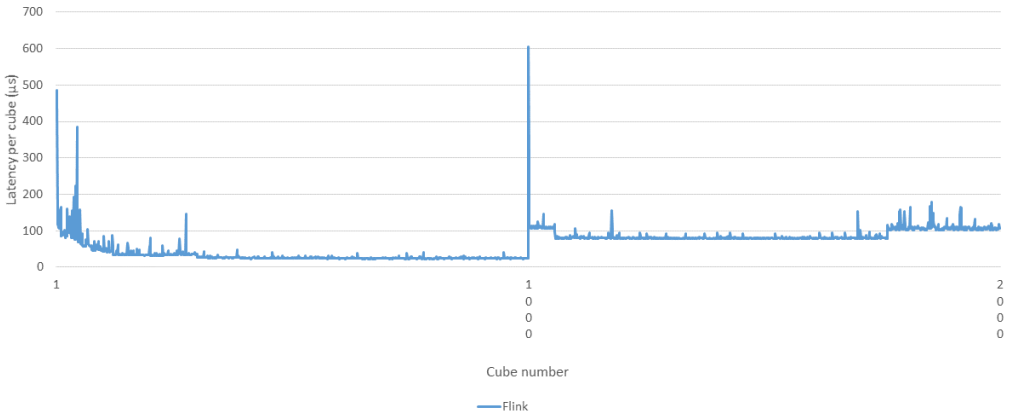



**Figure 4.15:** Latencies of each cube for Flink in the stability use case with small sized cubes, dimension 8x8x8 & 16x16x16.

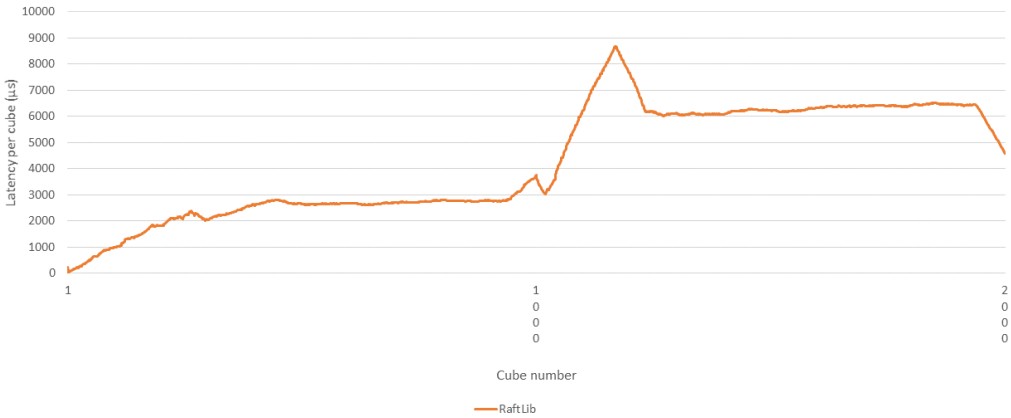



**Figure 4.16:** Latencies of each cube for RaftLib in the stability use case with small sized cubes, dimension 8x8x8 & 16x16x16.

## 4.6 Engineering efficiency

The engineering efficiency results were produced by evaluating the metrics described in Section 3.5.2. These metrics were evaluated using our own experience of working with the frameworks, and graded using a scale of 1 to 10. The results are placed in Table 4.1, also containing the lines of code for the performance use case for all frameworks and the benchmark.

| | RaftLib | Flink | Benchmark |
|---|---|---|---|
| Learnability | 5 | 6 | - |
| Usability | 5 | 8 | - |
| Setup | 8 | 5 | - |
| Understandability | 6 | 7 | - |
| Sustainability | 3 | 9 | - |
| Lines of code | 108 | 25 | 55 |

**Table 4.1:** Table rating each framework in engineering efficiency, taking "x-abilities" into account. The ranking is 1-10, the higher the score the better. Also showing the lines of code of the performance use case.

The scores to the different categories were first set separately. When both of us had written our assessment of the frameworks individually, we compared each others scores and discussed how we would rate the frameworks based on the individual results for each category. Both of us agreed that Flink was the most impressive of the two when it came to engineering efficiency. Once you had passed Flink's learning curve, it was smooth and easy to work with. RaftLib was easier in the beginning, it was very simple to learn the essentials and to start implement basic functionality. It was however more complicated and harder to do more complex things due to some functionality not working for the framework, at least to our experience. E.g. when creating longer chains in the graph, we followed the syntax described on RaftLib's wiki, without getting it to work.

When looking at the given scores in Table 4.1, RaftLib and Flink receives an average score of 5.4 and 7, respectively. Flink also having considerably less lines of code than Flink, as well as the benchmark.

# Chapter 5
# Discussion

## 5.1 Performance, scaling & resource saturation

When looking at how the frameworks scale compared with the benchmark in Figure 4.1, RaftLib follows tightly with just a bit more delay, laying around 0.6 seconds over the benchmark. Until RaftLib reaches a parallelism of thirteen worker kernels, it keeps decreasing the run-time. At fourteen it starts to increase, and drops in performance. This could be due to RaftLib creating supervisory threads, and when approaching the amount of available cores, it becomes overloaded. In this use case, RaftLib also uses a producer kernel, but this could be compared with the main method in the benchmark and the source method in Flink. This producer kernel should not add extra burden to the implementation for RaftLib, since no sign for reduced performance can be seen in Flink when approaching the roof of available cores.

When looking at the scaling speedup, RaftLib reaches its maximum speedup at thirteen worker kernels, indicating that RaftLib does not perform as well when reaching the available core limit. To see how all the benchmark and the frameworks performed when approaching the core limit, the resource saturation use case was set up. In Figure 4.4 its shown that the benchmark is not affected much at all, neither is Flink. Flink having a slightly bigger performance drop than the benchmark, probably due to it also running supervisory threads in the background. However, the performance drop for Flink is still not as big as for RaftLib, which is almost a whole second worse when running with 8/8 cores.

The expectation in the beginning was that the frameworks would have worse performance than the thread pool style benchmark, and to measure this a slowdown metric was setup. I.e. how many percent slower were the frameworks compared with the benchmark, as can be seen in Figure 4.3. As mentioned above, RaftLib performs as expected and achieves

good results and stays close to the benchmark with a slight linear increase until it reaches the available core limit. What was most surprising was that Flink had a negative "slowdown", being faster than the benchmark at all times. Flink has a negative slowdown of -14.44% when running with sixteen cores. The best "slowdown" of -30.12% is when running with one worker thread, i.e. no parallelism. It seems very suspicious and strange that a framework is faster than a traditional solution using one thread iterating through a vector. Since no resource sharing problems can occur when running with only one thread in the benchmark, its really odd that Flink is faster. Even though Flink seems to be great at moving data quickly, which is essential for real time stream processing, the benchmark threads simply reads from a location in memory and should be more effective. We have come up with two explanations which might explain why Flink is so fast. Either Flink "cheats" by somehow preprocessing the data, or the matrix multiplication implementation is performed faster in Java than in C++ optimized with the O3 flag. Both seem unlikely, for the former, why have custom sources if they are not treated as "real" streams? For the latter case, the matrix multiplication algorithm in both Java and C++ is a classic $O(n^3)$ implementation, and is implemented as likely as possible in both languages. If any implementation would be faster, our expectation was that the C++ benchmark and RaftLib would have an advantage, but it might be that Java was better at optimizing an inefficient algorithm like ours. External libraries would be used in real-world implementations and would give a more reliable result than given here. Another possibility can be that Flink parallelizes behind the scenes, but to our knowledge it does not. When we ran our use cases we used *htop*, an interactive process viewer for Unix, showing which of the cores were being used at the moment.

When looking at the scaling- speedup and efficiency metric, more expected results is presented. They describe how well the frameworks perform the parallelized tasks compared to themselves. The benchmark has both the best speedup and the highest efficiency at a parallelism of sixteen cores, with a speedup of 15.71, and a efficiency of 0.98. While utilizing all sixteen cores RaftLib's speedup was 12.03 and Flink's was 13.82. The efficiency was 0.75 for RaftLib and 0.86 for Flink. But since RaftLib got saturated at 13 cores, its best speedup and efficiency was 12.44 and 0.96, respectively, indicating that RaftLib has a better self-scaling than Flink, even though Flink seems to perform better execution wise.

## 5.2 Streaming throughput

In the first run measuring streaming throughput, 16/16 worker threads were used. The result was quite expected after looking at the result for the performance use case in Figure 4.1. Flink was above RaftLib in thoughput per 10 seconds over a duration of 10 minutes. Flink also had a quite stable throughput with not that big of a difference. Meanwhile RaftLib was not stable at all, jumped between a throughput of 11000 and 21000. This is most likely caused by using 16 worker threads along with an additional producer and consumer thread, plus the overhead threads overlooking the system. Resulting in saturation of resources, and take its toll on the performance.

To make the use case more fair for RaftLib, we ran it again, this time with 8/16 available

cores. To our surprise RaftLib now outperformed Flink in throughput, while also having a throughput higher than itself when running on 16/16 cores. We then remembered that the input used in the performance use case was 100x100 sized matrices, but in the streaming case we used 30x30x30 sized cubes. This cube size corresponds to a size of approximately 164x164 sized matrices. I.e. $30 * 30 * 30 = 27000$ and $\sqrt{27000} \approx 164$. This hinted that the frameworks' performance was somewhat dependent on what size the data had when sent over the streams. Section 5.4 covers the results for sending different sized matrices in the frameworks.

## 5.3 Load balancing

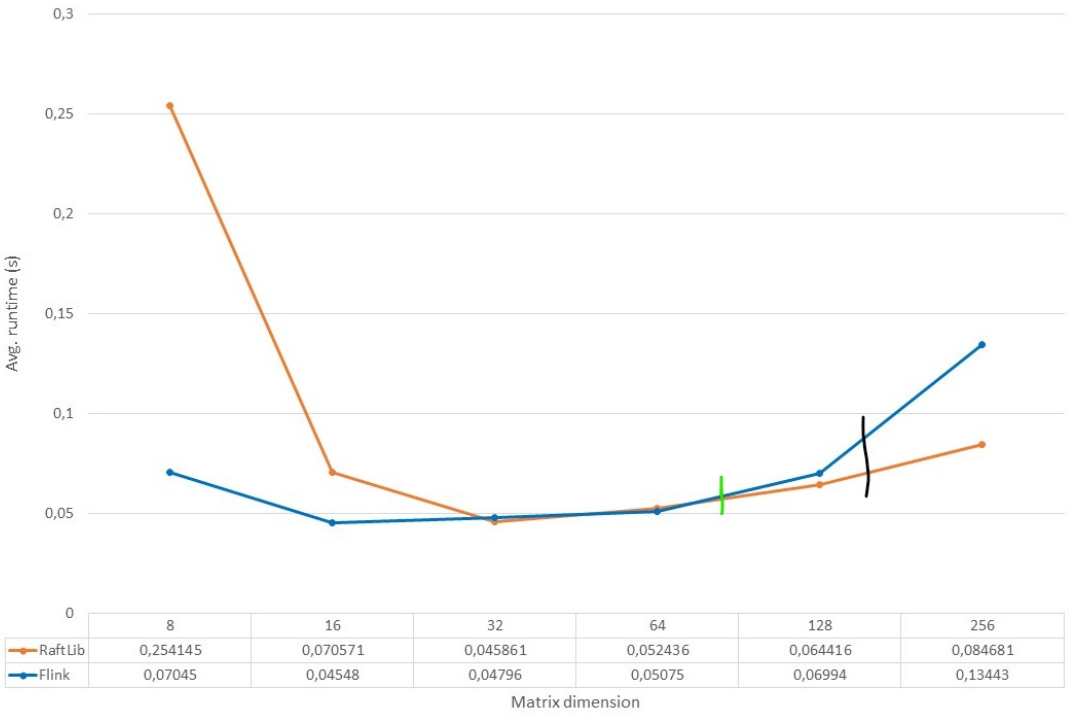
The load is not really evenly distributed in either of the frameworks, as seen in Figure 4.8. Flink has the most even distribution and might use some load balancing strategy. One could however think that after five hours it should not be to hard to accomplish a better and more even distribution. RaftLib does not show any big sign for a load balancing strategy at all. The biggest difference between idleness of cores for RaftLib is 87%, when looking at core zero and seven. This indicates that a *least used* strategy is not used to schedule processes in RaftLib.

## 5.4 Data granularity

The results from the streaming throughput use case in Section 4.3 showed that the performance was affected by the amount of data contained in an object. Therefore a use case was developed to see at what point the frameworks were most effective. An assumption that sending fewer references to objects containing more data would increase performance, since moving data is costly. This was the case up to a certain point. As can be seen in Figure 4.5, after this point both frameworks' performance decreased. A likely explanation why the performance decreases after a certain size on the objects is that the stream buffer sizes have an upper bound in size. For RaftLib this is 8 MB as mentioned in Section 1.4.1. When the objects are sufficiently large, these buffers are filled very quickly, which negatively affects performance.

Flink was faster when working with smaller object, having best performance when the matrices had a dimension of sixteen, while RaftLib had a better curve when working with larger objects, being at its best having a dimension of thirty-two for the matrices. Overall RaftLib and Flink was quite equal when sending the different object sizes. However, RaftLib is really slow when sending really small objects, while Flink prefers it.

The results from the data granularity use case explains why RaftLib performed better at the streaming throughput use case. As can be seen in Figure 5.1, Flink is much slower than RaftLib when working on the object sizes used in the streaming throughput use case. This likely affected the results gathered when sending larger cubes in the stream throughput and stability use cases.




| | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| RaftLib | 0,254145 | 0,070571 | 0,045861 | 0,052436 | 0,064416 | 0,084681 |
| Flink | 0,07045 | 0,04548 | 0,04796 | 0,05075 | 0,06994 | 0,13443 |

**Figure 5.1:** Image showing the difference in matrix-dimension of the input in the performance and streaming throughput use cases. The green and black lines marking the approximate matrix dimensions for the performance and streaming throughput use cases, respectively.

## 5.5 Stability

As can be seen in Figure 4.9 we see that RaftLib clearly has a much bigger latency than Flink. It has 60-70 times higher latency than Flink, both with the smaller input cubes and the larger. A possible reason for the higher latency in RaftLib could be that the use case was run with 16/16 worker threads, which has shown to be costly for RaftLib in the performance use case in Section 5.1. Even though this is a factor, it could not be the only reason, since the gap is so huge.

In Figure 4.13 and in Figure 4.14 it is shown more clearly that RaftLib successively increases each cubes latency 60 cubes ahead of the transition and then after, successively decreases it back before finding a new stable level of latency. In total its approximately 120 cubes before its stable again. Since RaftLib increase the latency before the transition, we think that it must oversee all the streams. This can be another reason that the overall latency is much higher.

Flink is much faster on finding a stable latency after the transition. However, in Figure 4.10 it seems that some occasional cubes receive a huge latency compared with their neighbors. A possible explanation for this could be that a specific thread is blocked by the scheduler due to another task having priority, and is not woken up for a longer duration of time.

In Figure 4.11 a more clear view of Flink's stability is presented (the highest latency value on position 910 has been reduced to give the graph a better ratio). Here we see that directly on cube number 1000 the latency jumps up, and quite fast, after ten cubes, finds its way back to a more stable latency. Then, after additional 500 cubes it jumps up in latency again. So even if Flink does a step-wise increase in latency, its much faster on transitioning and finding a new stable latency than RaftLib, which has a more successive increase. While RaftLib did the transition under e.g. 120 cubes, Flink did it in approximately 10 cubes as seen in Figure 4.12.

With the result gained from the data granularity use case in Section 5.4, we saw that the size of the data actually correlated with the performance of the frameworks. By knowing this, the stability use case was run a second time, this time with smaller data to see if the actual result matched with the expectation. The answer was yes, Flink performed even better than RaftLib with smaller data, which according to Section 5.4 should be the case. Flink had around 100 times faster latency than RaftLib with the smallest cubes and 60-80 times higher than the larger cubes when looking at random picked values in the graphs. For example, looking at corresponding values of Figure 4.15 and 4.16 the following was observed: Element 860 (smaller cubes with a dimension of 8x8x8) in Flink had a latency of 26,113$\mu$s while in Raftlib having a latency of 2767$\mu$s, being $\frac{2767}{26,113} \approx 106$ times higher.

The expected behavior of the fluctuation shown in Figure 3.4 correspond quite well with the actual result, specially for Flink. For RaftLib the actual result was worse than the orignal expectation.

## 5.6 Elasticity

Elasticity was one big reason why stream processing seemed like an interesting field to study. The possibility to have automated dynamic graph construction and resource allocation during run-time was very tempting. The instructions and documentation for RaftLib stated that auto scaling was supported. However when we tried to implement it in RaftLib by following the instructions we did not get it to work. Since other functionality in RaftLib was either only half-implemented or lacking completely we assumed that this was the case for auto scaling as well.

As mentioned in Section 1.4.2 Flink needs to restart and boot from a savepoint in order to increase the parallelism of a streaming job, not making it optimal when applied to e.g. a flight radar that continuously receives data blocks that need to be processed. The other dynamic resource allocation in Flink was associated with Yarn and Mesos as mentioned in Section 1.4.2, and was not tested due to being released towards the end of our study.

Furthermore, to test the efficiency of elasticity to any useful degree a more advanced use case with a more complex graph would have to be developed. We had initially planned for such a use case, but were forced to abandon the idea due to time considerations.

## 5.7 Engineering efficiency

From the results presented in Table 4.1, several conclusions have been drawn.

*Learnability* was the first engineering efficiency metric evaluated. RaftLib was relatively easy to start learning. With some available example programs and instructions from the documentation, basic programs could be implemented. Also, since RaftLib is mostly C++, the learning curve for the API was quite fast. The most problems were encountered later on, when working with more complex graphs and functionality, like dynamic parallelization as well as a general lack of tutorials.

With Flink this was the opposite. With an extensive API that needed to be learned, there was a much steeper learning curve than RaftLib, making it harder to get a quick understanding of the framework. However, once the basics were covered, Flink offered both more support through the community and available tutorials from their web page and the information needed was much easier to retrieve.

*Usability* was the second metric covered, and one of the more important. RaftLib is very much like working with C++ in general, with the parallelization simplified to a great extent. Whether that is a good thing depends on how good a certain developer is at C++, but in general our experience was that RaftLib did not add very much in addition to this simplified parallelism.

Flink has an extensive API that can be used at several levels of abstraction. We only worked with a small subset of the Datastream API, but that proved more than expressive enough for our purpose. It was easy to manipulate the parallelism and create different topologies. The code was very compact and easy to follow.

The *understandability* for both frameworks were quite high. The understandability for RaftLib basically depend a on the understandability of C++ itself, since the structure given by RaftLib is quite easy to understand. Once the API in Flink is learned, the code is compact, simple and clear to follow, making it easy to understand it.

There is a big gap at the moment between the two frameworks when it comes to their *sustainability*. Flink is being actively developed by The Apache Software Foundation and have several years of experience within stream and big data processing. They also have a large and thriving community and many large commercial users, giving them a quite secure foreseeable future. In contrast, RaftLib is quite young, with mostly one main contributor and creator, who seems to have taken a break from working on RaftLib. This leaves no guarantees that the framework will have continued development over the next upcoming years. RaftLib's community seems to be inactive and the framework is not being used by any companies as we know of, giving it an uncertain future. So far Flink is also supporting backward compatibility with earlier developed frameworks, like Storm. This indicates that Apache understand that their commercial users have need for long lived frameworks and stable APIs.

Looking at lines of code, Flink was drastically better in our test cases. Overall it had about four times less code than RaftLib when it came to a simple matrix multiplication application. Flink also had fewer rows than the benchmark, with an approximate of 25 rows of code against 55 rows. RaftLib however, due to lots of boilerplate code, was worse than the benchmark, landing on 108 lines of code. This is probably for being a quite small and easy example, and can probably be much more efficient in larger applications. Jonathan Beard has written a parallel bzip2 application in RaftLib to compare lines of code with a more traditional implementation of it [5]. The traditional implementation contains more than 4500 lines of code, while the RaftLib implementation landed around 240 rows. However, this was the first time we were introduced to both Flink and RaftLib, and the implementation could probably be even more optimized.

## 5.8 Incorporating TensorFlow

While testing TensorFlow it became apparent that it is very good at linear algebra operations. Since it lacked the capabilities of more complete stream processing frameworks, we instead tried to incorporate TensorFlow into our candidate frameworks. However, building and exporting a TensorFlow library for use in other frameworks proved difficult. While looking for solutions it became obvious that many developers are encountering the problem of importing TensorFlow functionality into their projects. Activity on programming message boards and the TensorFlow GitHub repository indicate Google is actively working on a solution, but as of early 2018 this is not solved. Forum posts by Google employees suggest that it is possible the situation has changed toward the end of 2018, which would make it feasible to incorporate tensors and TensorFlow linear algebraic operations into existing stream processing frameworks with little effort. When this is the case TensorFlow can be used to perform calculations in a SIMD (Single Instruction, Multiple Data) manner on individual cores, using another streaming framework to exploit the MIMD (Multiple Instructions, Multiple Data) space to spread the tensor operations to many cores. This would likely yield good performance.

# Chapter 6
# Conclusions

The initial scope was to study what could be the next generations signal processing architecture, and implement a prototype of a stable API. The API needed to be both engineering efficient as well as being able to handle the high throughput, low latency and the high complexity requirements that modern signal processing systems requires. As mentioned earlier, this scope was narrowed down to investigate to what extent complete stream processing frameworks could solve these requirements. To this end, a survey was carried out and two candidate frameworks promising qualities were chosen for further examination: the C++ template library RaftLib, and Apache's Flink.

RaftLib is a framework that is well suited for single-machine parallelization with good performance in most use cases. It is simple to learn and easy to work with. When the amount of worker kernels reaches the limit of available cores, the framework experiences a decrease in performance due to the use of supervisory threads. This needs to be kept in mind to get the best possible performance. The streams seem to have a relatively high latency, which could be a disqualifying factor for latency sensitive applications such as signal processing. Furthermore, it is a small open-source project with having both few developers and a small community, both the developers and the community seems a bit inactive giving the framework an uncertain future.

Flink, a well supported framework developed and updated continuously by The Apache Software Foundation, a large corporation. The community is large, thriving and active and is also used by commercial users. It has a rich API and it is very easy to implement parallel applications in it once the initial learning phase is passed. It is mostly used on distributed clusters of servers, but it also has a good single-machine performance with very low latency streams. Overall Flink seems like a suitable candidate framework for future projects exploring the intersection of signal and stream processing.

More and more stream processing frameworks like Flink and RaftLib are actively being ex-

plored and developed, and stream processing in general is growing for every year. Stream processing frameworks do seem to have potential to handle and satisfy the high demands of an AESA-radar. However, to be able to give a solid conclusion, further and more correct tests with more realistic data needs to be made.

# Chapter 7
# Future work

In general stream processing is an interesting and growing field with several interesting areas. For example, see how stream processing languages applied to different kinds of hardware, e.g. FPGAs, as presented in [36], can contribute towards increased performance within signal processing chains. As well as if the use of stream processing architectures with programmable stream processors can aid in increasing performance and flexibility within signal processing, as mentioned in [27].

Continued work regarding stream processing frameworks needs to be done. For example by implementing more realistic signal processing chains to see how the frameworks fulfill the requirements of an AESA-radar. Also test with more realistic hardware architectures, e.g. running with a larger cluster or by adding GPUs or FPGAs. Additional frameworks not tested in this thesis could be tested more thoroughly, e.g. WallarooLabs Wallaroo.

At first the plan was to continuously update and add functionality to the benchmark in parallel with the frameworks to simulate a more complex signal processing system. This could be a good idea to do, but with the given time frame it would be too stressful. This resulted in only using the benchmark to compare slowdown, scalability and engineering efficiency in terms of code in one of the use cases.

Adding external libraries for e.g. linear algebra operations to the frameworks to see how they collaborate with them. As mentioned in Section 5.8, an interesting avenue of exploration would be to incorporate Google's TensorFlow library into a full-fledged stream processing framework. Google is working on making it possible to export TensorFlow functionality as a library in the near future. TensorFlow has very good linear algebraic performance. Combining this performance with an engineering efficient and stable stream processing framework could result in a complete, stable and efficient solution.

# Bibliography


[1] Anders Åhlander, Anders Åström, Bertil Svensson, and Mikael Taveniku. Meeting engineer efficiency requirements in highly parallel signal processing by using platforms. In *International Conference on Parallel and Distributed Computing Systems*, pages 693–700, Nov. 2005.

[2] Data Artisans. What is stream processing. https://data-artisans.com/what-is-stream-processing, Accessed: 27 June 2018.

[3] Jonathan Beard. Raftlib: Simpler parallel programming. https://www.youtube.com/watch?v=IiQ787fJgmU#t=1h02m25s, Last edited: 12 June 2017, Accessed: 15 May 2018.

[4] Jonathan Beard. A short intro to stream processing. http://www.jonathanbeard.io/blog/2015/09/19/streaming-and-dataflow.html, Accessed: 18 September 2018.

[5] Jonathan C. Beard. Simplifying parallel applications for c++, an example parallel bzip2 using raftlib with performance comparison. https://medium.com/cat-dev-urandom/simplifying-parallel-applications-for-c-an-example-parallel-bzip2-using-raftlib-with-performance-f69cc8f7f962, Last Edited: Jan 10, 2016, Accessed: 29 June 2018.

[6] Jonathan C. Beard. *Online Modellig and Tuning of Parallel Stream Processing Systems*. PhD thesis, Washington University in St. Louis, 8 2015.

[7] Jonathan C. Beard. Accessing data from within a kernel. https://github.com/RaftLib/RaftLib/wiki/Accessing-data-from-within-a-kernel, Accessed: 12 June 2018, 10 2017.

[8] Jonathan C. Beard. Building kernels. https://github.com/RaftLib/RaftLib/wiki/Building-Kernels, Accessed: 09 September 2018, 10 2017.

[9] Jonathan C. Beard. Linking kernels. https://github.com/RaftLib/RaftLib/wiki/Linking-Kernels, Accessed: 12 June 2018, 10 2017.

[10] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: A c++ template library for high performance stream parallel processing. *International Journal of High Performance Computing Applications*, 2016.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink tm : Stream and batch processing in a single engine. 2016.

[12] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Proceedings of the 2008 Symposium on Application Specific Processors*, SASP '08, pages 101–107, Washington, DC, USA, 2008. IEEE Computer Society.

[13] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 362–376, Berlin, Heidelberg, 1974. Springer-Verlag.

[14] Per Ericsson and Anders Åhlander. Requirements for radar signal processing. Company unclassified, technical report 6/0363-FCP1041180 en, Saab AB, 2010.

[15] The Apache Software Foundation. Apache hadoop yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, Accessed: 05 July 2018.

[16] The Apache Software Foundation. Apache spark. https://spark.apache.org/, Accessed: 2 July 2018.

[17] The Apache Software Foundation. Apache storm. http://storm.apache.org/, Accessed: 2 July 2018.

[18] The Apache Software Foundation. Dataflow programming model. https://ci.apache.org/projects/flink/flink-docs-release-1.5/concepts/programming-model.html, Accessed: 27 June 2018.

[19] The Apache Software Foundation. Distributed runtime environment. https://ci.apache.org/projects/flink/flink-docs-release-1.5/concepts/runtime.html, Accessed: 27 June 2018.

[20] The Apache Software Foundation. Introduction to apache flink. https://flink.apache.org/introduction.html, Accessed: 27 June 2018.

[21] The Apache Software Foundation. Mesos, a distributed systems kernel. http://mesos.apache.org/, Accessed: 05 July 2018.

[22] The Apache Software Foundation. State backends. https://ci.apache.org/projects/flink/flink-docs-release-1.5/ops/state/state_backends.html, Accessed: 27 June 2018.

[23] Google. Tensorflow. https://www.tensorflow.org/, Accessed: 2 July 2018.

[24] Hendarmawan Hendarmawan, Mpho Gololo, Qian Zhao, and Masahiro Iida. High level stream processing with fpga. In *The 11th International Student Conference on Advanced Science and Technology ICAST 2016*, 12 2016.

[25] Fabian Hueske. Apache flink 1.5.0 release announcement. https://flink.apache.org/news/2017/02/06/release-1.2.0.html, Last Edited: 25 May 2018, Accessed: 05 July 2018.

[26] WallarooLabs Inc. Wallaroo. https://www.wallaroolabs.com/, Accessed: 2 July 2018.

[27] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, Jung Ho Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, Aug 2003.

[28] Z. Karakaya, A. Yazici, and M. Alayyoub. A comparison of stream processing frameworks. In *2017 International Conference on Computer and Applications (ICCA)*, pages 1–12, Sept 2017.

[29] Robert Metzger. Announcing apache flink 1.2.0. https://flink.apache.org/news/2017/02/06/release-1.2.0.html, Last Edited: 06 February 2017, Accessed: 05 July 2018.

[30] Robert Metzger. Memory management in streaming mode. http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/Memory-Management-in-Streaming-mode-td13317.html, Last Edited: 29 August 2016, Accessed: 27 June 2018.

[31] Milos Nikolic, Badrish Chandramouli, and Jonathan Goldstein. Enabling signal processing over data streams. https://blog.acolyer.org/2017/08/10/enabling-signal-processing-over-data-streams/, Last Edited: 10 August 2017, Accessed: 2 July 2018.

[32] Milos Nikolic, Badrish Chandramouli, and Jonathan Goldstein. Enabling signal processing over data streams. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 95–108, New York, NY, USA, 2017. ACM.

[33] The Open MPI Project. Portable hardware locality (hwloc). https://www.open-mpi.org/projects/hwloc/, Last Edited: 8 June 2018, Accessed: 4 July 2018.

[34] RocksDB. A persistent key-value store for fast storage environments. https://rocksdb.org/, Accessed: 06 July 2018.

[35] Matthias J. Sax. Storm compatibility in apache flink: How to run existing storm topologies on flink. https://flink.apache.org/news/2015/12/11/storm-compatibility.html/, Accessed: 2 July 2018.

[36] J. Serot, F. Berry, and S. Ahmed. Implementing stream-processing applications on fpgas: A dsl-based approach. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 130–137, Sept 2011.

[37] Dragan Stancevic. Zero copy i: User-mode perspective. https://www.linuxjournal.com/article/6345?page=0,0, Accessed: 20 June 2018, 1 2003.

[38] Fabian Hüske The Apache Software Foundation. Introducing stream windows in apache flink. https://flink.apache.org/news/2015/12/04/Introducing-windows.html, Last Edited: 04 Dec 2015, Accessed: 27 June 2018.

[39] Fabian Hüske The Apache Software Foundation. Juggling with bits and bytes. https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html, Last Edited: 11 May 2015, Accessed: 27 June 2018.

[40] Qian Zhao, Motoki Amagasaki, Masahiro Iida, Morihiro Kuga, and Toshinori Sueyoshi. A study of heterogeneous computing design method based on virtualization technology. *SIGARCH Comput. Archit. News*, 44(4):86–91, January 2017.

# Appendices

# Appendix A
# Test environment

**Custom BIOS settings:**
Inter® Turbo Boost Technology: Disabled.
Enhanced Intel SpeedStep® Technology: Disabled.
ACPI C2/C3 & ACPI C3 Disabled.
Intel® Hyper-Threading Technology: Disabled.

| | |
|---|---|
| Architecture | x86_64 |
| CPU op-mode (s) | 32-bit, 6-bit |
| Byte Order | Little Endian |
| CPU(s) | 16 |
| On-line CPU(s) list | 0-15 |
| Thread(s) per core | 1 |
| Core(s) per socket | 8 |
| Socket(s) | 2 |
| NUMA node(s) | 2 |
| Vendor ID | GenuineIntel |
| CPU Familiy | 6 |
| Model | 45 |
| Model name | Intel(R) Xeon(R) CPU E5-2650 0 @2.00GHz |
| Stepping | 7 |
| CPU max MHz | 2000.0000 |
| CPU min MHz | 1200,0000 |
| BogoMIPS | 3990,48 |
| Virtualization | VT-x |
| L1d cache | 32K |
| L1i cache | 32K |
| L2 cache | 256K |
| L3 cache | 20480K |
| NUMA node0 CPU(s) | 0-7 |
| NUMA node1 CPU(s) | 8-15 |
| Flags | fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx lahf_lm epb pti retpoline tpr_shadow vnmi flexpriority ept vpid xsaveopt dtherm arat pln pts |

**Table A.1:** Table showing the test server specifications.

# Appendix B

# RaftLib example code

```
#include <raft>
#include <raftio>
#include <iostream>
#include "matrix.h"

using namespace std;

int nr_ports = 4;
const int nr_matrices = 100;
Matrix data_input_stream[nr_matrices];
size_t gen_count(0);

class producer : public raft::parallel_k
{
private:
Matrix matrix;
public:
    producer() : raft::parallel_k()
    {
        for(int i = 0; i < nr_ports; ++i)
        {
            addPortTo< Matrix>( output );
        }
    }
    virtual raft::kstatus run()
    {
        if (gen_count < nr_matrices) {
            for( auto &port : output )
```

```
            {
                if (gen_count<nr_matrices){
                    matrix = data_input_stream[gen_count++];
                    auto out( port.template allocate_s< Matrix >() );
                    (*out) = std::move( matrix );
                }
                else { break; }
            }
            return( raft::proceed );
        }
        return( raft::stop );
    };
};

class processor : public raft::kernel
{
public:
    processor() : raft::kernel()
    {
        input.addPort< Matrix >( "in" );
        output.addPort< Matrix >( "out" );
    }

    processor(const processor &proc)
    {
        input.addPort< Matrix >( "in" );
        output.addPort< Matrix >( "out" );
    }

virtual ~processor() = default;

    virtual raft::kstatus run()
    {
Matrix mat;
        mat = input[ "in" ].peek< Matrix >();
        Matrix result;
        result.matrix_mul(mat, mat);
        input[ "in" ].recycle();

        auto out( output["out"].template allocate_s< Matrix >() );
        (*out) = std::move( result );
        return( raft::proceed );

    }
    CLONE();
};
```

```
class consumer : public raft::parallel_k
{
private:
    Matrix mat;
public:
    consumer() : raft::parallel_k()
    {
        for(int i = 0; i < nr_ports; ++i)
        {
            addPortTo< Matrix >( input );
        }
    }

    virtual raft::kstatus run()
    {
        for( auto &port : input )
        {
            try {
                mat = port.peek< Matrix >();
                mat.matrix_print();
                port.recycle();

            } catch(ClosedPortAccessException) { break; }
        }
        return( raft::proceed );
    }
};

int  main()
{
    for(int i = 0; i < nr_matrices; ++i){
        Matrix mat;
        mat.generate_matrix();
        data_input_stream[i] = mat;
    }
    producer    a;
    processor   b;
    consumer    c;

    raft::map m;
    m += a <= b >= c;
    m.exe();

    return( EXIT_SUCCESS );
}
```

# Appendix C

# Flink example code

```
/**
* Stream application
*/
package org.apache.flink.quickstart.ThesisCode;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.
    StreamExecutionEnvironment;

public class MatrixMulExample {
        public static void main(String[] args)
            throws Exception {

            final StreamExecutionEnvironment env =
                    StreamExecutionEnvironment.
                        getExecutionEnvironment();

            DataStream<Matrix> matrixStream =
                env.addSource(new MatrixSource())
                    .map(new MapFunction<Matrix, Matrix>() {
                        @Override
                        public Matrix map(Matrix matrix)
                            throws Exception {
                            return matrix.matrixMul();
                        }
                    })
                    .setParallelism(4);
```

```
            matrixStream.print();
            env.execute("Matrix Stream of Sweetness");
        }
}

/**
* Stream source
*/
package org.apache.flink.quickstart.ThesisCode;

import org.apache.flink.streaming.api.functions.
    source.SourceFunction;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class SourceMatrices implements SourceFunction<Matrix> {
    private volatile boolean isRunning = true;
    ArrayList<Matrix> matrices = new ArrayList<>();
    private static final int matrixDim = 100;
    private static final int nrMatrices = 4000;
    private int index = 0;
    private volatile long startTime;

    @Override
    public void run(SourceContext<Matrix> ctx) throws Exception {
        ReadFromFile("/home/exjobbarna/matrices.txt", matrices);

        Matrix matrix;
        MatrixMul.setStartTime(System.currentTimeMillis());

        for(int i = 0; i < nrMatrices; i++) {
            matrix = matrices.get(index++);
            index = index % 4000;
            ctx.collect(matrix);
        }
    }

    @Override
    public void cancel() {
        isRunning = false;
    }
}
```

# Appendix D

# Benchmark code

```
#include <iostream>
#include <sys/time.h>
#include <vector>
#include <fstream>
#include <sstream>
#include <iterator>
#include "matrix.h"

/** The number of matrices computed over. **/
#define NBR_OF_MATRICES 4000
/** The number of runs done. **/
#define NBR_OF_RUNS 100
using namespace std;

/* Mutex lock of the index couter of the vector
    containng the matrices. **/
pthread_mutex_t is_counter_lock;
int is_counter = 0;
Matrix data_input_stream[NBR_OF_MATRICES];

/** Wall time function returning the current time in seconds. **/
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL))
return 0;
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}
```

```
/** Prints a string to file. **/
void print_results_to_file(string results, string file_name){
cout << "Printing results to file: " << file_name << endl;
ofstream myfile;
myfile.open (file_name);
myfile << results;
myfile.close();
cout << "Done with printing to file." << endl << endl;
}

/** Generates and then prints the matrices to file for future
    runs with identical input. **/
void print_matrices_to_file(int nr_of_matrices, string file_name){
cout << "Printing matrices to file.." << endl;
ofstream myfile;
myfile.open (file_name);

for(int i = 0; i < nr_of_matrices; i++)
{
Matrix mat;
mat.generate_matrix();
string mat_string = mat.to_string();
myfile << mat_string;
}
myfile.close();
cout << "Done with printing to file." << endl;
}

/** Reads matrices from file and stores them in an array. **/
void read_matrices_from_file(std::string file_name) {
cout << "Saving matrices to memory.." << endl;
ifstream myfile;
string line;
myfile.open (file_name);
Matrix m;
int row = 0;
int idx = 0;
while(std::getline(myfile, line)) {
stringstream ss(line);
istream_iterator<string> begin(ss);
istream_iterator<string> end;
vector<string> vstrings(begin, end);

for (int col = 0; col < vstrings.size(); ++col)
{
int value = stoi(vstrings[col]);
```

```
m.set_value(row, col, value);
}
row++;
if (row == m.dim())
{
if (idx >= NBR_OF_MATRICES)
break;
data_input_stream[idx++] = m;
row = 0;
}
}
cout << "Done saving matrices." << endl << endl;
}

/** The PThreads matrix multiplication method.
    Locks the counter, store the counter_index,
    unlocks the counter and performs the
matrix multiplication on given index in the array. **/
void* mat_mul(void* arg)
{
while(true){
pthread_mutex_lock(&is_counter_lock);
if (is_counter < NBR_OF_MATRICES)
{
int index = is_counter++ % NBR_OF_MATRICES;
pthread_mutex_unlock(&is_counter_lock);
Matrix mat;

mat.matrix_mul(data_input_stream[index],
    data_input_stream[index]);
} else {
pthread_mutex_unlock(&is_counter_lock);
break;
}
}
}

/**
The main method, starting workers based on parameters:
-Minimum worker threads
-Maximum worker threads
-increment number.
The main runs the application NBR_OF_RUNS times
for each worker thread setup.
**/
int main (int argc, char* argv[])
```

```
{
if (argc < 4)
{
        // Tell the user how to run the program
        std::cerr << "Usage: " << argv[0] << " min_num_workers "
<< "max_num_workers " << "worker_increment " << std::endl;
        return 1;
    }
int min_nr_workers = atoi(argv[1]);
int max_nr_workers = atoi(argv[2]);
int worker_incr = atoi(argv[3]);

cout << "RUNNING: BENCHMARK PERFORMANCE - MINIMAL." << endl << endl;
//print_matrices_to_file(NBR_OF_MATRICES, "matrices.txt");
read_matrices_from_file("/home/exjobbarna/matrices.txt");
pthread_mutex_init(&is_counter_lock, NULL);
srand(time(NULL));

/** Prints **/
cout << "SIZE OF INPUT DATA STREAM: " << NBR_OF_MATRICES << endl;
cout << "NUMBER OF RUNS: " << NBR_OF_RUNS << endl;
cout << "NUMBER OF WORKERS: FROM " << min_nr_workers << " TO "
    << max_nr_workers << endl << endl;
string result_string;

/** Run the application with different worker thread setups.
    Min to Max with increment. **/
for(int NUM_THREADS = min_nr_workers; NUM_THREADS
    <= max_nr_workers; NUM_THREADS += worker_incr)
{
vector<double> times(NBR_OF_RUNS);
double avg_run_time = 0;
  double varianceSum = 0;

/** Run the specific setup NBR_OF_RUNS times. **/
for (int i = 0; i < NBR_OF_RUNS; ++i)
{
/** Declare threads. **/
pthread_t* threads = new pthread_t [NUM_THREADS];
int rc;
/** Lock the counter so the computation
begins after the last thread is started. **/
pthread_mutex_lock(&is_counter_lock);
/** Start the threads. **/
for (int j = 0; j < NUM_THREADS; ++j)
{
```

```
if (rc = pthread_create(&threads[j], NULL, mat_mul, NULL))
throw rc;
}
/** Unlock the counter and start the walltime. **/
pthread_mutex_unlock(&is_counter_lock);
double start_wall_time = get_wall_time();
for (int j = 0; j < NUM_THREADS; ++j)
{
int a = pthread_join(threads[j], NULL);
}
/** Stop walltime and get run-time. Then reset/delete threads. **/
double end_wall_time = get_wall_time();
delete threads;

/** Store run-times for average run-time calculation below.
Reset all parameters. **/
is_counter = 0;
times[i] = end_wall_time - start_wall_time;
cout << "Run " << i+1 << ": " << times[i] << " : " <<
    NUM_THREADS << "\n";
avg_run_time += times[i];
}

/** Calculate all average run-times and variances,
save to string and print/store to file. **/
avg_run_time /= NBR_OF_RUNS;

ostringstream avg_strs, num_thread_strs, variance_ss;
string variance_s;

avg_strs << avg_run_time;
string avg_str = avg_strs.str();

num_thread_strs << NUM_THREADS;
string nr_threads = num_thread_strs.str();

for(int i = 0; i < times.size();++i)
{
    double diff = times[i]-avg_run_time;
    varianceSum += (diff*diff);
}
double variance = varianceSum / times.size() ;

variance_ss << variance;
variance_s = variance_ss.str();
```

```
result_string += avg_str + " " + nr_threads + " " +
    variance_s + "\n";

cout << endl << "Avg run-time: " << avg_run_time;
cout << endl << "Variance: " << variance << endl;
}
print_results_to_file(result_string,
    "/home/turbin/results/results_bench_performance_min_40kmats.txt");
return 0;
}
```

# Appendix E
# List of Changes

### Since 2018/08/31

- Updated the template for the report.
- Added an additional appendix chapter, the "List of Changes".
- Moved the chapter "Candidate frameworks" before the "Approach" chapter according to comments from the opponents.
- Removed the explanation of all different window types for Flink in Section 3.2.4 "Windows", according to comment from opponent.

### Since 2018/09/6-9

- Updated language in the report according to the examiner's notes.
- Removed some unrelated sections in Related work.

### Since 2018/09/10-16

- Updated language in the report according to the examiner's notes.
- Edited and added more relations between requirements, metrics and use cases, according to the examiner.
- Restructured sections in the approach, result and discussion chapters to make them follow the same pattern.
- Added changes to the results based on comments from the examiner, i.e. discussed the individual results more.
- Did updates to the report in general based on examiner's comment.

## Since 2018/09/18-19

- Fixed references to be IEEE alphabetically format.
- Moved Chapter "Candidate frameworks" to introduction as a section and renamed it "Stream processing frameworks".
- Replaced a Wikipedia reference.
- Removed/changed section titles in Related work to give it a better flow, also added more explanations on how they relate to this project.

**EXAMENSARBETE** High-performance signal processing for digital AESA-radar
**STUDENT** Alexander Olsson (LTH)
**HANDLEDARE** Jörn Janneck (LTH)
**EXAMINATOR** Flavius Gruian (LTH)

# Flight-radar Processing Using Streaming Frameworks

POPULÄRVETENSKAPLIG SAMMANFATTNING **Alexander Olsson (LTH)**

The demands on signal processing for digital flight-radars are rising rapidly. To solve these demands, stream processing frameworks are evaluated. These frameworks are an interesting and engineer efficient solution to help reduce development time, as well as keeping a good performance.

The lifespan for a typical radar application is around 15-20 years. This translates to several generations when it comes to the lifespan of technology. Therefore it is important to have a stable and engineering efficient API, which does not need to be updated every time the hardware is being improved.

Apart from this, the demands for fast and correct information in flight-radars sets high requirements on the system. For a potential target, e.g. another airplane, to be discovered and evaluated by the radar, lots of computations needs to be performed. To successfully do this parallel hardware architectures are being used, performing several operations concurrently.

What makes frameworks interesting when it comes to radar applications, is that they can scan the hardware and be able to adapt itself accordingly. Making them a suitable choice for long lived APIs. They also assist with the parallelization, thereby contributing towards engineering efficiency.

The frameworks belong to a fairly old computer paradigm called stream processing. Stream processing focuses on handling data as soon as it arrives to the system, as well as being effective on moving data. Complete stream processing frameworks focusing on real-time signal processing are evaluated, and noted to be an interesting solution to solve the issues raised by flight-radars.

The main frameworks covered are: RaftLib - a template library for C++ and Apache's Flink - a Java/Scala library. Flink is showing most promise of the two, and RaftLib also being an interesting candidate, but being a bit undeveloped with an unsure future.

Other interesting areas within stream processing are also covered and can be read in the thesis; High-performance signal processing for digital AESA-radar by Alexander Olsson, dat12aol@student.lu.se - Faculty of Engineering, LTH.