Otto-von-Guericke-Universität Magdeburg
Faculty of Computer Science

Accenture GmbH

Master's Thesis

The Role Of GPUs In Big Data Scientific Software Using BAXdem

Author:

Apoorva Patrikar

February 09, 2018

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake,
M.Sc. Gabriel Campero and M.Sc. Roman Zoun

Databases and Software Engineering Workgroup,
University of Magdeburg

Dipl. Kay Schallert

Chair of Bioprocess Engineering, University of Magdeburg

Ph.D. Andy Bruntsch and M.Sc. Lukas Feuerstein

Accenture GmbH

Patrikar, Apoorva:
The Role Of GPUs In Big Data Scientific Software Using BAXdem
Master's Thesis, Otto-von-Guericke-Universität Magdeburg
Faculty of Computer Science, 2018.

Abstract

Trends show a shift from batch to stream processing, less to more parallel hardware, from historical analytics to real-time analytics, big data to fast data frameworks. Despite the interest in these trends, there is limited research on how they affect each other. In this project we consider such line of inquiry, seeking to bridge the gap between applications with big data components and high performance computing. We adopt a biological case study, which encompasses data analysis for protein identification. More specifically, we select the algorithm called X!Tandem and implement some of its core functions, while using scalable methods for processing retrieved data from a database and for storing theoretical spectra.

We implement an end-to-end analysis framework that includes: configurable filtering criteria for loading experimental spectra, pair-building logic for determining candidate peptides for scoring, the scoring of pairs, the collection of scores and their ranking. The original X!Tandem program was very carefully mimicked, while at the same time leveraging the scalable features of general big data frameworks. We run our end-to-end system prototype utilizing Cassandra, Kafka and Python jobs. We provide early experimental results highlighting the benefits of different configurations and the effect of capitalizing on GPU usage through the execution path, through OpenCL implementations. We report speedups from 53x to 6x comparing GPU vs. CPU scaled-out systems. We suggest a performance analysis methodology based on queueing theory, to guide the optimization process. Among our findings we observe that batching is highly beneficial and, in fact, necessary for processing huge amounts of data simultaneously on a GPU. It not only speeds up the process but also effectively utilizes the system resources. We also observe that the data-flow in the prototype can be affected by overheads and non-deterministic response times from Kafka and Cassandra.

As a result we establish that exploiting GPUs alongside big data is feasible and, relatively easy to implement. Furthermore for our implementation we find that hardware-acceleration can lead to larger improvements than scaling-out. However more improvements could be possible with support from the entire data processing architecture such that acceleration benefits are not hidden by overheads from other components. Thus, we conclude that hardware acceleration in big data frameworks is a holistic task. In this task it is necessary for optimizations to work in unison: hardware-tuning should be included both in the internal components of the supporting frameworks and in the application level, without them affecting one another.

Acknowledgements

I am excited, happy and proud to successfully present my thesis. However, this would not have been possible without support, guidance and encouragement from the people who helped me throughout the process. First and foremost, I'd like to thank Otto-von-Guericke University and Accenture for agreeing to this collaborative effort, and making it a success.

I start by thanking my university supervisor, Gabriel Campero, who believed in my potential and encouraged me to take up this topic, mentored me through the journey and was my guru. He patiently endured my silly questions and doubts, helped me run the tests when I could not be physically present in the university, explained every little detail, and supported me in every way that he could. I'd also like to thank Roman Zoun for introducing me to the confluence of Informatics and Biotechnology. Moreover, he is also responsible for creation of the FASTA loader tool in Java, for which I am more than grateful as it formed the bridge between the input and processing of this thesis. In addition, Roman also provided me with some very good feedback that helped me improve the quality of this thesis. Last but not the least, I'd like to express my gratitude to Kay Schallert for his expertise on the process, algorithm, equipments and terminologies of protein identification and X!Tandem.

At Accenture GmbH, I'd like to acknowledge the efforts of my supervisor in Munich, Lukas Feuerstein and my colleague Andy Bruntsch for accepting my ideas about this thesis and pushing me to pursue it. They assisted, corrected and improved my work. They also helped me get in touch with the experts on the relevant topics, that they met in their professional journey. Without their guided directions, this journey would have been much tougher for me.

Finally, I thank my mother, Prachi Patrikar, my family and my friends for patiently enduring my blabber of the topic, the difficulties I faced and nervous talks, and turned them upside down by believing in me and encouraging me to successfully finish what I started. I am really grateful to all the people mentioned, and those who I might have forgotten but helped me nevertheless.

Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.
Magdeburg, February 09th 2018

Apoorva Patrikar

Contents

List of Figures

List of Tables

Listings

1. INTRODUCTION

The decreasing price of sensors and the diversity of novel applications based on sensor networks, has contributed in making these technologies hugely popular nowadays under the umbrella term Internet of Things (IOT). Gartner[1] predicts that connected devices will grow in number, as they have in previous years, in both business and consumer sections. This growth is expected to be around 20-50 billion connected devices, not only by Gartner, but also by Cisco systems [8], Ericsson[2] and some others. This trend asks for adopting streaming techniques to foster real-time analytics of data [9][10] and specialized techniques to foster the management of large amounts of fast data, and real-time analysis of it. Some of these techniques include stream processing, and the use of scalable big or fast data frameworks.

Springer's Encyclopedia of Database Systems defines streaming as "A process involving continuous data stream processing for filtering, aggregation, correlation, transformation, pattern matching and discovery, and domain-specific temporal analytics" [11]. In addition, it also states the requirements of streaming applications: high throughput, low latency, fault tolerance and some persistence capabilities. Streams are well known for event processing and logging [12], however, they are used for other purposes such as data delivery in real-time environments.

A definition of "big data" can be: data which is high in volume, velocity and/or variety, that traditional tools are unable to handle [13]. These days, the Velocity-V of the 3 Vs of big data is growing rapidly. A new term called "fast data" is coined to express it. It is big data in essence, but with a far greater velocity of data production. The business value and novel opportunities derived from data freshness and up-to-date analysis, contribute to make this an important data management topic.

[1]Gartner Inc., "https://www.gartner.com/newsroom/id/3598917", February 7, 2017

[2]Ericsson's press conference: "CEO to shareholders: 50 billion connections 2020", online: https://www.ericsson.com/en/press-releases/2010/4/ceo-to-shareholders-50-billion-connections-2020

From an operational standpoint, this kind of compute-intensive analysis for fast data can be supported in two ways: either by developing a framework with components that can handle big or fast data and scaling out the application, or by using more powerful hardware and scaling up the system.

This work intends to explore the combination of hardware and software to enhance a big data streaming application.

1.1 Problem Description

Business processes need to be revised in a big data environment. Instead of gathering a certain amount of data to process, e.g. daily reports; interactive dashboards and up-to-the-minute analysis can be performed by end users over data as it is ingested by the system. This means that the system must support the processing of small individual data items, thus enabling real-time analysis. This requirement brings forth various data management challenges like:

1. Moving the processing logic from batch to tuple-at-a-time or micro-batches
2. Speeding up of core processing tasks such as data cleaning, pre-processing during ingestion, integration and analysis [14] [15]
3. Real-time integration in a heterogeneous environment[14]
4. Data privacy handling[14]
5. Handling imperfections in a data stream[15]

In this study we intend to focus on the solutions for the first two problems. The first problem is a classic change from batch to stream processing. The second one is more complex, since the speeding up of big data encompasses two possibilities (scaling out or leveraging specialized hardware), with trade-offs between them.

Streaming: Big Data Ingestion

Data streams are simply fast flowing data, that are received item by item. A stream continuously generates data without knowledge of how much data it is going to produce [10]. Springer's Database Encyclopedia defines data streams as follows [11]:
"A data stream S is an ordered collection of data items s1, s2,...,sn where items are continuously generated by one or more sources and processed by one or more components, which cannot control the order of arrival of these items."

Data streams can either be base streams (directly generated by the source) or derived (preprocessed). Streams are ingested and processed immediately. Streaming systems can broadly be classified as conventional and reactive [10]. Conventional streaming systems run for days, months and years with or without spikes, with or without crashes

[10]. A queue/buffer usually handles the rate of data production, with decisions about what to do when the buffer is full or to have an unbounded buffer but risk a very large queue resulting into a crash. Reactive streams on the other hand, have bidirectional communication between the producer and consumer and thus handle traffic in a bounded buffer.

When we talk about streaming in general, there are two scenarios involved:

1. Stream ingestion
   Data is fed into the system tuple-at-a-time (receiving one item at a time) or in some cases, in micro batches (receiving a very small number of items to process them as a unit). Tuple-at-a-time ingestion makes a truly real time system, whereas systems with micro-batch ingestions are near-real-time systems with a trade-off between real-time and batch systems. Well known examples of stream ingesting frameworks are Kafka (Section 2.5.1), Kinesis or Spark.

2. Stream transformation
   The stream needs to be transformed into another structure at real-time in order to be consumed. An incremental processing style handles this situation. Applications such as Flink, Spark Engine, Samza, Storm are notable on the market as candidates who make stream transformation and processing possible.

These concepts are further explained in Chapter 2.

Speed-up of Big Data Processing

As mentioned earlier, to process data faster in a big data environment, we can use either hardware acceleration or a big data framework.

Hardware Acceleration is the focus of High Performance Computing (HPC) i.e. delivering better computing performance by improving and/or increasing the hardware. On the other hand, Big Data distributions deal with processing huge amounts of data at a higher level and are generally built for clouds but not for hardware acceleration [16][17]. Therefore large scale applications often have to make a choice between the two. In an ideal situation, this is not a choice that should be made, as Big Data frameworks should be able to exploit the possibilities of modern hardware. This brings us to the question of why is it not generally done.

The key problem is that Big Data frameworks are built on very high-level tools as opposed to low-level languages such as C [1]. They focus on scaling out but not on exploiting the optimal use of better infrastructure like GPUs and other hardware accelerators, which is the focus of HPC. This makes it difficult to port to big data technologies, which operate on a higher level and less hardware focused. This situation is problematic for both fields: HPC (generally what scientific applications choose [18]) and Big Data frameworks.

More importantly, there is no comprehensive study on how GPUs can be integrated into big data frameworks for everyday users, specially when combined with the needs of streaming applications, and neither for how performance analysis should be done when considering this.

1.2 Motivation

While several organizations are moving from batch to stream processing, the problem of Big Data vs. HPC has not been entirely addressed because hardware acceleration for Big Data technologies is a relatively new possibility.

Motivation for technology

Big Data processing frameworks are relevant for almost every field of science these days. Hence there is a clear need of bridging the gap between HPC and Big Data processing technologies, to benefit scientific applications by making better use of hardware. This is also an interesting problem because it implies bringing a rigorous hardware-centric analysis into Big Data processing frameworks, leading to fine-tuned optimizations for HPC processing.

Using fast data frameworks in conjunction with modern hardware is interesting and important because streaming and scaling are beneficial to a system's speed, efficiency and performance. Moore's law indicates that hardware can be expected to become more and more parallel, requiring either specialized programs (what we use in this project), smarter parallelizing compilers, or runtime environments. Since streaming and HPC seem to be following different trends, efforts of cohesion and integration seem necessary (more details in chapter 2 -Section 2.2).

This gap between HPC and frameworks is what motivated us for this research. Another small motivation is that there are some extensions, libraries and programs that allow modern-hardware programming on Hadoop (explained in Chapter 7). However, this still involves hardware-specific programming which requires extensive knowledge, for example GPU programming which is then compiled and integrated with existing code automatically. We are motivated to explore alternatives for removing this drawback by studying existing technologies like Tensorflow (Section 2.5.3) which is run using Python, a modern language, and generates CUDA (Compute Unified Device Architecture) automatically in the background, with minimal code changes.

We propose one alternative to bridge the gap between the the technologies, or in the very least, make the choice disappear for a case study which will be explained in Chapter 3.

Motivation for case study

For our research we selected a practical scientific application which had requirements for scalable processing over massive data, with near real-time analysis. The case study is, specifically, the task of matching experimental to theoretical proteins based on their spectra. It is a task from the field of metaproteomics.

Metaproteomics is the study of proteins in living organisms. Tests, especially ones related to identification of proteins, are run on samples collected from the environment and matched against not only protein databases but also gene annotations, cell lysis (breaking of a cell membrane) information and data collected from other genome activities [19]. The problem with metaproteomics is that the structure of microbial communities is unknown and there are a thousand different species [20]. Therefore it needs to be matched with several databases [21] for sound analysis. Unfortunately, we cannot decrease the number of comparisons because then we risk losing potential match candidates. Therefore, most algorithms [22][23][24][19] do a repetitive search through multiple runs before analyzing results. However, even the best algorithm is limited to the data we have already collected [21]. Databases like NCBI and UniProt (in detail in Chapter 2) provide a comprehensive repository, but it is notably large in volume and using all of it requires heavy computations, several resources and is time consuming.

The selected case study aims to re-implement a part of a protein identification algorithm to make it suitable for scalable stream processing, and hence to execute it in a fast and efficient manner. Protein identification is a process where MS/MS spectra (explained in Section 3.1.2.1) of an obtained sample is matched against all relevant MS/MS spectra obtained from a protein database. The results of such identification are a score determining the closeness of the match, and are referred to as PSM[3] (Peptide Spectrum Match). Once this score is determined scientists, the end users of the application, can decide if the proteins are a match or further studies are required for matching them.

In our work we aim to create a proof-of-concept for the scientific of protein identification. Our implementation must be able to, with minimal enhancements, be moved to a cloud computing environment. More importantly, in doing so, we should seize the advantages of HPC, such as GPU processing. Since we cannot re-implement and transform an entire protein identification (we select X!Tandem) algorithm in the scope of this thesis, we will only offer a proof-of-concept that it can be done, with all the core logic of the algorithm. Another advantage of the architecture we use is, that we may plug in other protein search algorithms at a later stage, making it more versatile and adaptive. This can be done by replacing a simple component in our architecture (a program called scorer), with the logic of another desired search algorithm.

1.3 Objective

In this thesis we provide an evaluation on the state of affairs regarding streaming and hardware acceleration for big data processing frameworks. In order to offer technical insights we selected a specific use case (Protein Identification, as explained in Chapter 3). Based on our consideration of the requirements, we selected a specific technology stack and we proposed an architecture, details of which can be found in Section 4.1.

We present our implementation of this design, called BAXdem, which stands for Big Accelerated X!Tandem because we try to re-implement X!Tandem partially using big data tools, and accelerate it on the GPU using parallel programming and some libraries. The key contribution of our work is the implementation of a prototype for protein

[3]PSM in detail and pictorial representation: http://cores.imp.ac.at/protein-chemistry/faqs Section: What is PSM

identification which scores similarities between datasets in a hardware-accelerated big data environment.

In our evaluation we test the scalability (scale up and scale out) and acceleration of our prototype for protein identification. We evaluate our implementation on real world data, w.r.t. an implementation with and without hardware acceleration. We aim for this research to be able to contribute to the database community, specifically to the research for integrating general calculations (i.e., more than SQL) [25][26] or GPU acceleration into database applications [27][28] and to evaluate HPC-based fast data performance.

1.4 Research Questions

With this thesis, we wish to answer the following general questions:

1. Does an HPC-based fast data architecture bring benefits to a streaming system? Or, is it worthwhile for scientific applications to move from high performance computing to big data frameworks if the gap between the two technologies is reduced?

2. Which technique among scaling out and scaling up is more powerful for hardware accelerated big data systems?

1.5 Evaluation Method

This thesis is implementation based. Research conducted before the implementation focused on X!Tandem and the confluence of hardware acceleration and tools designed for big data. After defining the architecture for the prototype (BAXdem), we realize parts of X!tandem on it. We run the scoring component on a CPU and on a GPGPU, and then we compare the performance. To test its cloud compatibility, we scale and compare the results. With this comparison, it should be clear if HPC brings benefits. We also test small and big batches to evaluate the prototype better.

1.6 Structure Of Work

Chapter 2 contains all the technical information needed to understand the research in this thesis. It covers streaming, HPC, hardware-acceleration and big data processing.

Chapter 3 gives details about the experimental setup starting from input samples, their digitized output, that acts as the input to our system, processing and analysis of data, and the final product. It also provides context on protein databases.

The implemented prototype is explained in Chapter 4. The chapter explains the structure, components and working of the prototype. This section also justifies the selected software for architectural decisions of the components.

Results from the experiments are reported in Chapter 5. This chapter describes the test environment, test plan, and the results of running the prototype on the CPU and the GPU, and the effect of scaling and batching on the speed of data processing.

Chapter 6 further analyses the results obtained in Chapter 5 describes the performance criteria and rates BAXdem on them. It also enlists the factors affecting the evaluation of the selected case study.

Chapter 7 describes similar work done in the case study as well as the new outlook of a mixed high level software-hardware framework.

Finally we conclude and propose future work in Chapter 8.

2. TECHNICAL BACKGROUND

Since this Thesis adopts a biological case study (protein identification) to examine the role of modern hardware-accelerated big data processing, it is beneficial to understand both the IT and biological background of the technologies and entities involved.

In this section software and hardware components related to the thesis and use case are described. The structure of this section is as follows:

- In Section 2.1 we discuss streaming systems. We define streaming and describe the architecture of different types of streaming systems, and the components therein. After that, the applications that allow end users to perform stream transformation and processing, are elucidated and compared.
- In Section 2.2 we introduce the gap between big data and HPC research. The section reports the technical differences between the two technologies. It also explains why most scientific applications need to choose between hardware acceleration and big/fast data frameworks.
- Section Section 2.3 introduces hardware acceleration. We present the core ideas of devices like FPGAs, GPUs, APUs and MICs.
- Finally, considering that GPUs are the focus of this thesis, we provide a complete description of their main features, including programming frameworks and related memory models in Section 2.4.

2.1 Streaming

The promise or focus of big data is to answer questions based on aggregated or collected i.e. offline or batch data. This is usually accomplished using Hadoop's map (distribute) reduce (collect) approach to large scale processing. This is an example of

Bulk Synchronous Parallelism, an approach where parallel execution is interleaved with synchronization steps, which mark the time of the system. Fast data on the other hand, utilizes streaming, which focuses on real-time production of data and results. Unlike BSP, streaming is based on a message-passing in a peer-to-peer architecture where producers and consumers need to reach a consensus on the relative order of messages received [9].

Authors have also proposed that large scale batch processing and stream processing can be further distinguished by their requirements for data storage (Figure 2.1 [11]). Batch processing assumes the complete input data to be present, often in the form of a database. Data is queried when required, and the database returns the results after filtering, altering (e.g. string operations, integer operations, conversion to specific formats) and/or aggregating the queried data. Processing of data in this case naturally happens when it is outbound.

On the other hand, streaming architectures make it possible to pre-process data items, prior to storing them into the database.

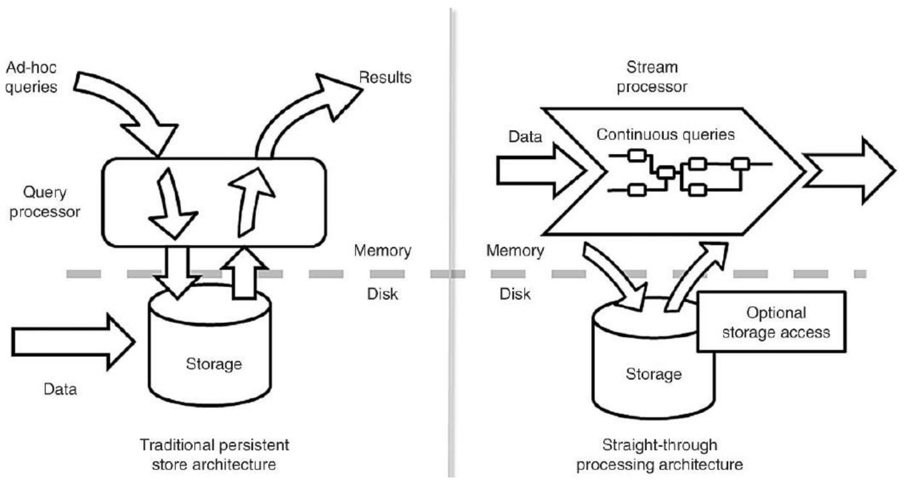

Figure 2.1: Storage-oriented and streaming-oriented architectures

Pre-processing data items saves time while returning results. In addition, the overhead of indexing is also reduced. The stream processor (Figure 2.1) processes an input tuple at a time, or a window of tuples as a unit. It contains a stream processing engine (SPE) that is comprised of the following [11]:

- Stream Manager:
  Interacts with the network layer and manages stream I/O as well as format compatibility for incoming data from sources and sinks
- Storage and queue manager:
  Handles memory resources, disk access and persistence of data in disk by maintaining cursors on external tables and performing asynchronous read and write operations during stream processing

- Queue Manager:
  Ensures the availability of memory resources and interacts with other system components to when availability is compromised
- Scheduler:
  Determines an execution strategy for queries and threads based on statistics from previous operations
- Query optimizer:
  Adapts queries to perform better, based on statistics of the execution engine and other monitoring results
- Continuous Query plans:
  As input data becomes available, the queries continuously execute.

Now that we have a brief idea about streaming and its two core applications: ingestion and transformation (Chapter 1).

2.1.1 Stream Ingestion

This section describes streaming in its entirety as a concept. Figure 2.2 gives an overview of the complete streaming process:

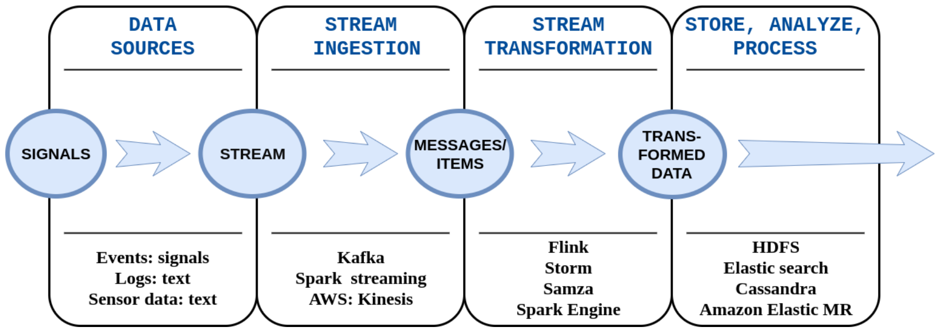

Figure 2.2: Stages of stream processing

Data comes from a source like events, logs, sensors; sensors and social media being the primary focus these days; when triggered by a signal or action. Then this data is fed as input by a system capable of handling real time data ingestion. This data, still on an item level, often needs to be modified into a format or structure that the target system will use. This is referred to as stream transformation. At this stage, the data stream is ready to be stored, analyzed or used for some benefits.

Based on the business requirements, a number of streaming solutions are available. Applications such as Kafka [29], Kinesis (Amazon Web server), Spark streaming among others, provide an interface for stream ingestion i.e., inbound streams. Stream transformation or modification is provided by Flink, Storm, Samza or Spark engine. Once the data stream is ready to enter the target system, it can be stored, processed or analyzed using HDFS (Hadoop Distributed File System), Elastic Search, Cassandra and others.

The choice between these applications depends on a number of factors like[1]:

- Delivery semantics for message processing
- Fault Tolerance and/or replication
- Committing and replaying of messages
- Message Windowing
- Message Ordering
- Community Support, Examples, Libraries and Code Reuse
- Processing Latency
- Analytics support

Delivery semantics for message processing refers to the guarantee of message delivery. Messages are either data, meta data or logs produced by the system. Delivery semantics can be one of the following:

1. Exactly once: As the name suggests, exactly once semantics means that a message is provided to the application exactly once. The message will not be replayed.
2. At most once: The message will reach the system once, regardless of failures. This poses a risk of losing messages.
3. At least once: It guarantees the delivery of every message. Duplicates may, however, be present.

The choice naturally depends on the criticality and handling capabilities of the system. Every streaming product comes with its own set of fault tolerance and backup methods. Depending on how critical the information is, or how well it can handle failures, the streaming application is selected.

Similar to delivery semantics, an application is selected based on its capability to persist a message or to receive it multiple times. Some applications such as Spark Streaming, with work on micro-batches for streaming, require a streaming application with message windowing functions. It may be beneficial to use an application with built-in windowing functions. If it is not critical to do so, custom implementation can also enable windowing.

[1]"A Survey of Big Data Streaming Engines", by Andrew Aslinger, 13.12.2013, http://owaaa.github.io/bigdata/streaming/ec2/amazon/architecture/2013/12/13/streaming-engine-survey.html

	Kafka	Spark	Kinesis
Streaming	True	Micro batching	True
Delivery	Exactly once	Exactly once	At least once
Advantages	Scalable, Durable, Reliable and Fast	Easy Hadoop Integration	Scalability, Easy administration, Easy Amazon AWS integration
Disadvantages	Self configuration, Business code to handle Exactly once semantics	Not truly real real-time, Small file problem and Higher latency	Expensive in comparison and Higher latency and throughput than Kafka

Table 2.2: Difference between commercial frameworks that provide stream ingestion

Not every message passing system is synchronous. If message orders are imperative and the synchronization overhead is not an issue then an application with message ordering can be chosen.

Community Support, Examples, Libraries and Code Reuse also play an important part in the selection process for a message passing system. Debugging is easier and faster when an application's community support is active, documentation is regularly updated, useful libraries are available, and code reuse is possible. Analytics support can either be an in-built function, provided by libraries, or by interfaces to external tools. One may choose a streaming application based on whether it has a legacy system for analytics and/or its compatibility with tools and libraries.

Time required to process a stream could be an important factor while selecting applications, based on whether the system is affected by certain latency. Table 2.2 shows the difference between commercial frameworks that provide stream ingestion.

Kafka (Section 2.5.1) and Amazon's Kinesis enable developers to program the application at a tuple-level making them truly real-time ingestion systems. Spark on the other hand is capable of creating micro batches, which is not a perfect real-time scenario, but close. This is not necessarily a bad thing. For most applications, this suffices.

Kafka and Spark guarantee that each message enters the system exactly once, whereas Kinesis guarantees it at least once. This means when duplicate messages cause no problem in the application, the choice is open. But if the application requires strictly unique messages, Kinesis is not an option.

Kafka is known to be scalable, durable, reliable and swift which makes it an excellent candidate for a fast data architectural component [10]. However, it needs to be configured by the developer and this requires specific knowledge. Also, if the delivery semantics is a problem, this needs to be handled by the business code. Spark is easy to integrate with a Hadoop[2] application. This is advantageous because a very high number of big data applications use Hadoop today. However, the latency is high as it is not a truly real-time system and it has the same small file problem (small amounts of data in a large number of files is inefficient) as Hadoop. Kinesis can easily be integrated with an Amazon AWS application which provides easy administration and good scalability. However, this is not as fast and efficient as Kafka but better in those aspects than Spark. In addition, it can be expensive to use.

2.1.2 Stream Transformation

Moving on from ingestion of a stream to its transformation, Table 2.4 shows the difference between applications that provide stream transformation or processing.

Samza has no native support for batch streaming. Its competitors however can work in both modes: batch and stream. Flink and Samza are truly real-time systems. Spark rather works on micro-batches to simulate a real-time environment. Storm at its core is essentially a real-time system. Its variant called Trident however, enables micro-batching.

Consequently the former provides at-least-once delivery semantics (explained in stream ingestion), while the latter provides exactly-once semantics, similar to Flink and Spark. Therefore the choice depends on the idempotence of the application that needs streaming support. Spark Engine is expected to require more time to run as compared to others.

All of the contestants come with their own set of advantages and disadvantages as seen in Table 2.4. Ultimately, the choice depends on the target application. For example, Flink is fast and contains a self-optimization module, but as one of the latest frameworks, is still slightly immature in comparison. It is also known to have certain memory limitations, for its pipelined execution. Spark is one of the most widely used stream processing engines. It has a large support group and analysis tools in the market. However, in comparison to others, it does have a higher latency. Samza is a stateful stream processing engine, but has at least once semantics, and therefore, message deliveries could need management via programming. Storm provides good integration possibilities with YARN, but does not guarantee ordering of messages.

Apart from the commercial frameworks that provide stream processing, transformation engines can also be created with the application framework which is used for a project. It requires compatibility with streaming tools, which is generally found in higher level languages such as Java and Python.

[2]http://hadoop.apache.org

	Flink	Spark Engine	Samza	Storm
Stream and Batch support	Both	Both	Only Stream	Both
Streaming	True streaming	On micro batches	True streaming	Core: True streaming Trident: Micro batches
Guarantees	Exactly-once	Exactly-once	At least once	Core: At least once Trident: Exactly-once
Latency	Very Low	High	Low	Low
Throughput	High	High	Depends on Kafka	Low
Advantages	Less configuration, fast, own memory management, self optimization	Fast, versatile (stand alone / existing cluster), several useful libraries,	Loosely tied processing steps, stateful, easier high level extractions	Integration with YARN possible
Limitations	Comparatively immature, memory limitation (for pipelined execution)	expensive (memory), high latency, possible resource scarcity	Bad for use cases that require exactly once semantics	No ordering guarantee of messages

Table 2.4: Difference between commercial frameworks that enable stream processing

Thus, streams can and need to be transformed, in order to process the data contained in them. After the stream in transformed, the data is finally ready to be stored in a database, analyzed by an application or processed by another module.

2.2 The Gap: Big Data Processing vs. HPC

We have discussed about the importance of both HPC and Big Data. This section compares the mainstream organization of these technologies, based on a study by Anzt et al. [1].

A typical big data stack (left) uses commodity hardware and Ethernet networks to form a cluster. Big Data and Apache Hadoop[3] are often heard in the same sentence since Hadoop is often used to build big data applications. It consists of a distributed file system called HDFS, which usually lies on the base of the software pile. Hbase is a scalable, distributed database for large tables built on top of HDFS. MapReduce, a system that processes large amounts of data parallely, is the backbone of Hadoop-based systems. Analytical software (Pig, Hive, Mahout) works above and with this structure. Streaming of big data is possible with tools like Kafka[29], Storm etc.

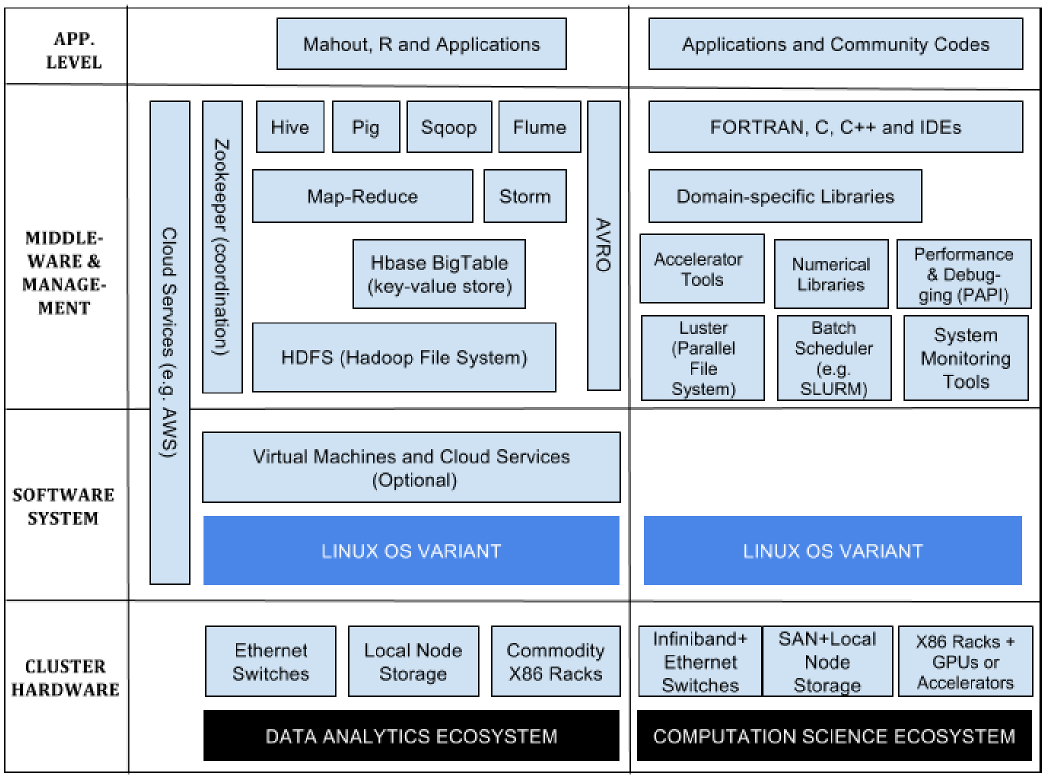
APP. LEVEL
MIDDLE-WARE & MANAGE-MENT
SOFTWARE SYSTEM
CLUSTER HARDWARE
Mahout, R and Applications
Cloud Services (e.g. AWS)
Zookeeper (coordination)
Hive
Pig
Sqoop
Flume
Map-Reduce
Storm
Hbase BigTable (key-value store)
HDFS (Hadoop File System)
AVRO
Virtual Machines and Cloud Services (Optional)
LINUX OS VARIANT
Ethernet Switches
Local Node Storage
Commodity X86 Racks
DATA ANALYTICS ECOSYSTEM
Applications and Community Codes
FORTRAN, C, C++ and IDEs
Domain-specific Libraries
Accelerator Tools
Numerical Libraries
Performance & Debug-ging (PAPI)
Luster (Parallel File System)
Batch Scheduler (e.g. SLURM)
System Monitoring Tools
LINUX OS VARIANT
Infiniband+ Ethernet Switches
SAN+Local Node Storage
X86 Racks + GPUs or Accelerators
COMPUTATION SCIENCE ECOSYSTEM

Figure 2.3: Examples of mainstream stacks of HPC and Big Data [1]

In contrast, the HPC stack is built on server-level processors and co-processors as shown in Figure 2.3. Parallel file systems like Lustre and batch schedulers like SLURM form the basis of operating libraries in a mainstream HPC stack, see Figure 2.3. Accelerator tools and libraries for debugging, numerical computations and performance as well as

[3]Apache Hadoop community, “Welcome to Apache^{TM} Hadoop®!”, http://hadoop.apache.org/

domain specific packages usually form the layer above it. For analysis, C, C++ and Fortran programs are written and are normally used with CUDA code and OpenCL. Thus, the picture of low level vs high level optimization areas becomes clearer after understanding Figure 2.3.

Both these technologies focus on different approaches to deal with data processing and handle situations in their own ways. Big Data Applications require memory and bandwidth, which is not the focus of modern architectures like GPUs. However, GPUs are helpful if the task is computation-intensive and the data volume it consumes is not too huge[30]. Therefore if an application chooses Big Data Frameworks, it might fail to exploit the possibilities of modern hardware, leading to underutilization and higher runtimes. On the other hand if an application chooses to use HPC, it might miss out on some of the interesting software features of Big Data processing frameworks like reliability, abstractions for parallel processing, ease of programming, ease of integration with other tools, etc.

We have explained the notion, the architecture and decision making factors for streaming systems. Now we will understand the concept of hardware acceleration and compare devices that provide it.

2.3 Hardware Acceleration

Traditionally, everyday computing systems run on a simple CPU. However, certain devices or hardware can enhance its performance in some aspects. This is hardware acceleration in the simplest of terms. It offloads or redirects certain tasks to this specialized device hereby increasing the computational speed, efficiency, image quality, or any of them combined.

Hardware acceleration has been evolving since the 1980s, also known as the Cray era. Vector supercomputing was one of the earliest forms of hardware acceleration[1]. It provided faster computation through implementing instructions on a one dimensional vector, as opposed to single data items on a scalar processor. The industry then moved on to Massively Parallel Processing (MPPs) and Shared Memory Multiprocessors (SMPs). As the name suggests, MPPs bring better performance by parallelizing computations and running them simultaneously. The parallelism is possible through having multiple processors. SMPs also proved to be efficient by using multiple processors, but each of these independent processors shared the same memory. In the 20th century, it was common to attach devices or cards to the processor. Using sound cards for better audio or graphics cards for enhancing image and/or video quality is also a form of hardware acceleration.

In today's world hardware acceleration can be achieved using devices or chips to enhance a system's performance. However the notion is quite old. It was proposed by D.J. DeWitt before 1980 which was especially designed for relational database management[31]. However with advances in technology, the world has taken it up a notch and we now have the following devices which provide a tremendous potential in processing speed:

2.3.1 FPGA: Field Programmable Gate Arrays

FPGAs are one of the earliest forms of accelerators which are nothing but configurable logic blocks (multiple logic cells and a switch box) [30]. Components in a typical FPGA can be observed on Figure 2.4[32].

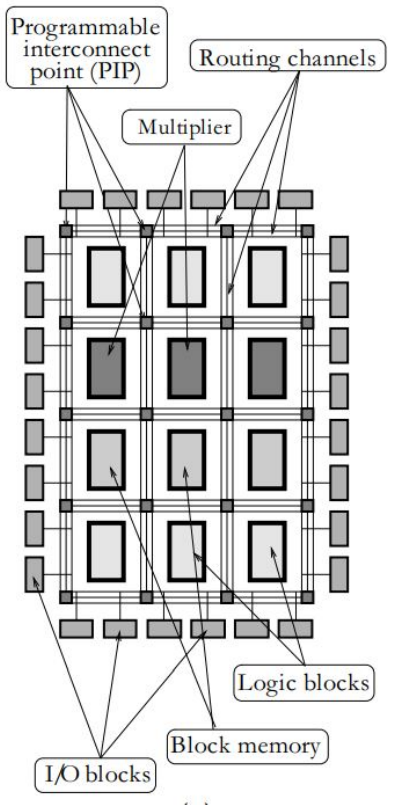

Figure 2.4: Most common FGPA architecture

The logic blocks (small bit tables) can be identified as the elementary unit of the device. A programmable I/O block with flip-flops is at its periphery. Some multiplier units are at its heart. All the units are connected through a routing line. FPGAs performed better than native floating point design in cores and showed tremendous potential for digital signal processing and control systems in the early 2000s. With advances today, FPGA based systems are used in sensor applications[33], graph computation[34], deep convolutional neural networks[35] and several others. Its features of high performance, on-chip RAMs, reconfigurability, fast development round, etc make them appropriate to use.

2.3.2 GPU: Graphics Processing Units

It is a well known fact that GPUs were originally used in gaming to boost video performance. They were a mere enhancement attached to video cards[36]. GPUs proved to be beneficial for computations more general than image rendering, especially database operations [37][38][39][40][41]. When GPUs are used for purposes other than graphical enhancement, it is referred to as GPGPU (General Purpose computing on Graphics Processing Unit). GPUs gained recognition as co-processors in a hybrid environment ever since Govindaraju et al.[42] came up with algorithms that make use of GPUs to run database operations faster. Figure 2.5[36] shows a basic system that uses GPU as a co-processor. In such environments GPUs are beneficial for operations which are similar and need to be performed several times [4].

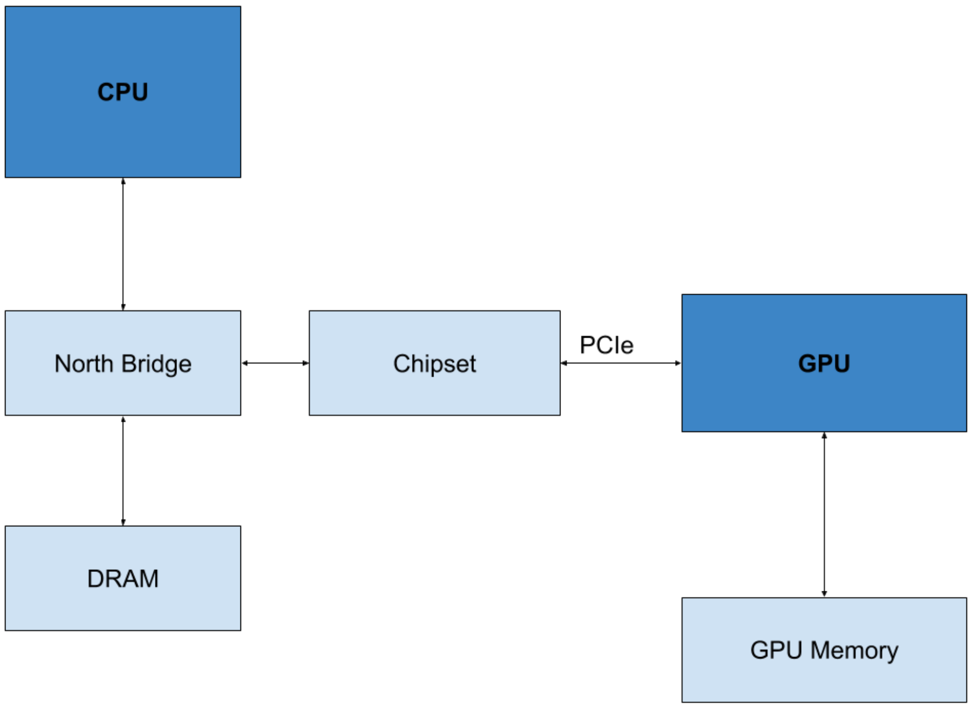

Figure 2.5: A Co-processing System with a GPU

A GPU is an accelerator. In other words, it is a co-processor used for enhancement purposes, but not the main processor. It is connected to the host through a bus, namely the PCIe (PCI: Peripheral Component Interconnect Express) bus. This is usually connected to the CPU through the chipset[5], which is connected to the Northbridge[6].

GPUs enable data parallelism, in a form that is orthogonal to the commonly used task parallelism of CPUs. Specifically, CPUs, by being composed of complex cores are able to schedule operations in parallel which are different to each other. This execution style is task parallelism, which can also be called Multiple-Instructions Multiple-Data (MIMD), given that different operations are performed in parallel over different data

[4]Kinetica, "Maximizing Data Analytics Price/Performance with GPU acceleration", https://www.kinetica.com/resources/

[5]An integrated circuit on the motherboard responsible for managing communication between the processor, memory and other devices

[6]The core of the chipset logic i.e. a connection between the processor and other devices

items. Unlike CPUs, GPUs are conformed of millions of very simple cores. As such, the cores are arranged to perform the same exact instructions in parallel, over different data items. This is data parallelism. It is a form of Single-Instruction Multiple-Data parallel processing (SIMD). GPUs can only perform SIMD processing. Applications considering the usage of GPUs need to adapt to these characteristics. Another example of SIMD processing is found in vectorized instructions in CPUs. CPUs and GPUs both have their own memories. The GPU memory, however, is smaller in comparison.

2.3.3 APU: Accelerated Processing Units

APUs, formerly known as Fusion, fused CPUs and GPUs on a single chip. APUs can be programmed using technologies like DirectCompute and OpenCL for creating thread level parallelism[43]. Their major objectives include enhancement and improvement of multimedia and vector-processing applications, reduction in energy consumption, in addition to better visual (2D as well as 3D) graphics. Figure 2.6[43] explains the basic units that make up an APU.

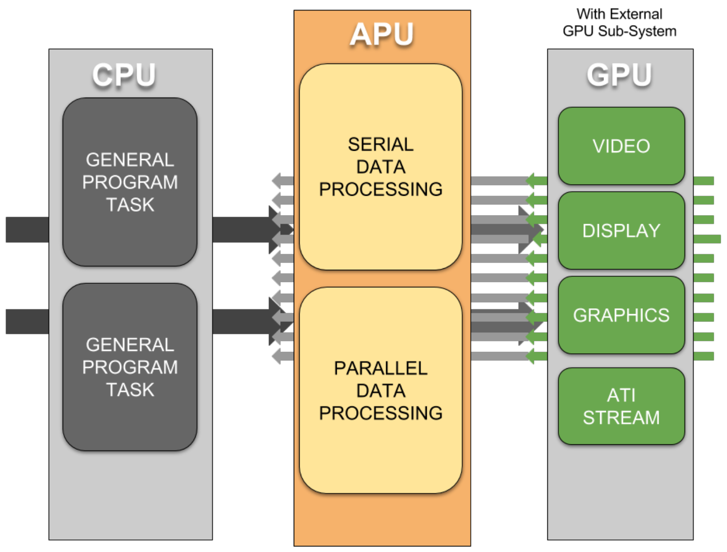

Figure 2.6: APU's arrangement

APUs are a systematic collaboration of multi-core CPUs for performing general program tasks and GPUs for parallel computing as well as advanced video performance. The APU resides with units for serial as well as parallel data processing, thereby forming a heterogeneous environment.

This architecture reduces the memory limitation of GPUs since the main memory is shared on the die, but separated for each core type[30]. All of these components are on the same silicon die and connected through a high speed bus. Thus it handles scalar workloads through x86 cores and vector workloads through GPUs.

2.3.4 MIC: Many Integrated Core

Intel's MIC was based on three projects namely Larrabee, 80-core Tera-scale research processor, and the Single-chip Cloud Computer processor known as SCC [30]. As opposed to current GPU trends, MIC aimed for highly parallel general-purpose computing through the shared memory execution model[44]. It comprises many processing cores that enables scalability, vector units that boost computational performance and multi-threading to use resources well. It supports languages like C, C++, Fortran with OpenCL, MPI and other wrappers on the higher level and also processor-level instructions.

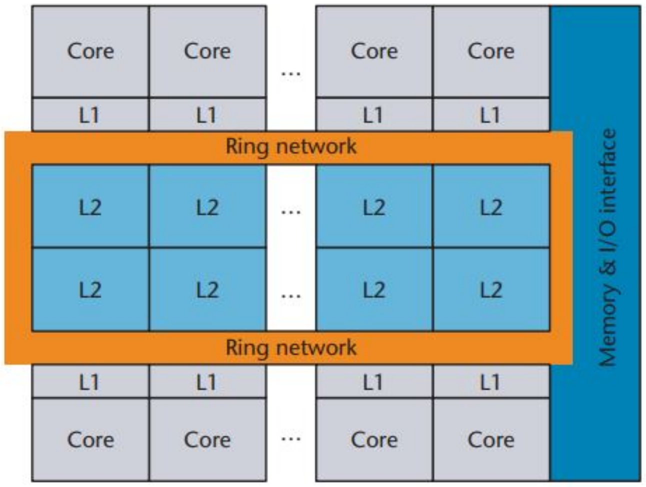

Figure 2.7: Architecture of a MIC

The Figure 2.7[45] shows MIC's micro-architecture. It is developed as an architecture called Knights Ferry with 32 general purpose cores, a 32 KB L1 cache and 256 KB L2 cache, which plugs into a PCI Express slot. The bidirectional ring interconnect is in contact with cores, caches and memory . Each core provides high throughput for parallel operations. It finds applications in medical imaging, weather prediction, computer-aided design, trading etc.

2.3.5 Summary Of Hardware Acceleration

The literature review for the mentioned devices brought forward some comparable features. Table 2.5 briefly summarizes the forms of hardware acceleration that were described in the previous sections.

The hardware accelerators vary widely, from their concepts and architecture to their pros and cons. FPGAs are basically programmable digital chips (gate arrays). GPUs on the other hand, specifically GPGPUs, are co-processors along with the CPUs that perform compute intensive tasks in parallel. Data is passed via buses between the CPU and GPU. CPUs can perform SIMD or MIMD processing, whereas GPUs are limited to SIMD. APUs encompass both CPUs and GPUs on the same die, thereby reducing the communication overheads. Whereas MICs are a group of several chips of the same family to enable parallel programming.

We see that although FPGAs are the oldest form of hardware acceleration, they are still available in the market, although not utilized on a large scale. GPUs are modern and widely used. However their small on-chip memory brings some limitations, which can be handled in a programmatic manner. APUs are efficient but not as prominent in the market as GPUs, and have a margin for GPU starvation. MICs are also fast processors, more flexible in terms of programming but have a high power consumption.

	FPGA	**GPU**	**APU**	**MIC**
Acronym for	Field Programmable Gate Array	Graphics Processing Unit	Accelerated Processing Unit	Many integrated core .
Market dominators	Xilinx	Nvidia	AMD	Intel
What it is	Configurable logic blocks	A chip that performs compute-intensive tasks better than CPUs	An integrated chip with CPUs and GPUs for enhancing both scalar and vector workloads .	A set of multicore chips with highly parallel capabilities
Advantages	Performs better than native floating point design	1. Very fast computations of mathematically intensive calculations 2. Hence, outstanding performance of vector workloads and thereby image/video processing .	Efficient general task performance(x86 cores), and also better compute-intensive calculations(GPU)	Offers much more flexibility than GPU (CUDA/OpenCL) programming and a simplified handling for programmers
Disadvantages	Not explored to its full potential	Small on-chip memory	Possible GPU starvation	Higher power consumption .
Examples	Xilinx XC2064, Stratix 5	GeForce, Nvidia Tesla, Radeon RX, AMD FireStream etc	Llano, Kaveri,Raven Ridge etc	Xeon Phi

Table 2.5: Summary of devices used for hardware acceleration

FPGAs are produced by Xilinx, Altera and Actel. Nvidia, Radeon and AMD are some of the top manufacturers of GPUs for computation as well as gaming. AMD however is a bigger producer of APUs and MICs are owned by Intel. Xeon Phi was Intel's widespread success of MICs.

Apart from technicalities, it is also interesting to compare these technologies by their popularity. To achieve this Figure 2.8 gives an overview of Google Trends[7] for the above mentioned topics as of January 2018. It clearly shows that GPUs are currently the most popular choice. It is followed by both APUs and FPGAs which occasionally interchange their ranks for the second position.

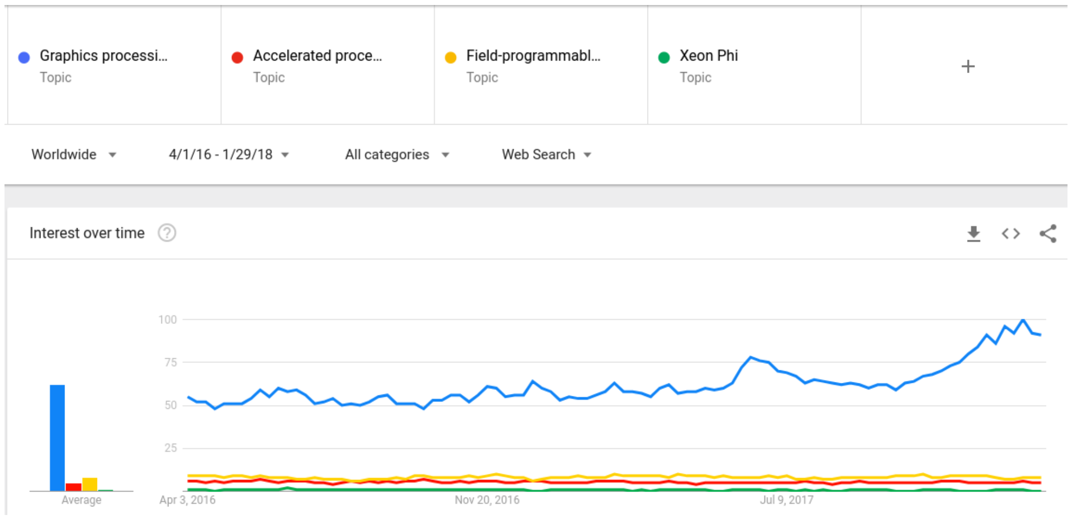

Figure 2.8: Google Trends of hardware acceleration devices

A comparison was also made for the candidates with respect to their search terms and examples of these topics. In every case, GPUs were a clearly eminent. Assuming that the prevalence of GPUs would benefit us from a user-support perspective, we chose GPUs for hardware acceleration in our use-case. Therefore we dive deeper into GPUs and explain them in the next section.

2.4 GPU / GPGPU

The history of GPUs and uses of GPGPUs have been briefly discussed in Section 2.3.2. This section discusses GPGPUs in more detail. The reason that GPUs are fast with arithmetic calculations, is the high bandwidth of their floating-point units and hardware multithreading[36]. This is explored with SIMT (Single Instruction Multiple Threads) or SIMD (Single Instruction Multiple Data) execution techniques. In other words, a single instruction is fetched and decoded, but executed on several data operands. The degree of hardware multithreading is much higher in GPUs than CPUs, given their larger register size. By means of multithreading, GPUs are capable of processing multiple objects simultaneously.

SIMD instruction units and multithreading units can be found on GPU components called SP (Stream Processors). They are visualized in Figure 2.9[46]. Multiple SPs

[7]Google Trends results for reference: https://trends.google.com/trends/explore?date=2016-04-01%202018-01-29&q=%2Fm%2F02211f,%2Fm%2F0g5q45d,%2Fm%2F02ytr,%2Fm%2F0c3yqx_

make up an SM (Streaming Multiprocessor). The GPU shown has a total of 128 SPs, organized into 16 SMs in the form of 8 TPCs (Texture/Processor Clusters) which enable parallel processing of an instruction on multiple data. A TPC contains a geometry controller[8] and SMs.

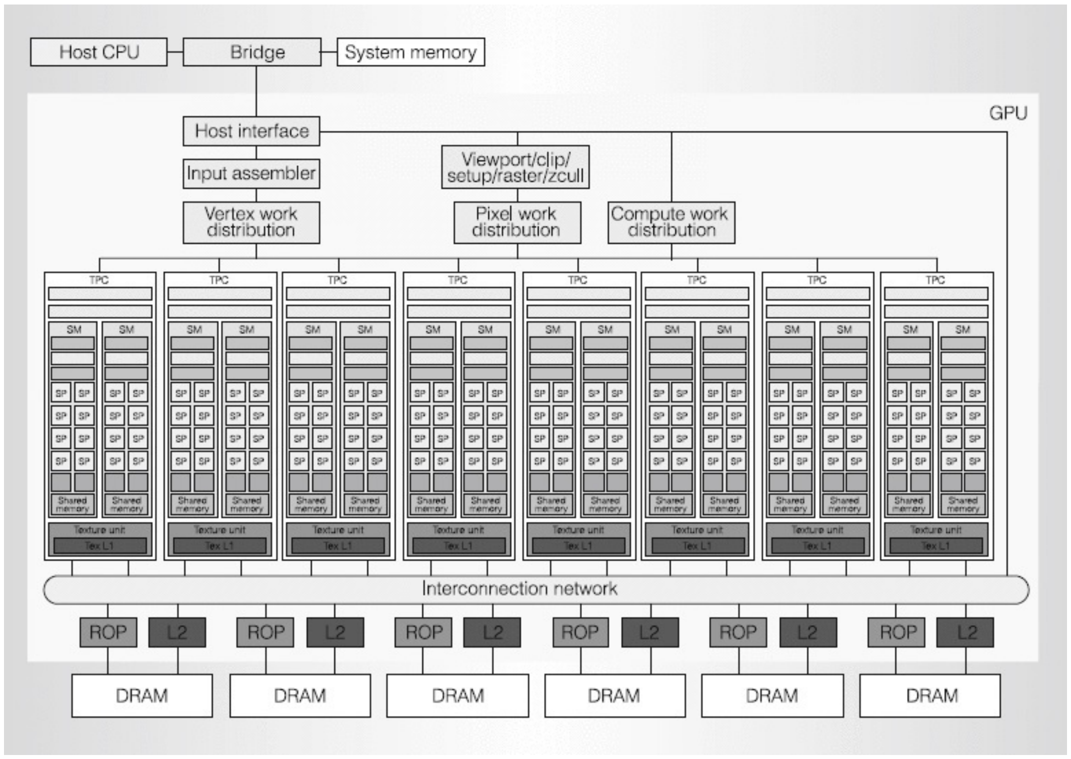

Figure 2.9: Block diagram of Nvidia's Tesla (as GeForce 8800 GPU)

Primarily, the processing is done in TPCs, with SPs being the thread processors, performing operations like add, multiply, multiply-add and other comparison and conversion operations. GPUs have a scalable memory system in the form of external memory i.e. DRAM and ROPs (Raster Operation Processors)[9]. The values for processing and read requests are communicated using the communication network. The texture units (TEX) are responsible for texture mapping[10]. In GPGPUs, the maps store data required for processing.

GPGPUs are mainly programmed using C or C++ extensions which will be explained in Section 2.4.1.

[8]An image, on the screen space, needs to be managed, starting from its vertices. This is performed by a geometry controller and then handed over to SMs.

[9]ROPs handle a GPU's colour and geometry rendering

[10]Mapping detailed information about colour, texture and other information to a computer-rendered image

2.4.1 Execution Model

C or C++ extensions like CUDA[47] (Compute Unified Device Architecture) and OpenCL[48] (Open Computing Language) are the two trending languages for GPGPU programming. For GPGPU programming, with GPU as a co-processor, the host code runs on the host processor (CPU), and kernels run on the co-processor (GPU). Both CUDA and OpenCL revolve around this paradigm [49][36], with minor differences in their definitions of a kernel.

2.4.1.0.1 CUDA's Execution Model

CUDA provides a C/C++ interface to program GPUs [47]. The program function is written as a 'kernel', which is executed when commanded to. The execution is performed by smaller individual components called threads, which enable parallel processing. Threads gain their identity by up-to a 3-dimensional vector, forming a block of threads: thread block. Using the indices of these blocks, one can program threads to execute the kernel. Thread blocks can also be grouped further into one, two or three-dimensional 'grids'. The execution model of CUDA can be summarized as follows [47][36]:

- Kernel:
  The piece of code that runs on the GPU.
- Threads:
  A component that executes the kernel function on a part of data.
- Thread block:
  A group of threads that share a piece of memory and are expected to reside on the same processor core. As of 2017, an average GPUs consists of 1024 threads in a block[11].
- Grid:
  One-dimensional, two-dimensional, or three-dimensional collection of thread blocks. The size of the data being processed or the number of processors play an important role in determining how many blocks make up a grid. Grids have the scope of global memory.

OpenCL is an alternative to programming GPUs. We will now discuss it.

2.4.1.0.2 OpenCL's Execution Model

OpenCL operates on work-items which are a result of kernels created on the CPU[48]. A command triggers the creation of an index space, on each of which an instance of kernel

[11]http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

executes. The co-ordinate position of this instance in the index space identifies it (global ID), and it is termed as a work-item. Thus, a kernel defines and issues a a sequence of instructions. These instructions are received by multiple work-items who process the same sequence i.e., parallel processing. Multiple work-items make a work-group, and all items execute concurrently within. The execution model of OpenCL can be summarized as follows:

- Devices:
  OpenCL compatible co-processors that are installed along with the host
- Kernel:
  The piece of code that runs on a device. It is categorized as OpenCL kernels (written and compiled with the C programming language wrapper) and native kernels (pre-written and accessed within the programmer's code).
- Work-items:
  Components that perform instructions, on an atomic level.
- Work-group:
  A logical collection of work-items that have the scope of the global index space, within which every work-item concurrently processes the same set of instructions.
- Program objects:
  Executable code that forms the kernel.
- Memory objects:
  Analogous to programming variables, objects that contain values and are used while processing by the devices.

We have thus discussed CUDA and openCL: some of the languages that enable GPU programming. We will now move on to the general memory model of GPUs, namely, the memory hierarchy and the significance of each type of memory.

2.4.2 Memory Model

The memory model to program GPUs consists of 3 major divisions: local, shared and global [36][46]. The memory hierarchy, in general, is explained in the table below:

The scope of local memory is limited to a thread, shared memory to a thread-block and global memory to a kernel. Section 2.4.1 discusses threads, blocks and kernels in a detailed manner. The local and global memory reside in the external DRAM whereas the shared memory is programmed in the SM. Memory instructions like load-local, store-local, load-shared, store-shared, load-global, store-global are implemented to pass data to and from the memory. Local and global memory have an access latency of 100 cycles, but shared memory is faster with a latency of only 4-32 cycles.

Apart from these, GPUs also have constant (small graphics programming variables) and texture memory (2D and 3D texture data) spaces, however they are more relevant to graphics rendering[36].

	Local	**Shared**	**Global**
Scope	Thread	Thread Block	Kernel
Position in GPU	External DRAM	Shared multiport	External DRAM
Memory instructions	load-local, store-local	load-shared, store-shared	load-global, store-global
Access latency (approx.)	100 cycles	4 - 32 cycles	100 cycles

Table 2.7: Memory model in GPGPUs

2.5 Supporting Components Of A Fast Data Framework

The following sections present three supporting components which we identify as part of end-to-end fast data frameworks. This list is not comprehensive, it merely intends to introduce the basic features of these components, in consideration that they were necessary to adopt in our case study. After providing general features about these components we discuss in more detail the specific instances of the components which we selected for our study (Apache Kafka, Cassandra and TensorFlow).

2.5.1 Large-Scale Processing Frameworks

Large-scale processing frameworks are the foremost component of a Big Data technology stack. Their central role is to provide parallel and distributed computation with a set of simple abstractions, such that developers are shielded from the complexity of handling explicit inter-node communication, load-balancing, fault-tolerance, scheduling decisions and other low-level operational matters.

Similar to query engines in relational databases, the fundamental elements of these frameworks are:

- Primitives for data distribution and placement.
- Support for resource elasticity, such that compute nodes can be added and removed.
- A dataflow description of queries (i.e., a graph where each node represents an operation that transforms data, and edges represent transfers).

One paradigm for organizing the computation described by a dataflow graph could be called *Single Process Multiple Data*. In this approach, jobs are scheduled to perform the same process over different data items. Map Reduce (explained later in Section 2.5.1), one example of batch processing, uses this paradigm.

Other paradigm has been, *Bulk synchronous processing (BSP)* [50] which was proposed as a more general parallel processing model than Map Reduce. In this model, computation proceeds in iterations (like Map Reduce), after each iteration the models needs to synchronize before moving to a next iteration. Different from Map Reduce, here the mappers can keep alive and carry on their results to a next iteration. Graph processing frameworks, like Google Pregel and Apache Giraph follow this approach, which is beneficial for graph large-scale graph traversals. Similarly, some dataflow-based systems like Pig and Apache Hive, have found BSP to be useful for supporting SQL workloads. Spark too originated as a BSP system which is distinguished by having a core in-memory design. It has evolved to become a streaming framework.

Finally, as described before, streaming systems, constitute a third paradigm. Early systems implementing stream processing, like IBM System S and Borealis developed core ideas for fault-tolerance in these frameworks. However they have been criticized for their lack of support for elasticity [51]. Later tools like those discussed in Section 2.1 overcome this limitation.

In a streaming environment, the large scale processing framework consists of components responsible for stream ingestion as well as processing (Section 2.1). When using a streaming system, the consumers of the messages can be scheduled and placed at the application level, giving users control over these aspects, and possibly easing the evaluation process.

From components of this type, we discuss with more detail, some examples in the next sections.

Spark Streaming and Spark Engine

With the advent of Hadoop, big data ingestion and processing became widespread. The map reduce model splits or “maps” the large amount data to different “reducers” and performs the same operation on them in parallel. It then aggregates their outputs to form a single final result. It is a good fit for applications requiring a single-pass, batch processing model[52]. However, the need for more complex multi-pass algorithms and for interactive ad-hoc queries has lead to the development of frameworks supporting bulk synchronous parallel models and streaming, like Apache Spark.

As a distinguishing factor, Spark stores data in Resilient Distributed Datasets (RDDs) and provides good fault tolerance using this. Data is stored in memory, thereby accelerating the process and making queries faster than other distributed file systems. Spark provides a Java/Scala API for programming the application.

A large scale distributed streaming system could use Spark streaming for stream ingestion, and the Spark engine for stream processing. In comparison to its competitors it has some limitations as follows[53]:

- A higher latency due to key-value in-memory mapping for big data.
- The lack of event time processing concepts, such as windowed functions, could make stream ingestion problematic.
- Maintaining and tuning it could be considered cumbersome.

However, this is a good option for existing Hadoop users. It provides a big collection of machine learning libraries, is compatible with other streaming frameworks such as Kinesis and Kafka and provides the possibility to create batches (for batched programming) or micro batches (for streaming).

Kinesis and Spark Engine

Another alternative would be to use Kinesis for stream ingestion and to process it using the Spark engine. Kinesis, specifically Kinesis Streams, is a distributed message passing system for large scale big data applications [54]. It uses a publish subscribe paradigm, which means that the sender sends data, but not directly to a receiver [55]. Any receiver that is subscribed to a class of data will then receive the packets that have been sent. The unit of data that is sent and received is called a "Message". A streaming application can periodically send event messages to Kinesis Streams [54].

Data streamed by it could then be connected to a Spark application which could process it in a distributed manner. Alternatively, for a purely Amazon suite, one could also opt for Kinesis Firehose for smaller processing such as data loading into AWS (Amazon Web Services).

Kinesis with Spark guarantees near real time processing, because the event absorption of Kinesis is very fast. The data can be sent to Spark in batches of desired sizes, and one can then benefit from the libraries and support of the large Spark community. Amazon products such as Kinesis and Firehose have a built-in metric system capable of monitoring CPU usage, memory consumption, rate of incoming requests and load on the system. However, with Kinesis such an architecture could get expensive and might not be flexible enough for all application needs [54].

Kafka

Another approach could use Kafka for stream processing.

Apache Kafka is a distributed streaming platform which was originally created for log passing systems [29], in LinkedIn, and then taken over by the Apache Foundation. Kafka is a publish-subscribe system [55], much like Kinesis Streams. The architecture of Kafka is shown in Figure 2.10.

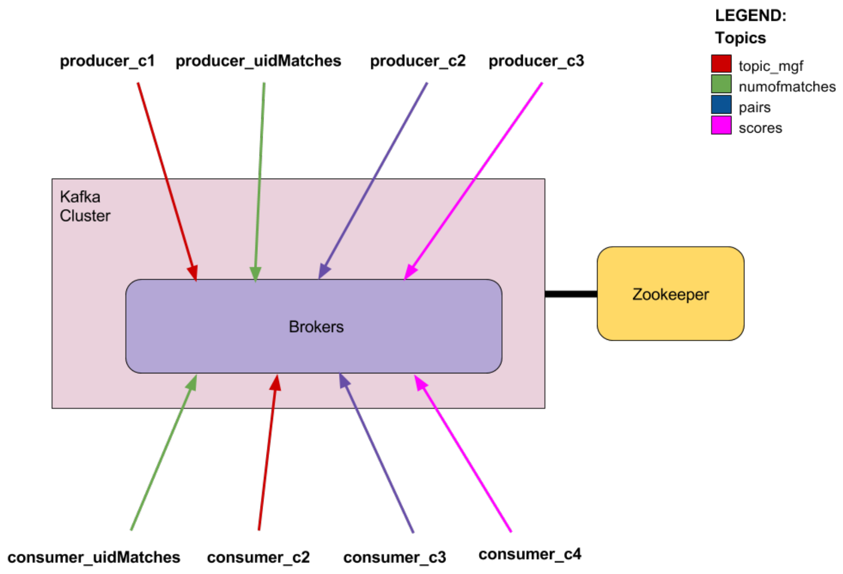

Figure 2.10: Working principle of Kafka

Kafka works as follows [29]: A “Topic” defines a stream of messages of a particular type. These messages arrive from a “Producer”. The published messages are stored in a “Broker”, which are a set of servers. Messages stored in these servers can then be used by “Consumers” once they subscribe to the brokers. Topics can further be broken down into “Partitions” if required [55]. This allows optimizations in a network as a topic can then horizontally scale and perform better. As a stream processing system, data in each topic moves in a single stream towards the consumers.

Producers are generally not very system-aware. They are not aware of which consumer or which partition the messages are delivered to. Although this is configurable, it is not required. Producers simply send messages to a broker, and the system evenly distributes them to the available partitions. Internally the producer sends serialized messages.

Messages can be identified by keys (optional) and values, and they serve as metadata for partition management. They can be asynchronous (producer waits for successful delivery before sending the next) or asynchronous.

Consumers must subscribe to a topic to receive messages and deserialize them, in the order in which they were produced. As Kafka has an at-least-once delivery semantics, every message is received by the consumer once or more. Duplicate messages are handled in our system using the Application logic. When a Kafka consumer reads a message from the topic, it increases the offset (positional metadata: an integer count) to know what to read next. Consumer groups divide the data in partitions to be read. Each consumer within the same consumer group reads one or more partitions, and this mapping is termed as “ownership”.

A Kafka broker is nothing but a Kafka server. It receives messages from the producers and assigns offsets to the messages and then fetches the required messages for the consumers. Apart from this, administrative tasks such as failure monitoring, assignment of messages and partitions are also the responsibility of the broker.

These messages can then be passed to an application framework written in any supported language (e.g. Python or Java), thereby allowing it access to the event data.

2.5.2 Distributed And Scalable Databases

Fast data frameworks rapidly generate and process data continuously. Therefore, they require a supporting infrastructure for data management. Distributed File Systems, like HDFS or the Google File System, next to in-memory grids (which are differentiated since files do not overflow from memory to disk), are one alternative approach. Through this users can define file formats and use them with Hadoop. However, to work with these, there is usually not a natively-supported specialized language for queries over the data and users have limited control over data placement.

Scalable distributed databases could be a better solution. A big or fast data framework could potentially be a large distributed framework. Hence scalability is a property worth having in such distributed databases. In general these databases can be relational (with examples like Google Spanner) or non-relational (being, for example, key value stores, key column value stores, or document stores).

Other database engines that have grown to incorporate scalable functionality are search engines (such as Elasticsearch and Solr), graph databases (such as Neo4j and Janus-Graph), RDF stores (such as Virtuoso) and time-series databases (such as InfluxDB), however they incorporate models for applications more specialized than the one being considered for our case study.

In the next section we discuss examples of scalable distributed database systems, under the key column value, key value and document models.

We do not discuss largely scalable databases with relational models, because Cockroach DB, the publicly available version of Google Cloud Spanner, which is the foremost example of an open source database in this group, is still in a relatively young phase of its development, and thus we did not consider it's use for this Thesis.

The motivation of this section is to provide insights into possible technologies that could support the case study selected.

Cassandra

Cassandra is a non-relational open-source distributed database, originally designed by Facebook for peer-to-peer connected nodes [56]. Unlike relational databases, Cassandra is not truly ACID (Atomicity, Consistency, Isolation, Durability) [57], but BASE (Basically Available Soft-state Eventual consistency) [58] compliant. CAP/Brewer's theorem states that it is impossible for a large scale distributed database system to have consistency, availability and partition tolerance all at once, and we need to compromise on at least one. Cassandra compromises on consistency and is eventually consistent, in order to provide good availability and partition tolerance [2]. Cassandra supports the Cassandra Query Language (CQL) which can be used to query Cassandra in an SQL-like fashion.

A Cassandra table is basically a multidimensional indexed map which is distributed across a cluster in nodes. It is optimized for storing large tables with flexible schema and sparseness. Cassandra can be called a key-column value store, since access to items requires row ids (i.e., keys) and then the key of columns [2]. Cassandra groups columns into simple (a group of columns, as shown in Figure 2.11) or super (a column family within a column family) column families [56]. Both keys and column values are used for sharding data, making the system more scalable. Moreover, Cassandra has a container called a "Keyspace" which can contain multiple tables which share the same characteristics.

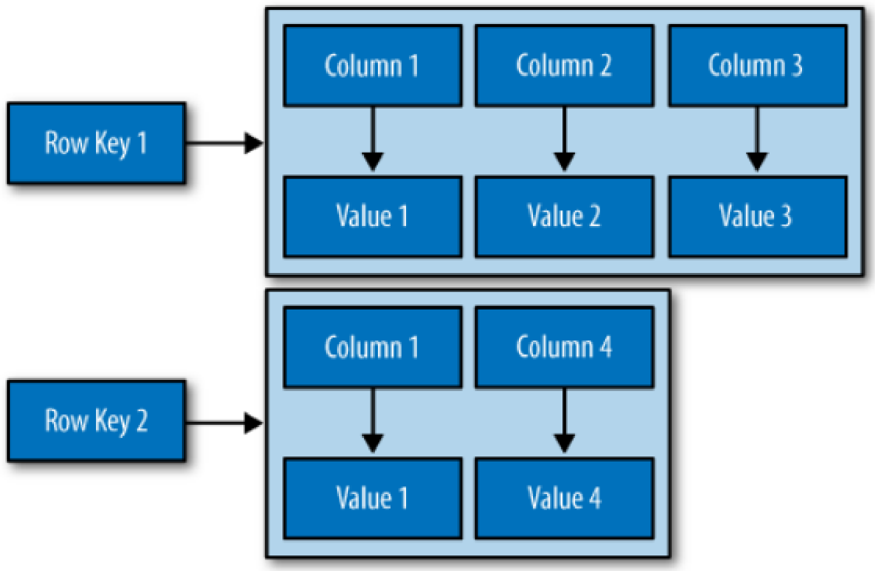

Figure 2.11: A column family in Cassandra [2]

Cassandra is capable of handling large amounts of distributed data, provides good availability since there is no single point of failure and provides high scalability. Availability and durability is achieved by replication of shards and consensus algorithms. A maximum replication factor is the number of nodes in the cluster. Scalability is made possible by dynamic partitioning across the cluster using consistent hashing [59].

In terms of it's features and data model, Cassandra is similar to other key-column value systems like Google's BigTable, Apache H-Base (the open source implementation of BigTable [60]) and Amazon's Dynamo.

Each node in Cassandra is aware of its responsibilities since it knows every other node in the cluster. Cassandra uses the Gossip protocol [61] for node communication. This algorithm deploys a three-phase acknowledgement message transfer scheme for communication that happens once every second in theory.

Thus, Cassandra is a useful, peer-to-peer distributed and scalable database.

Redis

Key-value stores provide a data model that is much simpler than key-column value stores. The core data abstraction is a pair of key, value items. The structure of the value items is transparent to the database and the user has complete control on how to map a structure item into the storage.

Given their simplicity, key-value stores are optimized for speedy data access and for memory management. Some applications could include the tracking of environmental variables in distributed environments of the caching of data between a database and an application.

Redis is an extensible key value store, named as a *REmote DIctionary Server*[60]. Data values in Redis can be simple like hashes, lists, or more complex like sorted sets, sets, bitmaps and hyperloglog sketches. Studies have found that hashes are very efficient for storing keys in Redis [12]. Add-on modules can extend the data types for values and the functionality of Redis. For example, through add-on modules Redis can support a search-engine (ReSearch), a graph database (Redis-Graph), a document store (ReJSON) and others. This possibility of extending core feature by adding modules is what makes Redis an extensible storage system.

Keys in Redis can be simple like strings or hashes [62].

Redis was written originally in C, and then ported to Lua, and has clients available in more than 30 languages.

Redis can be executed as a server via the redis-server command, and an easy access can be accomplished via the redis-cli. Queries in Redis are supported by a very simple query interface for basic operations such as GET key, SET key value, KEYS (to see all keys), MGET key1 key2... (to pass multiple keys), SETEX key value ttl (for setting an expire time for key storage), among others.

Redis can run either in-memory or with persistence. It can also act as a cache. Partitioning and data distribution is achieved by RedisCluster, with resource elasticity achieved by Redis Sentinel.

In terms of the CAP theorem, Redis claims to favor Consistency and Partition-tolerance over Availability.

Other key value stores are Voldemort and Memcached.

MongoDB

Document stores are also another example of NoSQL system. They are concerned with storing structured data with each item having a flexible schema, and nested relationships.

[12]http://instagram-engineering.tumblr.com/post/12202313862/storing-hundreds-of-millions-of-simple-key-value

These systems are better when, per design, the applications using it require items with more internal relationships than relationships between items [63].

XML and JSON are examples of popular document formats, each with corresponding navigation tools.

MongoDB is a distributed document store. Data in MongoDB is stored using BSON, a binary JSON format. Groups of documents are stored in collections. This grouping does not have any implication on the schema of the internal documents.

Given the centrality of documents for MongoDB, aggregations are a feature that is efficiently supported with a specialized aggregation framework.

Other characteristics of MongoDB include it's lack of support for memory mapped storaged, with the WiredTiger storage engine adopting a management of memory that resembles buffer management in traditional databases.

Regarding the CAP theorem MongoDB is expected to favor CP over A, in consideration that it is, by default, a single-master system.

Other examples of document stores are CouchDB and RethinkDB.

2.5.3 Processing Libraries and Programming Languages for Hardware-Optimized Operators

In order to utilize modern hardware, it is often necessary to adapt the code of applications such that they can exploit more parallelism, use better memory or adapt to processing models of novel devices. In general there are three possible general approaches for this adaptation:

- *Compiler support*, through which the code is automatically mapped to use modern hardware, at the cost perhaps of missing optimization opportunities.
- *Re-implementation in a specialized language*, through which programmers need to explicitly write their codes into parallel programming languages or libraries. This can be very efficient, but it can require significant effort from programmers.
- *Exploitation of hardware-optimized libraries*, in this case programmers adapt their code by explicitly using functionalities from libraries that utilize parallelism and hardware-tuned features. For example, this could be hardware-tuned data structures. Most hardware-accelerated libraries deal with specialized domains where algorithms and basic operations are standardized, for example Intel Nervana (for Deep Learning) or NVIDIA CUBLAS (for linear algebra). The benefit from this approach is that programmers need less effort than for re-implementation. The main problem is that the results might be less efficient than hand-tuned code also, libraries might introduce unforeseen limitations. Finally another limitation is that library implementations might not be generic enough to be applied for ad-hoc cases.

In the next small parts we introduce Tensorflow PyOpenCL as examples of a general library and a re-implementation approach, respectively. We do not discuss further examples, since a study of them is beyond the scope of our work.

Tensorflow

Tensorflow [64] (TF) is a dataflow system created to process large amounts of data in heterogeneous environments. It is designed to work efficiently with CPUs[13], GPUs[14] and even TPUs[15] [65] by mapping nodes of a dataflow graph[16] across one or multiple cores of devices. Vertices in TF represent computations and edges carry tensors, which are multidimensional arrays or vectors capable of representing 3D or higher dimensional objects. TF also integrates with hardware-tuned libraries, such as CUDNN and CUBLAS, thus it offers a set of accelerated operators for linear algebra and other operations [64]. These operators are carefully tuned for making efficient use of CPUs and GPUs.

TF has found a large application in the development of neural networks and training models (which are built over a core of matrices representing the edge relations connecting perceptrons through layers). A dataflow diagram is constructed to depict all computations and sub-computations, thereby forming a clear picture of the algorithm and opportunities of parallel computing. TF implements different types of queues, such as FIFO (First-In-First-Out) and operations on them such as Enqueue (insert) and Dequeue (pop). One may choose a queue best fitted to the algorithm. Such a construction of vertices, edges and queues allows for a sophisticated design of streaming computation between graphs and subgraphs.

TF has certain limitations in its role as a large-scale processing framework. Mainly that GPU access can only be done by one process at a time.

TF can be employed as a processing library that presents simple abstractions to developers, facilitating the use of specialized hardware without requiring the users to be concerned with low-level details.

To our knowledge there are few libraries available that combine hardware-accelerated functionality, with enough generality to support ad-hoc queries, as TF does.

PyOpenCL

Apart from Tensorflow, which offers high-level operators with a low-level GPU-accelerated implementation, users can implement their own GPU accelerated operators. One choice is to use OpenCL or CUDA, with a corresponding language-driver in the programming languages of choices.

[13]Central Processing Unit

[14]Graphics Processing Unit

[15]Tensor Processing Unit

[16]A directed graph with functions as nodes and their dependencies as connections between the inputs and outputs of these functions

Pyopencl[17] is a Python package, which provides an OpenCL API or driver, for Python [66]. Similarly, PyCuda allows users access to CUDA, using a Python library. They have a low level interface translation; it contains an object oriented shell on the lowest interface level. This allows all OpenCL/CUDA features to be scripting-compatible. They create runtime code on the GPU by passing GPU binaries using Cuda and OpenCL source code in string format.

2.6 Concluding Remarks

In order to study GPU-accelerated streaming for a real-world scientific application, we selected the case study of protein identification. In the preceeding sections we have covered the software and hardware basics, now we move towards the scientific application: its concepts, definitions and general background.

In this chapter we provided a comprehensive technical background for our work. We introduced key concepts of stream and batch processing, the two fundamental approaches for large scale data processing. We also presented and compared mainstream frameworks for these approaches. Next we established some distinctive characteristics of big data vs. HPC frameworks. This discussion was essential to our work, since in our research we study how to combine the best of both approaches. Furthermore we carefully described the main hardware accelerators: GPUs, FPGAs, MICs, and APUs. To conclude we established the most important features for GPGPU, which is the target hardware acceleration for our study: programming frameworks, execution and memory models.

In Chapter 7 we complement our technical background by covering related work in HPC and big data. In the next chapter we give comprehensive background on the case study in protein identification which we tackle for hardware acceleration in this thesis.

[17]https://mathema.tician.de/software/pyopencl/

3. CASE STUDY: PROTEIN IDENTIFIFCATION

This chapter explains in detail, the case study researched to support this thesis.

- Section 3.1.1 explains the biological background required to understand certain terms and concepts of proteins. The process of obtaining meaningful, calculable data from biological specimens, is explained in Section 3.1.2. Section 3.1.3 describes the experimental procedure, or, the setup for processing of proteins. The section provides a better understanding and appreciation, of the input data.

  Section 3.2 enumerates the available protein databases and their sources. This is followed by Section 3.3, which describes the algorithms available to carry out the task of protein identification.

- The algorithm used to identify proteins, the one BAXdem is greatly influenced by, is described in detail in Section 3.4.

3.1 Biological Background

This section will describe the required biological concepts, that helps understand the case study better.

3.1.1 Terms And Concepts In Proteomics

In this section we introduce the field of proteomics, providing further details on our specific scenario, which uses mass spectrometers for starting the protein identification process. Next we outline some relevant characteristics of the input data to the process (i.e., experimental and theoretical spectra) and we discuss protein databases and search engines, which are a common way of integrating theoretical spectra into the protein identification process.

We select this case study since it represents a real world challenge, with involves the use of large amounts of data and where the combined optimizations of hardware acceleration and large scale processing could have a positive impact. It fits the research question as it can be used in real-time in the future.

Proteomics refers to the study of proteomes [67]. A proteome is the entirety of protein contents of a given biological sample. Proteomics investigates the proteins and their functions in one species [21]. This thesis only deals with the identification of proteins, which falls under the umbrella of proteomics.

Proteins are macromolecules facilitating most of the functions in living organisms. Proteins are found in all living organisms [68]. They are responsible for activities such as DNA replication, transporting oxygen to the organs, providing energy and several others in organisms that they reside. Proteins are polymers of amino acids. A polymer is a molecule that consists of small building blocks (monomers), which are chained together. The polymer sequence defines the type and functions of proteins.

Amino Acids are the building blocks of proteins[68]. They contain carbon, hydrogen, nitrogen and oxygen. There are 20 amino acids commonly found in nature. Proteins are arranged to form large and complex molecules [68]. They are identified by a distinct order and set of amino acids. This arrangement can be perceived in four levels.

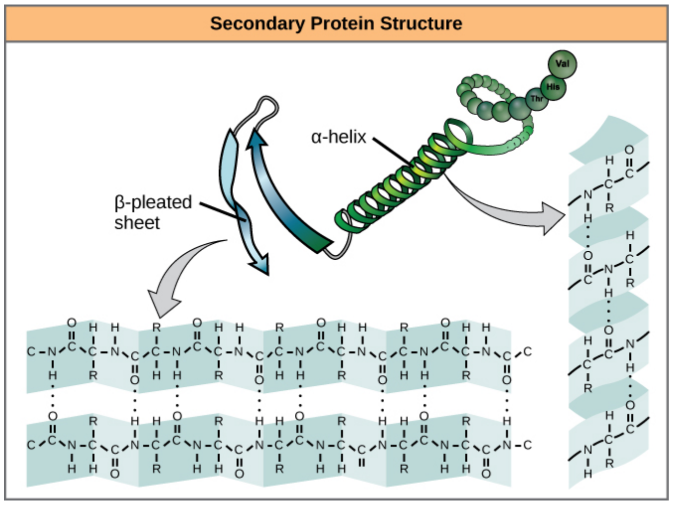

Figure 3.1: Secondary structure of proteins [a]

[a]OpenStax, "Biological Macromolecules", https://www.quizover.com/course/section/secondary-structure-proteins-by-openstax

We explain every level very briefly for better understanding of the input in our considered case study. However, the first level (primary sequence) is the most important one in this thesis and hence, is explained in more detail.

1. Primary Sequence: It is a linear order of amino acids in the polypeptide chain. These sequences are read as single letter codes from left to right. For eg. R-H-K-D-

E. Peptide bonds (a chemical bond between two amino acids) hold this structure together. Irrespective of the nature and type of the sequence, the primary structure is conventionally read from the N-terminus (amino terminus) to the C-terminus (carboxyl terminus), which are the two ends of the chain.

2. Secondary Structure: It is a conformation of amino acid residues, made of Alpha Helix, Beta strands and turns as shown in Figure 3.1. Thus, helices are visualized as spiral structures and strands as flattened ribbons.

3. Tertiary structure: It is a folded secondary structure forming a globular molecule. With modern technology, it is possible to Visualize a tertiary structure on a 3D-graph with x,y and z coordinates.

4. Quaternary structure: They comprise of multiple protein complexes. The interaction between proteins serves as the engine for its functions.

The case study considered in this thesis only deals with sequences of amino acids (primary sequence).

A peptide, in the context of proteomics, refers to fragment of a protein, obtained through sample preparation [68] [67]. Proteins are broken down into peptides by digestive enzymes [1]. The practical use of this case study is to identify the proteins which are a close match to a given peptide. For that, the researchers we collaborate with, have some pre-identified proteins derived from genomics experiments. Proteins are digested in silico using enzymes and create model peptides. Trypsin is used for our experiments. Details about how trypsin breaks the protein are explained in Section 3.1.2.

3.1.2 Mass Spectrometry Based Proteomics

Biological samples need to be processed in order to identify proteins in them. For this purpose, they are subjected to a device called mass spectrometer to separate their contents. This is the first step of the protein identification process. This method is called mass spectrometry and it separates the sample based on the masses of its components. The technology involved is described in this section.

Chromatographic Separation

Chromatography refers to separation of proteins [68]. It processes the sample (proteins and a liquid which is generally formed from cellular extract) and separates the analyte.

[1]Enzymes are proteins that catalyze chemical reactions. Proteins catalyze chemical reactions[68], which means they accelerate chemical processes but without disturbing the balance of other elements.

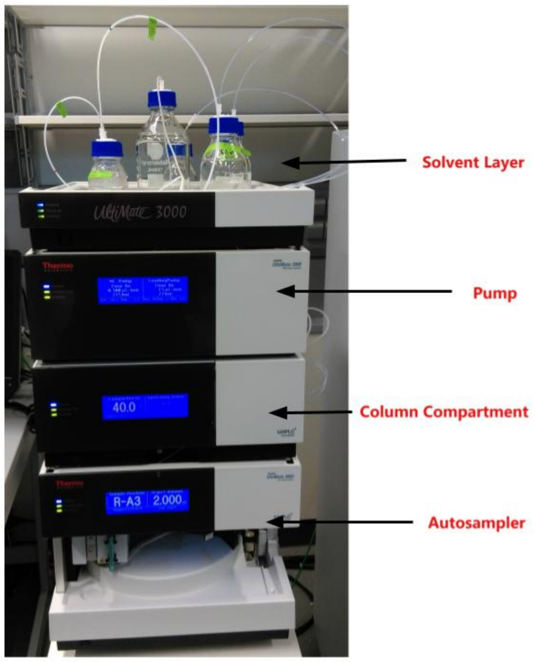

Figure 3.2: A protein sample fed to a chromatography system

A graph called chromatogram can then be created, with all (separated) elements in the sample. It plots the intensity against either time or volume. In an ideal situation, the highest peaks correspond to proteins of interest, thereby purifying the input. The case study uses the UltiMate 3000 model of a chromatographic instrument. Figure 3.2 shows the equipment used in the laboratory.

Mass Spectrometer

A mass spectrometer is a device that measures mass to charge ratio (m/z) and is thus capable of determining the mass of primary protein sequences [68]. It can be seen in Figure 3.7. It operates on the basis of m/z and consists of:

- Ion source: Ions are produced from the sample and they are charged either positively or negatively. Ions are generated either using laser technology or an electrospray[2].
- A mass analyzer: It performs the separation of ions based on their mass to charge ratio.
- Detector: It produces calculable signals.

[2]The protein sample mixed with cellular liquid is fed to an electrically charged needle. The drops that exit the needle are accelerated and a heated gas evaporates the solvent. The charged particles explode to form ions after a while[67]

Trypsin - Cleavage

Proteins are broken down to short peptides, using enzyme cleavage. The enzyme Trypsin cleaves the sample in the following way: The protein is split either on Arginine (R) or Lysine (K), without P directly following. This is called "Keil rule" [69].

3.1.2.1 Mass Spectra

A mass spectrum is a graph that plots the ion-intensity against ion-distribution (by mass) in the mixture. This plot of intensity on the Y-axis vs. mass-to-charge (m/z) ratio on the X-axis, represents the chemical analysis of fragmented samples.

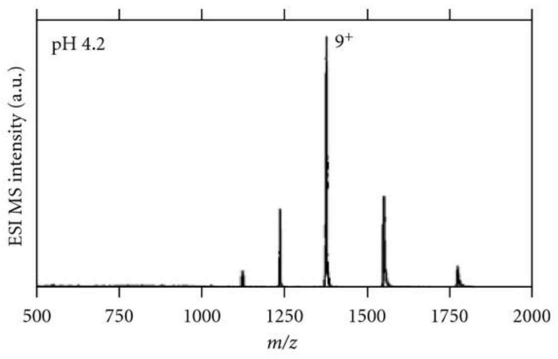

Figure 3.3: An example of mass spectra [3]

Figure 3.3 shows mass spectra from a sample of iron containing hemeprotein called cytochrome. The figure shows the separation of components and their intensities. Generally, the highest intensities are considered because they denote prominent substances. In proteomics, after tryptic digestion of proteins (Section 3.1.2), mass spectra of peptide ions is measured [19]. This is followed by detailing fragment ions to form sequences. Both steps combined, form an ms/ms spectra. The results can then be made durable.

3.1.2.2 Theoretical Spectra

The proteins, that we wish to match our samples against are termed as theoretical spectra. They are derived from one or more of the protein databases mentioned in Section 3.2. Theoretical spectra are mass spectra calculated from theoretical protein sequences. Technically speaking, a protein is digested by enzymes, like Trypsin. This results in formation of peptides. Peptides are further cleaved (Section 3.1.2) and fragmented, which results in theoretical spectra.

Data Format Of The Stored Theoretical Spectra

In the case study, the biological data is converted to a numeric format for calculation. There are several formats [3] available such as:

[3]https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=BlastHelp

- FASTA
  Proteins are stored as theoretical spectra in a format called FASTA. FASTA is a text based format that stores (biological) sequence data, such as the amino acid sequence (primary structure) of proteins using single-letter codes. This format describes the amino acid sequence in a single line, distinguished by a greater than sign ">" at the beginning. This follows the sequence of data.The following text is in FASTA format:

```
>mg|Testsequence_1 Methyl—coenzyme M reductase II subunit beta OS=
    Methanothermobacter thermautotrophicus
MPMYEDRIDLYGADGKLLEEDVPLEAVSPLKNPTIANLVSDVKRSVAVNLAGIEGSLKKA
ALGGKSNFIPGREVELPIVENAEAIAEKVKRLVQTSEDDDTNIRLINNGQQILVQVPTTR
MGVAADYTVSALVTGAAVVQAIIDEFDVDMFDANAVKTAVMGRYPQTVDFTGANLSTLLG
PPVLLEGLGYGLRNIMANHVVAITRKNTLNASALSSILEQTAMFETGDAVGAFERMHLLG
LAYQGLNANNLLFDLVKENSKGTVGTVIASLVERAIEDRVIKVASEMKSGYKMYEPADWA
LWNAYAATGLLAATIVNVGAARAAQGVASTVLYYNDILEYETGLPGVDFGRAMGTAVGFS
FFSHSIYGGGGPGIFHGNHVVTRHSKGFALPCVAAAMCLDAGTQMFSVEKTSGLIGSVYS
EIDYFREPIVNVAKGAAEIKDQL
```

Listing 3.1: FASTA example

- Bare Sequence
  This format is similar to FASTA, except that it is devoid of the first metadata line. It is simply a sequence with or without spaces at regular intervals. Such a format does not allow blank lines within the sequence because otherwise it is treated as the next item.

- Identifiers
  This format only has the accession number (a unique number given to a protein sequence in a database) and the version, the latter being optional. The two can be separated by a ".".

```
ABA68817.2
```

Listing 3.2: Identifiers example

Theoretical spectra can be obtained in the given formats. The current version of BAXdem is compatible with FASTA. However, if it were used in production, it could be made compatible with all the formats by making changes in the FASTA parser.

3.1.2.3 Experimental Spectra

Biological samples are digitized in the form of spectra. They are visualized as line graphs, with lines corresponding to the elements within. Experimental spectra are MS/MS mass spectra resulting from the measurement of a protein sample with the mass spectrometer. The case study requires us to find which proteins a sample of peptides originally belonged to, because their origin is no longer easily recognizable.

Data Format Of The Obtained Experimental Spectra

Information obtained from a mass spectrometer is often stored as text files with data about the sample. This section describes some of the conventional formats [70].

- MGF
  MGF stands for Mascot Generic Format, developed by Matrix Science for their tool called Mascot in the United Kingdom. Mascot is a widely used proteomics search engine. MGF files are bounded by statements “BEGIN IONS” and “END IONS”. They contain some metadata in their header area, which includes their title, mass, charge quantity and other information. Each MGF spectrum consists of two columns: the m/z value (to the left) and the intensity (to the right). An optional charge quantity forms the right most component along with a “+” or “-” to signify if it was a positive or negative charge.

```
BEGIN IONS
TITLE=Spectrum000107 RHPYFYAPELLYYANK +2 y— and b—series
PEPMASS=1023.01766  2000000
CHARGE=2+
RTINSECONDS=200
SCANS=25000
147.11285   100000  1+
157.10843   100000  1+
261.15578   100000  1+
294.16734   100000  1+
...         ...     ...
...         ...     ...
...         ...     ...
...         ...     ...
...         ...     ...

END IONS
```

Listing 3.3: MGF example

  Peptide information is represented as MGF files which are fed to BAXdem and are post-processed to match with FASTA.

- DTA
  This was one of the earliest MS/MS formats. Each spectrum was written to a separate file with a header describing its charge and mass. This was followed by all the m/z-intensity pairs that characterize the given spectrum. The filename was made up of metadata such as the scans and charge. Since the spectra are separated in files, concatenation tools needed to be used to aggregate the data.

- PKL and MS2
  These formats are similar to DTA, except the header contained more metadata and

provided a means to produce a series of spectra in a single file. The large number of spectra produced constantly, the storage required for the same, the inability to share headers and small sizes of individual files necessitated the unification of the output files [71].

We have thus covered the essential terms and concept of biology to understand the process of X!Tandem and BAXdem.

3.1.3 Biological Experiment Setup

The complete setup to produce peptides involves the biological specimen going through a number of steps, as shown in Figure 3.4.

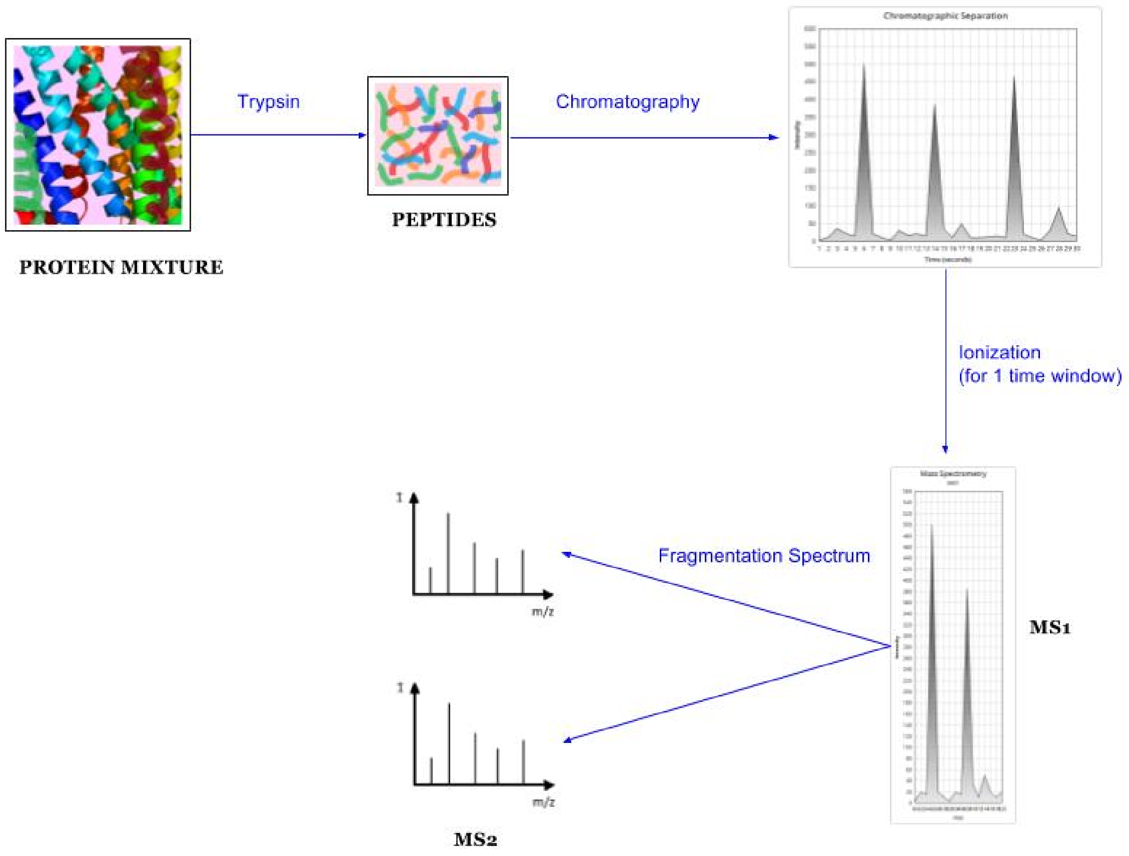

Figure 3.4: Experimental Setup for peptide generation

The process is as follows:

1. A protein mixture is first obtained. In our experiments we had some protein samples in the form of Influenza viruses, polluted water samples and drug infused human samples.

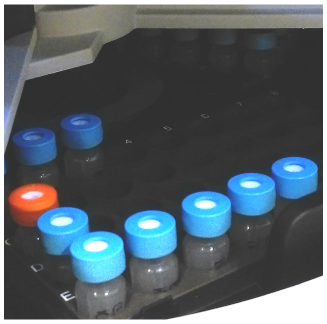

Figure 3.5: A protein sample fed to a chromatography system

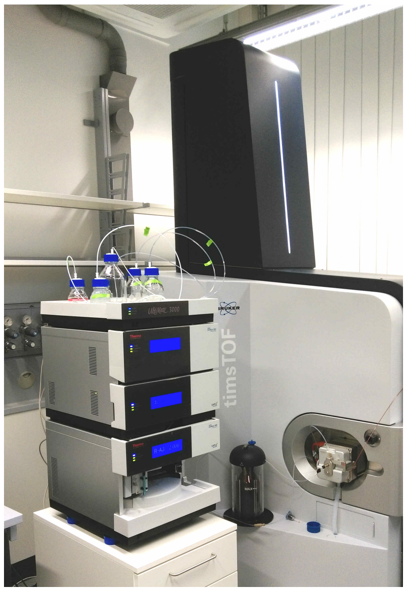

Figure 3.6: Physical experimental setup

2. The mixture is initially mixed with digestive enzymes(explained in Chapter 2) to break them into peptides.

3. The sample is collected in tubes and fed to a chromatography system in the auto-sampler tray as shown in Figure 3.5.

   The sample is extracted from the tray using the injection technique and loaded into the machine for further processing. Elements in the mixture are thus separated by chromatography, making a distinction between peptides, contamination or other elements in the solution mixture. The chromatographic device connected to the mass spectrometer is shown in Figure 3.6.

   The larger machine on the right is the mass spectrometer, the output of which can be represented as a graph (as shown in Figure 3.4) with peaks corresponding to various elements.

4. The purpose of the mass spectrometer is explained in Section 3.1.2. This section describes how it works internally. For ease of explanation, the process is divided into two parts: Ionization (Figure 3.7) and Mass separation(Figure 3.8).

   (a) Ionization: Before starting the process, the sample is ionized as shown in Figure 3.7. In our experiment, ionization is performed using an electrospray in the following manner:

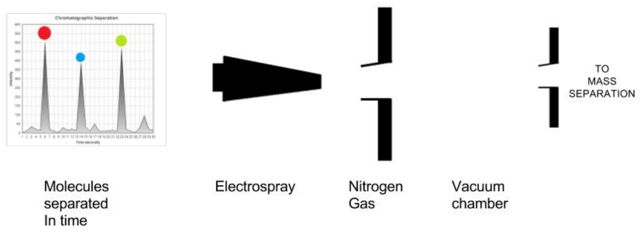

Figure 3.7: Ionization in Mass spectrometry

   The sample-mixture is dispersed using an electrospray inside the mass spectrometer as shown in Figure 3.7. The solvents evaporate in the process, leaving behind the significant elements. Ionization is performed by kicking off one or more electrons, thereby gaining a positive charge. An inert gas, Nitrogen in our setup, is used as the carrier of ions into the vacuum chamber.

   (b) Acceleration: Figure 3.8 describes the mass separation process.

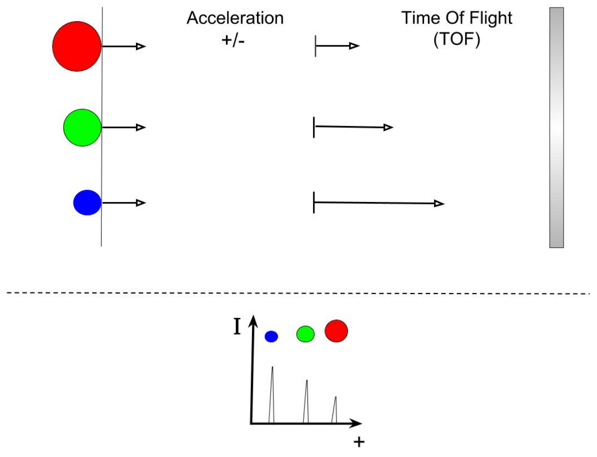

Figure 3.8: Mass separation in Mass spectrometry

   The ions are set in motion by introducing the same amount of kinetic energy in each of them. This is possible when the ions are moved from vacuum to a strongly charged electrical field. The ions are then beamed into a magnetic field with different velocities. Since ions of different elements have different masses and by that logic, different velocities, their Time of Flight (TOF) to a deflector surface varies thus forming different spectra, as Figure 3.8 depicts.

(c) Spectra are identified using a digitized technique, forming a graph with the distinction.

5. Once we have the mass spectrometry for one time window, we create a graph (MS1) with peaks identifying the peptides. At this point, it is still possible to identify which proteins they belong to.

6. In the next step, we take each peak from the result of the previous step, and fragment the spectra further, creating as many number of MS2 graphs, as the peaks in MS1. This is where it becomes impossible to identify which protein the fragmented peptide belongs to. Evidently, this is the starting point for this thesis.

3.2 Protein Databases

The input data to the experiments comes from protein databases. There are four highly recommended databases viz.

- Uniprot[4]: Europe

  It is a database of protein sequences and functions. It is an association of European Bioinformatics Institute[5], the Swiss Institute of Bioinformatics[6], and the Protein Information Resource[7].

- NCBI[8]: America

  The National Center for Biotechnology Information(NCBI) provides biomedical,genetic and genomic data to uncover new knowledge and thus understand the molecular and genetic processes so as to maintain health and control diseases.

- PDBj[9]: Japan

  The Protein Data Bank Japan contains macromolecular structures and some tools. It is has an association with RCSB(Rutgers and San Diego Universities), BMRB(Biological Magnetic Resonance Data Bank: America) and PDBe(Protein Data Bank in Europe: Europe),

- Customized Databases:

  There are some manually databases built from existing ones for better efficiency.

[4]European Bioinformatics Institute and SIB Swiss Institute of Bioinformatics and Protein Information Resource (PIR), http://www.uniprot.org/blast/

[5]A centre for research and services in Bioinformatics, supported by several European states

[6]An academic non-profit foundation, Switzerland

[7]An integrated public resource for Bioinformatics at Georgetown University Medical Center (United States of America)

[8]National Center for Biotechnology Information, https://www.ncbi.nlm.nih.gov/

[9]Members of Protein Data Bank Japan, https://pdbj.org/

These are the protein databases, available worldwide.

3.3 Protein Search Engines

X!Tandem[22] is a well known protein database search algorithm. Other algorithms like Sequest [23] and Mascot [24] are also used for the same purpose, with some differences in their working.

Sequest [23] searches amino acid sequences in a protein database which fall within the range of +/-1 atomic mass unit. It uses the mathematical cross-correlation[10] function to find out the resemblance between the m/z values of the given ion and those obtained from the database. It operates in four phases:

1. Reduction of files and file size of the tandem mass spectra
2. Filtering the protein database for matches, by comparing the molecular weights
3. Comparing the top 500 matches obtained in step 2 with the given amino acid sequence
4. Calculating a score for the matches and ranking them to find the best match

Thus Sequest, like other search engines, allows a given protein or peptide to be identified by its weight and fragmentation patterns, without manually interpreting the embedded data.

Mascot [24] is a probability-based protein search engine. The score of a match is the probability of the match being random. A high score has a low probability given the massive protein database entries. Therefore a threshold to select the best matches is crucial and the scores are denoted as $-10 * LOG_{10}(P)$, where P is the probability. An accepted frequency of randomness is 5%. Proteins selected thereby are considered as significant matches. Through multiple runs, the best match is selected from the significant matches by eliminating the candidates that have a probability greater than the threshold level.

X!Tandem [22] is a protein search engine based on similarity scoring. It operates in two phases [72]:

1. Select proteins from a protein database, that have a probability of being a match to the spectra: which needs to be identified. This is a naive match, with several false positives.
2. Compare the proteins with possible matches. Select proteins from the results and further filter them. Possible modifications e.g. missed cleavage (ref) and translation is now considered to filter the results.

[10]Cross-correlation: A function finds out the similarity between two given items, usually used to compare digital signals.

In the pre-implementation phase, samples of data for case study were tested with different protein search engines. The results of X!Tandem were the most suitable for them and therefore was chosen as the protein search engine for BAXdem. It will be explained in detail in Chapter 3.

3.4 X!Tandem: Working And Data Pipeline

Our implementation for a protein identification prototype is based on X!Tandem[22][73][74]. We select X!Tandem as a representative algorithm for protein identification. Considering that all protein identification algorithms share a common workflow, where experimental spectra is matched against theoretical ones. Our choice of algorithm has limited impact on the proposed architecture. Other algorithms could be plugged into our prototypical implementation, making it more versatile and adaptive. This can be done by replacing the component called scorer (in detail in Section 4.1), and replacing it with the logic of another desired search algorithm.

X!Tandem is an algorithm that identifies proteins from peptides (explained in Section 3.1.1). This thesis revolves around it, making it important to understand. However, we do not mimic the entire algorithm as we were time-bound and X!Tandem's massive amount of processing slightly interrupts with the wanted results. We only implement some parts of it which were critical.

3.4.1 Structure And Data-flow

X!Tandem's is an algorithm expressed in C++ with several modules interacting with each other. Its project organization helps understand the workflow and compilation method. It is shown in Figure 3.9 and best explained in the following way:
The folder “src” contains the source files i.e. header and implementation files. These files are compiled and cleaned to form binary files, stored in “bin”. Additionally they also contain the mass spectra. “fasta” contains the protein database file i.e. FASTA file. An XML file provides the configuration parameters. This is passed via command-line to one of the modules, which in turn calls other modules. Most of them have a header file (.h) and an implementation file (.cpp). In the end, the output is again stored in an XML file.

We only discuss a few important and relevant modules of X!Tandem in this section, that explains parts of the entire process in short.

- loadspectrum:
  This module standardizes and loads data from various mass spectrum file formats (files with ms/ms spectra) into a class named mspectrum.

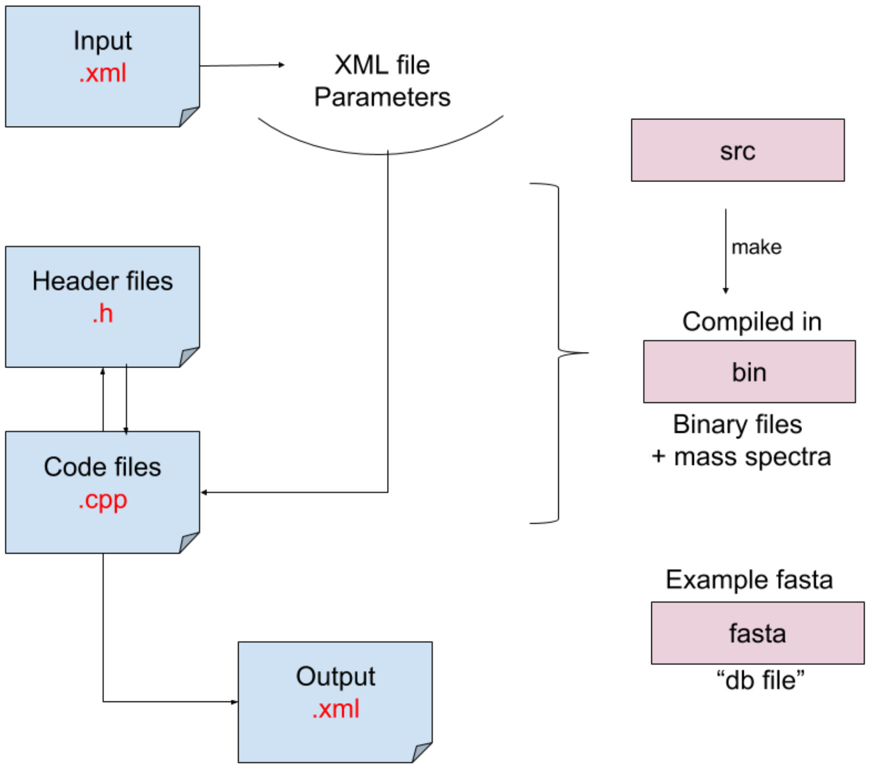

Figure 3.9: Software artifacts of X!Tandem

- masscalc:
  Transformation of proteins (FASTA data) requires calculation of molecular masses of individual atoms.

- mcleave:
  The mcleave module takes cleavage and residue information (as configured), parses the experimental protein, and then cleaves it. Trypsin, is passed as a string with value "[KR]|P", which instructs a cleavage on K or R, when it is not followed by P.

- mprocess:
  This is the main processing class. It orchestrates the complete protein identification program. It contains information about input parameters (m_xmlValues), performance (m_xmlPerformance), and the mass spectra in question (m_vSpectra).

- mscore:
  Mscore is responsible for comparing one protein sequence (FASTA) with several tandem mass spectra (MGF, in our case). It not only calculates the score, by calling pluggable scoring modules, it also optimizes the performance of scoring by using filters on the candidates (reducing the number of comparisons).

- mscore_tandem:
  This module is one example of pluggable scores in X!Tandem. It carries out the

dot product, i.e., the core of the scoring of peptides against mass spectra. It calculates convolution scores and hyper scores, used for further report generation on the protein identification results.

- msequenceserver:
  Proteins from a FASTA file are loaded as a list of amino acid sequences in a container called msequencecontainer by the msequenceserver. This encompasses an "interpreting-on-the-fly" functionality for theoretical spectra to be matched.

- mspectrumconditions:
  In addition, before the dot product, mspectrumconditions transforms the mass spectra to make them suitable for searching. Details of relevant spectra processing will be given in Section 4.1 - Section 4.1.2.

Based on the modules we mentioned, we present a step-by-step process describing protein identification in X!Tandem:

1. An input sequence of experimental spectra is processed sequentially using threading.
2. Configurable filtering criteria are employed to rule out potential non-matches.
3. For each experimental spectra, preloaded and un-interpreted FASTA input is considered as potential matches.
4. Only candidates passing some further criteria are interpreted and loaded for specific similarity scoring per experimental spectra.
5. These results are then collected for each experimental spectra and utilized to create histograms/reports.

These steps can be summarized as: ingestion, pair-building, scoring and score collection/visualization. We will now describe the scoring functions that the algorithm uses.

3.4.2 Scoring Functions

This section explains the different types of protein and similarity scores used in X!Tandem.

Dot Product

To match a peptide (explained in Section 3.1.1) against a protein, X!Tandem matches the level of overlap between them. The experimental spectrum (explained in Section 3.1.2.3) is reduced to only match peaks in the theoretical spectrum (explained in Section 3.1.2.2) and then calculates the dot product. The formula for dot product is as follows:

$$dp = \sum_{i=0}^{n} I_i P_i \tag{3.1}$$

where,
dp = Dot Product
I_i = Fragment of ion intensities derived from experimental spectrum (Section 3.1.2.3)
P_i = Presence in theoretical spectrum (Section 3.1.2.2) which is either 0 or 1
This eradicates the less probable matches by changing the value to zero for that possibility and thus enhancing the algorithm.

M-score

Computationally speaking, the representation is in form of two vectors with two values at each position[74]. The first vector(m_pplType) is a copy of the errors for the peptide's mass in the spectrum. Vector 2 is a stack of a vector of integer masses of peptides(m_plSeq) and a vector of weight factors for each m_plSeq(m_pfSeq). The two positions in each vector can be described as follows. One holds the value to be matched(m_lm) and the other to be multiplied(m_fI).

The total number of matches is called the "L-count". For all the positions that the m_lm value matches, their products are added to each other, this is calculated as the "F-score".

Additional Scores

The E-value is a score that measures the distance between the best match and the others.
Hyperscore measures the closeness between the spectrum and peptide model.

3.5 Concluding Remarks

To summarize this chapter, the source of peptides, and the important attributes of the input data are described, along with the databases that provide theoretical spectra. We discuss some available options for algorithms for protein identification. And finally, we explain in detail the chosen algorithm: X!Tandem. In the next chapter, we describe the prototype created to support this case study: BAXdem.

4. PROTOTYPE: BAXDEM

In this chapter we present our prototype for hardware accelerated big/fast data framework: BAXdem (Big Accelerated X!tandem).

- We start by presenting the structure, as we explain each of its constituent components in Section 4.1.
- In Section 4.2 the technologies we adopted, the motivation behind our choice of these technologies and specific version details which could be relevant to our tests, are described.
- To conclude we discuss how BAXdem compares to X!Tandem in Section 4.3.

4.1 BAXdem: Working And Architecture

Based on the workflow of X!Tandem, we propose a solution for a prototypical implementation covering some of the core functionality for protein identification, using frameworks for scalable large scale processing and hardware-acceleration.

BAXdem follows a modular architecture that integrates a stream ingestion framework for communication (Kafka), a cluster database (Cassandra) for managing theoretical spectra and storing scoring results, and a series of scoped programs to handle the complete process. The core implementation in BAXdem takes place in the latter programs. This was implemented in Python, which was our choice so we could include programming libraries capable of GPU-acceleration (TensorFlow).

Apart from these architectural components, some set up scripts and data loading programs are included with BAXdem. For example, the first task to be done for running BAXdem is to start a Cassandra script. This creates the keyspaces and tables wtihin. Secondly, input FASTA files have to be fed to an external FASTA loader. Then, the system is ready to run.

In Figure 4.1 we show the main components of BAXdem. In the following section, we discuss the components for data loading, pair building and scoring.

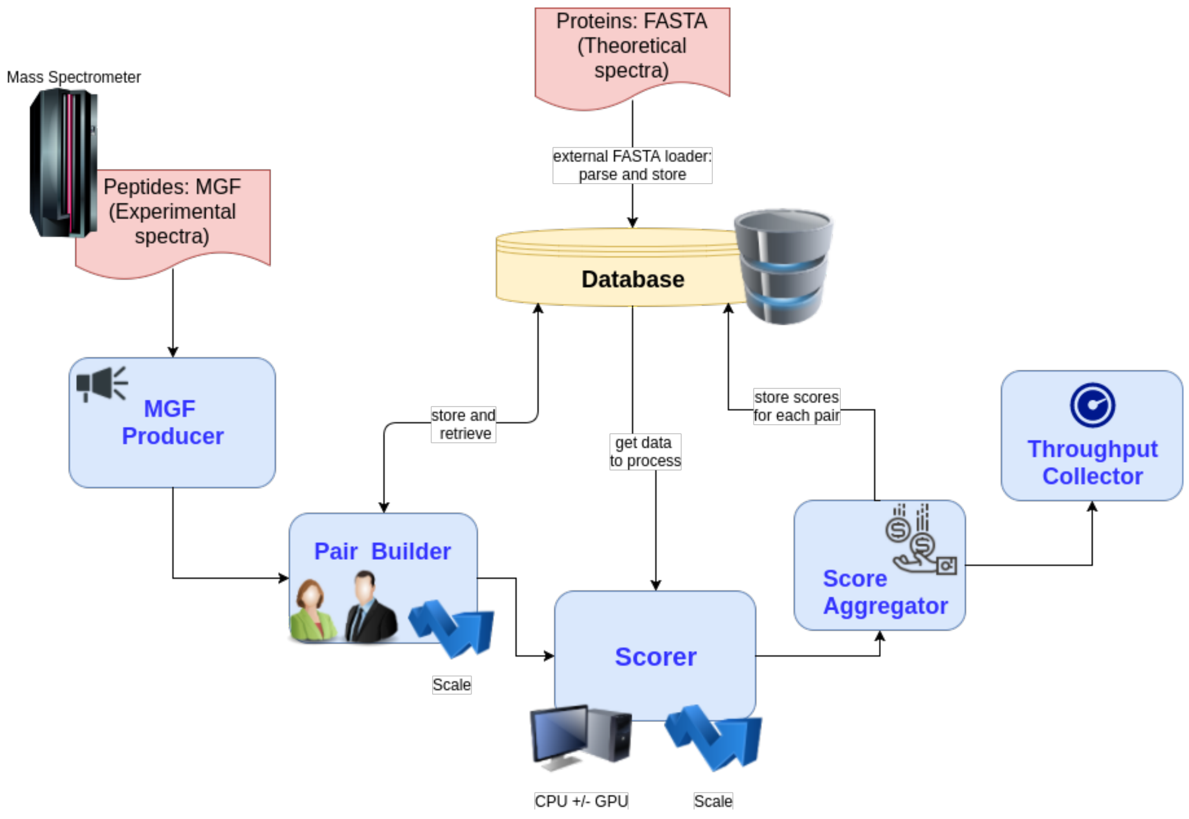

Figure 4.1: Architecture of the implemented prototype for the case study

The architecture can be better explained when broken down as the following modules:

4.1.1 Input Files

Theoretical spectra is obtained from FASTA files, or in other words, this is where the stream is ingested in the distributed fast data framework. These files are parsed into an array representation (m/z values of amino acids) and stored in Cassandra, under the keyspace "fasta". We use an external FASTA processor using the mzJava library[1] to parse the files, convert them to a 2D array format, and then store them in the database. This external tool is a Java application (.jar) that creates an upload folder on the local system and uploads the entire file to it as a byte-stream. The stream is parsed linewise, until it finds a protein. When a complete text of a protein is found (starting with >, explained in item Section 3.1.2.2 under Section 3.1.1), it is digested (using Trypsin logic, explained in Section 3.1.2). This forms peptides, that we use in scoring a match. Peptides are connected to their parent proteins via UUIDs. All of this data is mapped to Cassandra tables.

[1]https://mzjava.expasy.org/

The mass spectrometer, provides the piece of the puzzle that needs fitting: the experimental spectra (MGF data: explained in Section 3.1.2.3). Protein data for BAXdem is derived from FASTA format files. To enable them to be compared to MGF files, this spectra-data undergoes some post-processing. This file is also parsed and converted to an array of mass and intensity. It is passed to a module (MGF Producer) that sends these MGF files, spectra by spectra as a stream of micro batches. While collecting an entire spectra, this module also calculates the highest intensity of each spectra and passes it as a Kafka message key along with the spectra, since it is needed at a later stage.

4.1.2 Pair Builder

Each spectra is passed to the pair builder module (using the communication provided by the large-scale processing framework of choice). This module pairs the experimental spectra, with selected theoretical spectra. The experiments performed are mainly operated on theoretical spectra provided by UniProt (Footnote 4).

The spectra which is passed contains the identifier and the highest intensity of the peak, in addition to the contents of the mass spectra. The pair module outputs two things: metadata, i.e., how many pairs each of the experimental spectra had to the score collector, and real data: IDs of the matched pairs to the scorer. The purpose of creating pairs is to shortlist which proteins the peptides belong to, enabling the next module to identify the best and most probable match. To optimize the scoring, we implement five core filtering conditions [22][74] from X!Tandem for altering the mass spectra[2]:

1. Isotope deletion:
   This filter gets rid of isotopes. Mathematically speaking, it removes multiple entries within 0.95 Daltons of each other.

2. Remove small peaks:
   The intensities of the tandem mass spectra are normalized using the following formula:
   $$I_{norm} = I_{old}/(P_{high}/R_{dyn}) \tag{4.1}$$
   where,
   I_{norm} = Normalized intensity
   I_{old} = Original intensity
   P_{high} = Highest intensity in the spectra
   R_{dyn} = Dynamic range
   We set a filtering threshold of 1.0. All the peaks of spectra, with intensities lower than 1.0 are neglected.

[2]We note that other criteria, included into the X!Tandem process during the interpretation of the theoretical data, was not included into our prototype since it seemed beyond the scope of a basic implementation.

3. Isotope refining:
   For each spectrum peak, we look behind. If the current peak's m/z value is greater than 200 or the intensity is smaller than 1.5, the previous peak is not included.

4. Best peaks:
   max_peaks is a configurable variable, that sets the maximum number of peaks to be used. As default, we use 50 max_peaks.

5. Blur:
   This filter increases the numbers of peaks threefold. Let's consider one of the peaks as P_curr composed of m/z value MZ_curr and intensity I_curr. The following peak is added before P_curr:

   $$P_{prev} = (floor(P_{curr})/0.4) - 1 \quad \text{and} \quad I_{curr} \tag{4.2}$$

   P_{prev} is composed of MZ value calculated as "floor(P_{curr}) / 0.4" and intensity remains the same. The following peak is added after P_curr:

   $$P_{next} = (floor(P_{curr})/0.4) + 1 \quad \text{and} \quad I_{curr} \tag{4.3}$$

   The floor function rounds up a float to the lower integer, which is divided by 0.4. In the previous peak 1 is subtracted from the m/z value and 1 is added to the next.

   After these five steps, the spectrum is now suitable for searching and comparing.

On the other hand, to enhance the pairing of tandem mass spectra with peptides, we apply the following rules, like X!Tandem:

1. Only those theoretical spectra which have a peptide length of 5 to 50 are considered [22][73] to reduce the false discovery rate (FDR).

2. A tolerance factor caled "mdh" is created and calculated as follows [22][74]:

   $$mdh = ((pepmass - prot) * chg) + prot \tag{4.4}$$

   where,
   pepmass = peptide mass of the current experimental spectra
   prot = mass of a proton i.e. 1.00728
   chg = charge of the current experimental spectra
   Moreover, if the peptide mass of the experimental spectra is over 1000, we subtract the mass of neutron (1.003355) from "mdh". We then create a lower and upper limit to select theoretical spectra. We only choose those to pair which have a peptide mass between "low" and "high", with "low" and "high" calculated as:

   $$low = mdh - (mdh/10000) \tag{4.5}$$

   and

   $$high = mdh + (mdh/10000) \tag{4.6}$$

The filter aims at reducing the false positives, since two similar peptides with a prominent difference in masses can never belong to the same protein.

4.1.3 Scorer

The scorer receives each pair, meaning its IDs. It then fetches the data of these pairs from the large-scale distributed database of choice, and it calculates the M-score for each of them. Finally it sends all pairs along with their scores to the next module.

The scoring is described as follows:
Each MGF (experimental) spectra is matched with every peptide candidate (theoretical spectra) and a score is calculated for each of them, creating a cross join effect but selecting only the closest matches. This is when we can expect the impact of scale-out processing to be observed.

The scorer needs to be highly scalable. For faster processing, this module can also run a part of it on the GPU. Either libraries, parallel programming languages or specialized compilers can be used for this module. The core of the X!Tandem algorithm: the dot product, is implemented in this module.

4.1.4 Score Collector

The resulting scores, along with the pair IDs, are sent to the next module i.e. score collector. The score collector is tasked with multiprocessing data from two topics. It collects metadata regarding the number of pairs from the pairbuilder, and scores for each pair from the scorer. The module creates <key,value> pairs in a dictionary for each MGF spectrum, with the unique identifier as the key, and the number of pairs it has, as its value.

It simultaneously (after a processing delay of some microseconds) receives the scores of MGF-FASTA pairs from the scorer. Each pair is structured as follows:

$< MGF \quad identifier, \quad FASTA \quad identifier, \quad score, \quad time \quad consumption >$

On receiving this structure, the score collector creates objects for each item, with attributes such as a unique identifier, score with a match, and number of remaining matches. The remaining matches is set to the number of pairs that an MGF has (from before). It decrements the remaining-pairs-count for every score that it receives for a given MGF. Since the score collector already knows how many pairs to wait for, it collects all the scores until the remaining-pairs-count becomes zero. This is necessary because of the real-time processing style.

Once the score collector has all the scores, the pairs are sorted in the decreasing order of their scores. Therefore the top is always the highest score, meaning, the most probable match. This data is then sent to the results/throughput collector.

4.1.5 Results Collector

The results collector basically collects all matches of experimental spectra and the time they took for processing, and stores it in the distributed database.

Thus, we design the architecture of a scalable system that can help to identify peptides through large-scale processing. By making our system depend on common scalable components, such as a large-scale processing framework, a distributed database and a solution for hardware-acceleration, we offer a design that can be ported to different technology choices.

The source code of our first implementation of BAXdem, using a specific selection of technologies, can be found on Github at: https://github.com/apoorva-ovgu/ProteinIdentificationGPU. In the next section we discuss the technologies used to support BAXdem.

4.2 Data Processing Setup And Architectural Decisions

This section mentions which software technologies were chosen for his research, and explains them in short. The motivation for our choice and version information is also disclosed.

Our choice for large scale processing: Kafka

We selected Kafka as a stream ingestion framework, because of its support for streaming and the fact that it grants developers good control over process placement, which is achieved through consumer placement. Such a control might not be as straight-forward with systems like Spark. Furthermore, subtle effects in process-placement in systems like Spark, could influence our results, by using Kafka we aimed for implementation which could contribute to repeatable studies. MapReduce systems were not chosen since our application does not follow exactly this model. Tools like Kinesis were not selected because of their lack of open-source availability and requirements for purchasing their product for using the system.

We use kafka version 2.11 for message passing through the system for both data and metadata. The components and dataflow of Kafka in the implemented architecture can be seen in Figure 2.10 in Chapter 2. Our system has the following kafka components described as $TopicName(content)$, $ProducerName(content)$, $ConsumerName(topic)$:

- Topics: topic_mgf (MGFs), numofmatches (MGF metadata), pairs (MGF-FASTA matches), scores (scores for every match), results (hughest score for every MGF spectra)
- Producers: producer_c1 (MGFs), producer_uidMatches (MGF-FASTA matches), producer_c2 (matched pairs), producer_c3 (scores for every match)
- Consumers: consumer_c2 (topic_mgf), consumer_c3 (pairs), consumer_scores (scores), consumer_uidMatches (numofmatches)

Thus we use a kafka cluster as a queuing system in a streaming environment. Each module creates and listens to a specific relevant topic. As far as the metadata is concerned, IDs, number of pairs and time consumed for an action is frequently passed through the system. We use key-value based messages which allows us to send data and metadata in the same message.

Topics are crucial to our design. Messages in different areas are isolated by different topics. When subscribed, these messages are received by the interested party. Its characteristics and architecture helps us isolate data and smoothen communication and data passing through the system.

Our choice for a distributed database: Cassandra

Regarding the selection of a distributed database, we selected Cassandra based on the expectation that it could be one of the best choice for scalability and availability (at the cost of consistency on partition failures), with a combination of generality (being a key-value store) and SQL-like features. Systems like Redis or HBase could've been similar choices.

We did not select to use CockroachDB because of it's limited adoption in the market, suggesting that it is a tool still in development. We did not select a graph database or a document store, since our use case did not seem to require the specialized queries or functionalities related to these data models (for example, we do not need graph traversals or deeply nested document structures). Systems like MongoDB, although popular, do not fulfill the same expectations for scalability of Cassandra [75].

Other reasons for choosing Cassandra are based on a discussion by Hewitt et al. [2]. In their work, they propose that Cassandra is a good fit for a setup where the project involves one or more of the following:

1. Large deployments:
   Cassandra is good for scaled multi-node architectures. Although it runs in standalone mode, the optimizations may cause an overhead and thereby lower the quality of service.

2. Several writes:
   Cassandra performs better writes than scans. Updates caused by spikes, user reviews at irregular intervals and such non-uniform write queries can be performed very efficiently and speedily by Cassandra.

3. Geographical distribution:
   Configuring Cassandra as a multinode cluster is fairly simple. It is especially beneficial if data locality is achieved by having data near the query.

4. Evolving applications:
   Cassandra supports flexible schemas and therefore is a good option even for immature projects.

Regarding our choice of Cassandra, we should note that being an OLTP key-value store, Cassandra is not designed for large range queries. As a result, we consider that a better choice of distributed database might improve the observations in our case study.

In our case study, a large number of peptides (Section 3.1.1), proteins and scores (Section 3.4.2) are stored in a Cassandra database. Data is written as each spectrum arrives in the system, meaning it requires a lot of writes. We use a 3 node cluster and store the theoretical spectra in a distributed fashion. Also, we aim to have this setup integrated with the mass spectrometer some time in the future and test it as a truly real-time system. Therefore flexible schemas could be useful. Hence, Cassandra was chosen to be our database.

We use Cassandra version 3.11.1 in the experimental setup. Cassandra in our system is described as $keyspace(table1, table2, ..., tableN)$. The tables with a '#' sign mean that they were deprecated in the final version as they weren't critical and caused some overhead.

- fasta (pep_spec, #prot_pep, #prot_table)
- mgf (exp_spectrum)
- scores (psm)

The table pep_spec contains theoretical spectra, the sequences, masses and identification. In the early development stage, we also deployed tables prot_pep (containing the mapping information between peptides and proteins) and prot_table (containing protein information). However, since these tables were very large, it caused some overhead and slowed the system and performance. Therefore we only used all critical information and stored in pep_spec. This belongs to the keyspace "fasta". The keyspace "mgf" contains the table with experimental spectra generated by the producer. Finally, "psm" contains the scores of the matches.

Our choice of programming languages and solutions for GPU-acceleration: Python, Tensorflow and PyOpenCL

The project was written mainly in Python. This choice was based on the fact that Kafka and Cassandra, next to other components chosen for GPU acceleration (OpenCL and TensorFlow) are natively supported in this language.

Given the high compute intensity, the scorer module uses Tensorflow and python's openCL extension called pyopencl.

For the first one, our choice is based in the observation made on Section 2.5.3, that most hardware-accelerated libraries correspond to highly-specialized domains, and that TensorFlow represented a unique alternative in this field.

For the second one, our choice was between general (e.g. OpenCL) and vendor-specific languages (e.g. CUDA, which is tied to NVIDIA graphic cards). We selected the first alternative and considering our choice of Python, the election of PyOpenCL was determined.

Tensorflow

We select TensorFlow as a Python-based library offering support for GPU accelerated operations. To our knowledge this is one of the few mainstream, general, open-source possibilities for using GPU acceleration in a Big Data context.

We use Tensorflow to accelerate CPU calculations. The scorer module was originally designed and implemented using Tensorflow for both CPUs and GPUs. This transition from CPU to GPU would have been fast and seamless. But Tensorflow is a relatively young tool and does not have enough support and libraries to carry out the required functions on a GPU efficiently. In addition it does not allow multithreading. For this reason, the in the GPU version, Tensorflow is replaced by pyopencl.

PyOpenCL

We use Pyopencl to enable parallel programming on modern hardware. Pyopencl converts the function to parallel code, thereby making the process faster on GPUs. Parallel computation for GPU is implemented and passed to opencl context as follows:

```
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
prg = cl.Program(ctx, """
    __kernel void multiply(
    ushort exp_size,
    ushort theo_size,
    __global float *m_lM,      #experimental compare value
    __global float *m_fI,      #experimental score
    __global float *m_plSeq,   #theoreical compare value
    __global float *m_pfSeq,   #theoretical score
    __global float *c)
    {
      int gid = get_global_id(0);
      if (gid < theo_size){
       for (int i=0; i<exp_size; i++){
         if (m_lM[i] == m_plSeq[gid]){
           c[gid]= m_pfSeq[gid]*m_fI[i];
           return;
         }
       }
       c[gid] = 0;
      }
    }
    """).build()
```

Listing 4.1: GPU kernel proposed for batch-wise scoring

Batched data is passed to the code. First the context for opencl is created and is then passed to a buffer or queue. OpenCL code is passed to a variable as a string. Vectors are compared to each other in a parallel fashion and their corresponding scores are multiplied and stored in a variable c. The parallelism happens by a factor of N, N being the size of the resultant vector.

Cassandra, kafka and openCL are accessed using their Python APIs, cassandra-driver, kafka-python and pyopencl respectively. In addition, Python's tensorflow libraries was also used. CQL or Cassandra-SQL was used to query tables in Cassandra using Python libraries. We used Python 2.7 for implementation.

4.3 Algorithm And Architectural Comparison Of X!Tandem And BAXdem

This section describes the similarities and differences between X!Tandem and BAXdem. We start by specifying the logic used in both the algorithms, and then the language in which both were implemented.

Logic

X!Tandem is a massive and multi-pass algorithm, fine-tuned to identifying proteins. Figure 4.2 shows the state diagram of X!Tandem [3] on the left. It starts by supplying input path containing search constants such as error tolerance, threads, maximum and minimum values of parameters among others, to the program using an XML notation. It loads this information and starts the process. It then calculates individual scores based on refined iterations over the data. It calculates multiple scores and can create histograms from the results.

[3]http://www.thegpm.org/TANDEM/api/tandem-process-state.htm

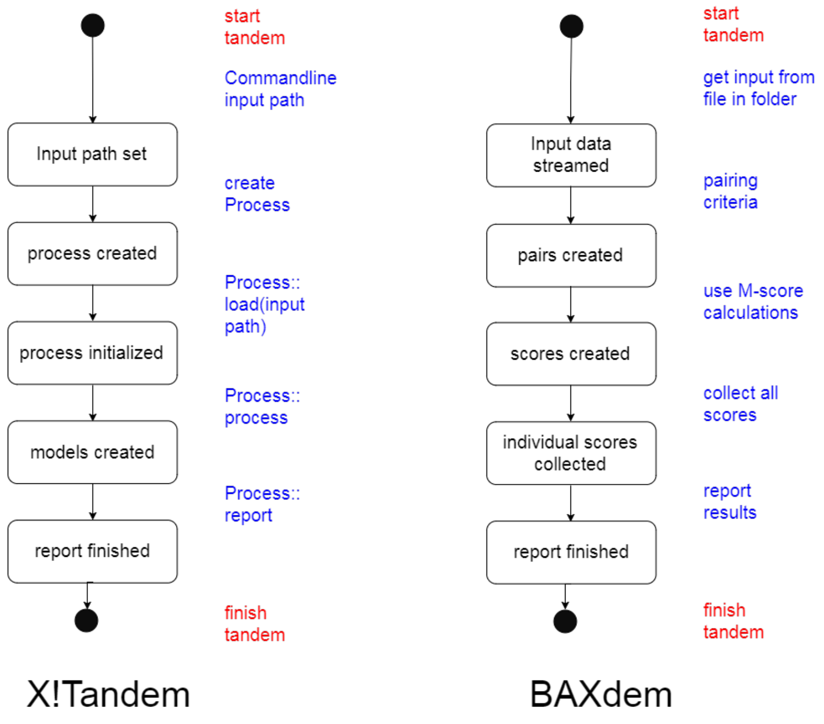

Figure 4.2: State diagram of X!Tandem and BAXdem

Our implementation (on the right in Figure 4.2) is simpler. It loads input data, stored in text format, from the given location (called "datafiles") which can be passed during runtime if required. The model is created in a per-spectrum basis on the fly. In other words, it creates pairs for each experimental spectrum as it arrives in the system. It only calculates one type of score: Mscore, and passes over the data one time, as opposed to X!Tandem. The system waits until all the pairs for a given spectrum is received, then collects all the scores and sorts them in descending order. The higher the score, the better the match. Data is stored to Cassandra as it arrives and the final scores are stored as well.

Thus, the core implementation logic of scoring is the same, but the rest of the working differs to enable achieving the goals of HPC and big data frameworks.

Language

X!Tandem was written in C++ while our implementation is mostly Python, and some OpenCL. C++ being a lower level language has speed at its advantage, but is known to have longer code and is not economical in terms of power and memory consumption [76]. Scripting languages can be faster for the processing time alone [77]. But the loading, pre-processing and post-processing makes scripting languages such as Python slower than C or C++.

Other key distinguishing factors are the use of message passing frameworks and cluster databases, which, although scalable, add overheads to our implementation.

We note that in this Thesis there is no performance comparison between X!Tandem and BAXdem, since the current version of the latter is only a prototype implementing basic features of X!Tandem, and, as a consequence, any comparison would be unfair to both tools.

4.4 Concluding Remarks

In summary, our prototype BAXdem is a modular implementation, with logic inspired by X!Tandem (but only implementing part of what X!Tandem covers). BAXdem uses Kafka, Cassandra and Python (with two libraries for GPU acceleration: Tensorflow and PyOpenCL). The implementation is made compatible with CPUs and GPUs. In our work we do not compare to X!Tandem, since BAXdem only implements a subset of X!Tandem tasks and thus, comparisons would be unfair.

We will now discuss the tests conducted on this prototype in Chapter 5.

5. TESTS

This chapter enlists the tests performed with BAXdem, to study the research questions, and its results.

Now that the case study, implementation, goals and working of our system are explained, we slowly move towards the results and tests conducted to study our research questions. We do so by comparing the two systems, the datasets, and speed of the system. Our goal is to find out if BAXdem is faster on the GPU, and by how much. In addition, we want to realize its scaling capabilities to ensure that it works with even high data loads by adding hardware if necessary. We organize our presentation as follows:

- We start by describing the datasets used for the experiments in Section 5.1.
- Then, Section 5.2 states the configuration of the hardware that was used to run the tests.
- This is followed by Section 5.3 which explains the tests and its results in detail. This is followed by a section that analyses individual components of the system.
- Finally, we evaluate the system, based on a queuing theory model, in Section 5.5.

Specifically with our tests we answer the following questions:

1. How does the performance improve by scaling the number of scorers in the CPU with big and small data? Is there any overhead related to message passing as data sizes increase? (Section 5.3.1)
2. To what extent can GPUs contribute to the performance of a scaled system? How does this compare to different CPU cases? Furthermore, what is the impact of batching for leveraging GPU performance and for reducing overheads from message passing? (Section 5.3.2)

3. What is the average impact of supporting technologies, and performance-impacting system configurations in the performance profile of BAXdem executions? (Section 5.4)

4. By looking at the system with a queueing theory model, can we determine the role of service and wait times in the resulting performance (thus, suggesting specific future directions for improvement)?Section 5.5

5.1 Dataset Description

For testing, we used small and big datasets of theoretical spectra and experimental spectra. These spectra are the output of the mass spectrometer, containing data from biological samples in a numeric format, making them calculable.

1. Experimental spectra dataset
   MGF files contain several experimental spectra. Each spectrum contains an average of 400 peaks. The small MGF consists of 10 spectra (approximately 4000 peaks) and the big dataset consists of a 30 spectra (approximately 12000 peaks). The experimental spectra originates from microorganisms, like those from contaminated water.

2. Theoretical spectra dataset
   Our theoretical spectrum was provided as FASTA files, with 553005 proteins, each of which is broken into hundreds of peptides, and stored in the database as a set of identifier, protein identifier, peptide sequence, mass and metadata. Most of the data is from UniProt's SwissProt. It is manually reviewed and annotated, thereby increasing its quality. It consists of protein samples from all over the environment including frog viruses, human fingers, African swine fever virus, organisms in strawberries, etc.

5.2 Hardware Setup

This section describes the hardware on which the tests were performed.

The system was implemented on a hybrid CPU/GPGPU architecture. It was tested on the same, and had the following configuration:
Memory: 7.6 GiB
Processor: Intel® Core$^{\text{TM}}$ i5 CPU 660
Co-processor: GeForce GTX 750/PCIe/SSE2
Speed: 3.33GHz
Cores: 4
OS: Ubuntu 16.04 64-bit

5.3 Tests

This section describes the results of the time consumption of BAXdem for various components and datasets. We start by showcasing our tests and results for BAXdem on the CPU, GPU and then finally a comparison of both approaches.

5.3.1 BAXdem On The CPU

Experiments were conducted on a CPU machine, as well as a hybrid CPU/GPU processor. This section describes the former. The tests commenced with profiling of the producers and consumers in the system, before moving on to testing its scalability.

Initial tests started with single node CPU machines. The correctness of this setup was acceptable, however, it took about a day to run. Therefore all further tests scaled-up the pair-builders to 4 and used a clustered setting.

Cassandra and Kafka were installed as a cluster across three nodes running a 64-bit Windows machine. The pair builder is scalable and the tested system requires the pair builder scaled at 4. Scaling is independent from placement. In our evaluations, unless stated otherwise we placed scaled items in a balanced manner across the nodes.

The scorer module, being compute intensive and having a potential for batches (explained in Section 5.3.2), was tested at different scales than the pair-builder: 1, 2, 4, 8, 16 and 32. Figure 5.1 shows the performance of our system processing small and big data at different levels of scaling. The X-axis shows the scaling factor and the Y-axis denotes the time required in seconds. The chart is log-scaled to make the results more visible.

The chart in Figure 5.1 is created from an average of the system over five stable runs. As expected, the total time required for the system to run on one node was the highest. The time considerably decreases while moving from a scaling factor of 1 to 2, 2 to 4 and 4 to 8. An increase in productivity by 50% or more is seen for small data processing. With a scaling factor of 16 and 32, some overhead is assumed as the execution-time gradually increases. Both big and small data perform the fastest on 8 nodes.

As opposed to small data, big data performs better on 8 to 32 nodes, rather than 2 to 8. This is as expected since more nodes can handle the load better, whereas the load is larger for lesser number of nodes.

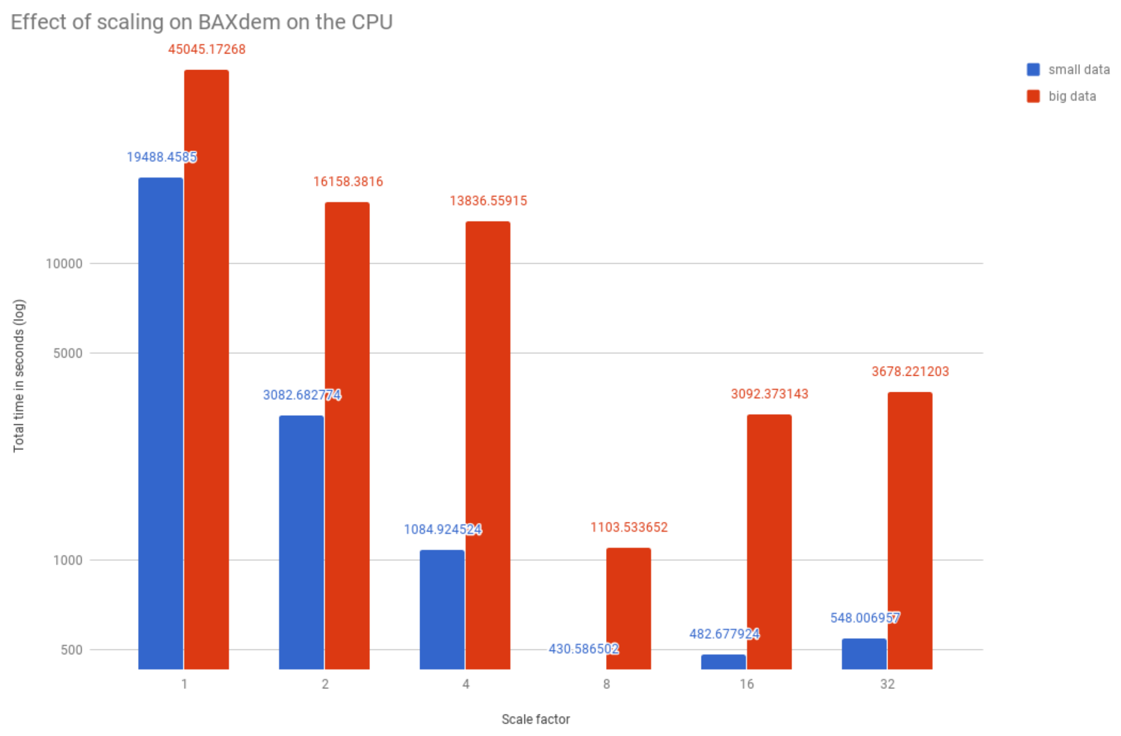

Figure 5.1: Effect of scaling scorers on the CPU

While running big data on nodes 1, 2 and 4 (especially 1, at times 2 and 4), Kafka's behavior was slightly unexpected. Not only did it break at times, it also at certain runs, did not pass along all the spectra, a small number of them were missing. We observe that big data results for a scaling factor of 1, 2 and 4 were noisy due to such behavior of Kafka, in spite of performing much more experimental repetitions.

Scalability and throughput

The chart in Figure 5.2 was created by measuring an average throughput (in terms of end-to-end processing of matches) per second.

The scalability of BAXdem, as the number of scorers increases, is evaluated in Figure 5.2. We observe close-to-linear improvements in total throughput for small data, except for the case of 32 scorers, where the overheads from non-deterministic Kafka and Cassandra operations damage our results.

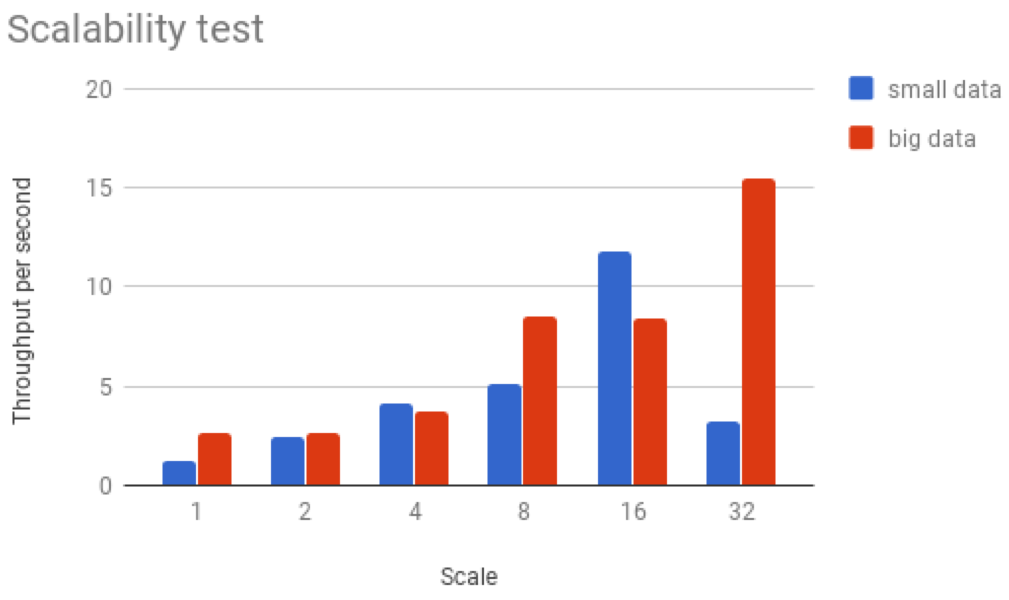

Figure 5.2: Scalability test

For big data we expect a sub-linear scaling, which can be reasonably ascribed to the bigger data sizes, adding more Kafka and Cassandra requests, increasing service and wait times. These overheads are more notable for big data sizes, suggesting that message passing can indeed add overheads to the overall execution. Such overheads can be alleviated with other configuration of the underlying frameworks and hardware. Network congestion, in particular, might have also influenced our observed results.

In spite of these limitations, the trends that we report in performance improvements indicate that our system is a coherently scalable system [4].

5.3.2 BAXdem On The GPU

Researchers believe that GPUs are sophisticated and complex to program [78]. However, the use of tools like Tensorflow (TF) can aid in simplifying the process.

Unfortunately, there are some limitations to this use. Among them, TF's single kernel execution which prevented us from concurrently running GPU kernels[1], thereby limiting our scaling capabilities. Specifically for our case, this prevented us from running 2 GPU-accelerated scorers in a single machine. One approach we attempted to overcome this was to enqueue kernels in a single TF process.

Another noteworthy limitation for adopting TF was that the single core GPU-accelerated operator in our case study was the dot product operation for scoring a pair of items. On early experiments we determined that simply running this on the GPU provided no

[1]https://github.com/tensorflow/tensorflow/issues/4552

gains, and that for our case study it was necessary to exploit parallelism at a higher level than simply the dot product on a pair.

As a result offering evaluations using TF would fail to answer our question on the contributions of the GPU in our scaled-out framework (as discussed in Section 5.3). Therefore we use for our evaluations parallel programming in pyopenCL, to exploit the advantages of a GPU. The specific details of our implementation for accelerating the pair scoring is presented in Section 4.2. In this section, we report the performance of our system, when run on GPUs.

5.3.2.1 Big Batches

The large wait times observed (i.e., time in sending Kafka messages and overheads for consumers to process them) encouraged us to batch the processing. Therefore, batches of 1000 messages were formed for testing on the GPU, with the added advantage of GPU behaviour (GPUs are best for calculating the same function multiple times, and their parallel capabilities make batching favorable).

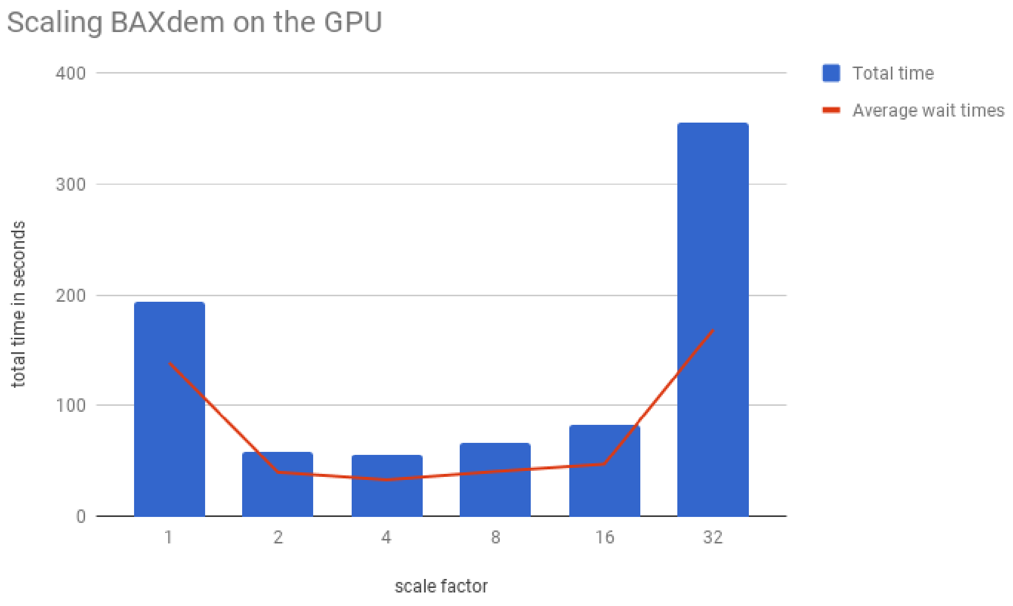

Figure 5.3: Effect of scaling with big batches on the GPU

The use of GPUs was indeed advantageous, as it reduced the run time drastically. As seen in Figure 5.3, it scales much better than CPU-only cases.

As opposed to the CPU version which gives its best performance at 8, the GPU gives its best performance at a scale factor of 4. The performance of concurrent GPU kernels we report, as evidenced when scaling over 2 scorers corresponds to a naive implementation, and, better system configurations might extend the limited performance gains observed when scaling beyond 4 scorers. BAXdem on a GPU is slower on a scale of 16, but still acceptable and better by far to the CPU counterpart. On the other hand, the overheads are larger with 32 scorers and the performance is poor.

5.3.2.2 Small Batches

Small batches (batches of 50 items) do not prove to be effective for scaling on the GPU, as evident in Figure 5.4. However, with only one scorer, the runtime is faster (although not more than 2 times faster) with small batches, than big batches. We discuss this later in this section. For every other scale size, big batches outperform small batches.

The total time required for scales of 1 to 4, gradually increase by a small margin. For a scale of 16 and 32 the latency is very high. The red line indicating the waiting time shows that the data spends a lot of time waiting in queues because the resources are fully utilized. In the case of 1 the wait time can be ascribed to the bulk of messages waiting to be processed by a single scorer, on the case of 32 the wait time can be ascribed to overheads related to Kafka and Cassandra, specially Kafka.

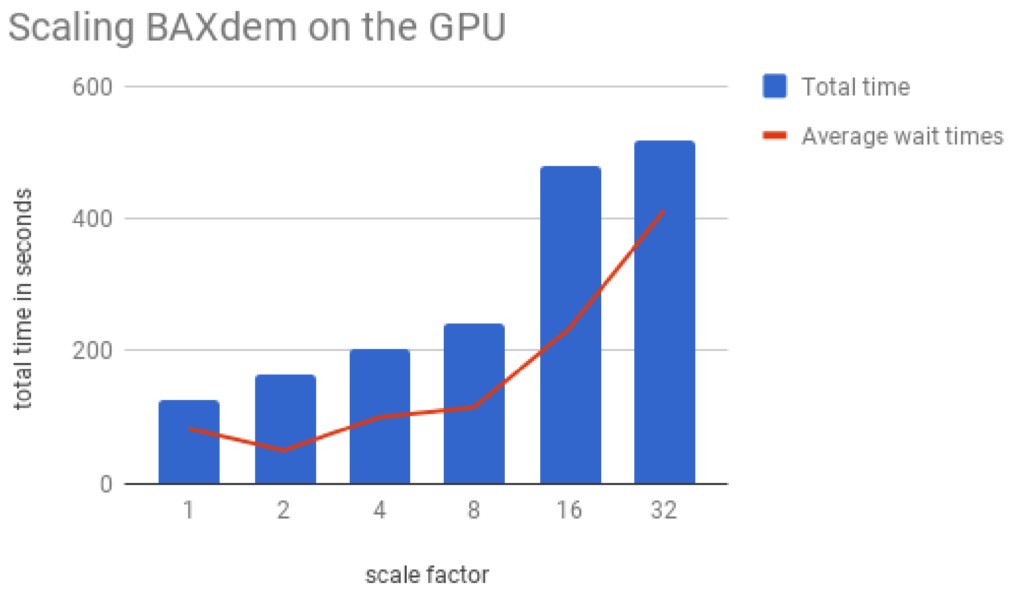

Figure 5.4: Effect of scaling with small batches on the GPU

On a single scale, we observe an average wait time of 83.11 seconds, as opposed to 138.91 seconds with bigger batch size. The waiting time is the lowest for 2 scorers, and relatively similar on scales of 1, 4 and 8. A steep increase is observed in scaled of 16 and 32, directly affecting the total time required.

Thus we observe that small batches are advantageous on a single scorer, but have some overhead and performs worse than big batches on scaled scorers.

The difference in performance for the one scorer over different batch sizes seem to indicate that large batches are helping mostly because they contribute to reduce network-bound message passing. Such effects are not seen for 1 scorer, as the message passing is not network-bound. In one of the next sections we consider with care the overhead from message passing.

5.3.2.3 Comparison of batch sizes

Small batches require a smaller service time than bigger ones, as shown in Table 5.1. The average time tends to be smaller because it processes smaller amount of data. But the number of times the processing needs to happen is much more than bigger batches and therefore do not perform well in general, in comparison. For example, to process a dataset of 100 items with batches of size 5, the average service time required is 0.032 seconds. The total time required would be 0.032 * 20 = 0.640 seconds. For bigger batches of size 50, say the average time taken is 0.208. Therefore the total time required would be 0.208 * 2 = 0.416 seconds, thereby operating faster than small batches.

Scale Factor	**Service Time (small batches)**	**Service Time (big batches)**
1	0.032	0.208
2	0.003	0.074
4	0.004	0.073

Table 5.1: Comparison of average service time (in seconds) for various batch sizes

Big batches for data processing show a growing trend for scaling (blue line in Figure 5.5). The total time required to process data on a single scorers, 2 scorers and 4 scorers is 193.2250 seconds, 58.5366 seconds and 55.9092 seconds respectively.

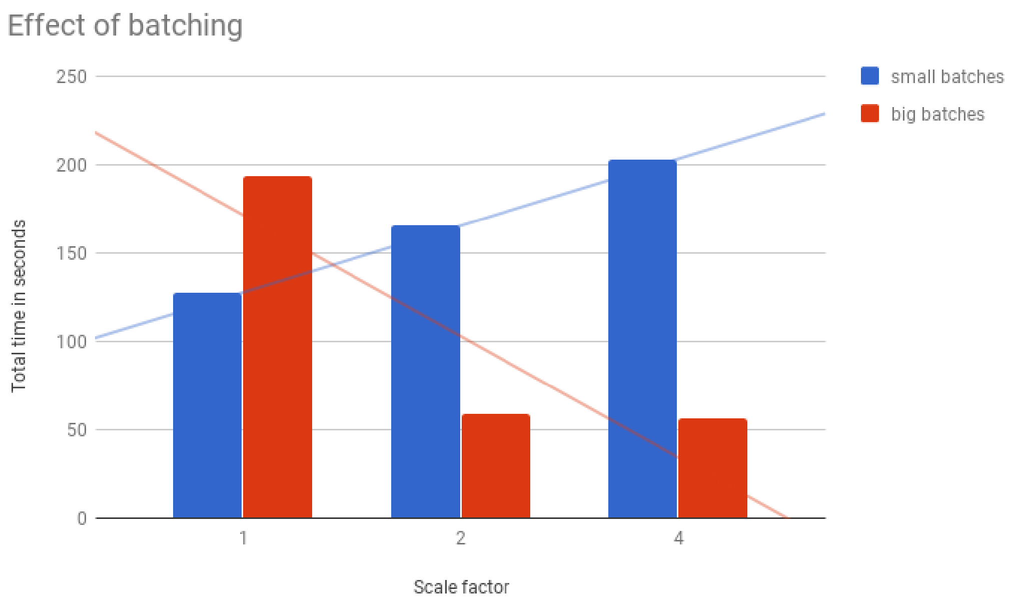

Figure 5.5: Comparison of total time for different batch sizes

On the other hand, the same times for small batches are 127.239 seconds, 165.6402 seconds and 202.8977 seconds. Therefore, it shows a decreasing trend (red line in Figure 5.5).

5.3.3 Comparison Of BAXdem On CPU & GPU

The difference between the total execution times on the CPU and GPU are shown in a graphical format in Figure 5.6.

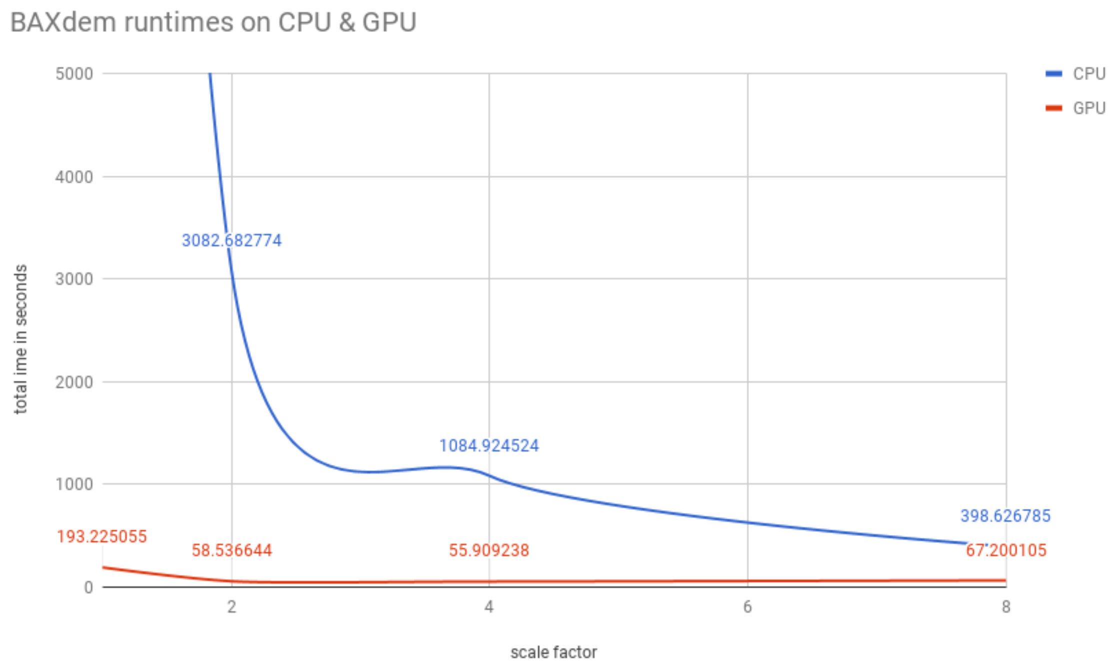

Figure 5.6: Comparison of total execution time on a CPU vs GPU

When tested on 1 scorer, the end to end total time was 19.8 times faster on the GPU already. This shows us the parallelism capabilities and massive potential of hardware acceleration for this particular scientific application.

5.4 Profiling Of Individual Parts

Profiling refers to sampling the state of the system at intervals [4]. This also helps in figuring out which parts of the program have an optimization potential in the future [79]. Python allows us to monitor the time spent on each function in the program by providing statistics at the end of the run, at the cost of some overhead.

We measure the performance of each component of our system to find bottlenecks. We find that majority of the time is spent in the following:

1. In queues, waiting to be processed (Kafka)
2. In range queries from the entire distributed database (Cassandra)

As mentioned several times in this thesis, the non-deterministic behaviour of Kafka and Cassandra proves to be the weakest point of BAXdem. In addition, they are the most time consuming components as well, as clearly shown in Figure 5.7.

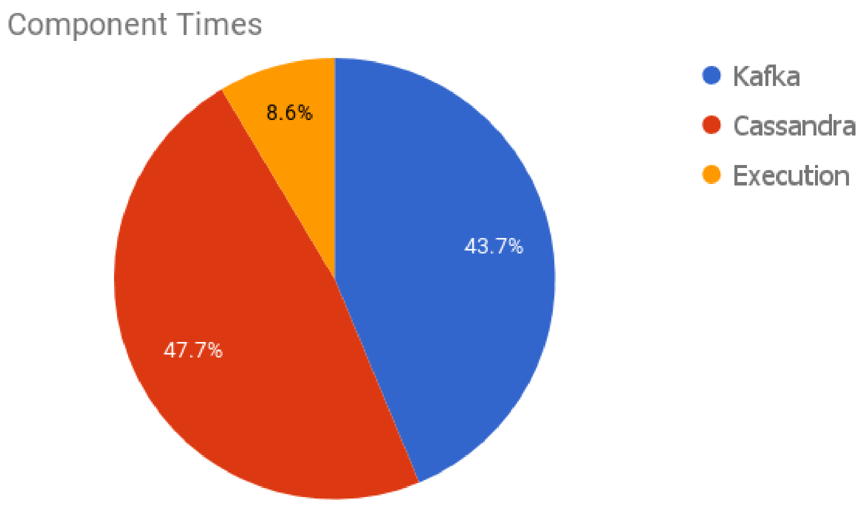

Figure 5.7: Pie chart of various system components and their time required

Cassandra is the most time consuming component in BAXdem. The values in Figure 5.7 are an average of multiple runs, and include all the interactions with the component, from start to end. A major chunk of Cassandra time is from the pair builder, where it scans the theoretical spectra database with several range queries. Cassandra is known to be slower for reads which are not point queries, which makes this behaviour understandable. In addition, scaled-out database access can be affected by network overheads.

Next in line follows Kafka. Messages right from the producer to results collector is passed using Kafka. However, the major chunk of this time comes from all initial message requests. Kafka seems to work smoother as and when the time passes, but the first message of every topic is always the slowest.

Finally the execution of core BAXdem tasks takes the third position in time consumed, but this is very less compared to other components.

From this performance decomposition we can determine that a trade-off between network traffic (increased with more messages/small batches, also with scaling beyond one node), and scale factor needs to be carefully chosen on each run. We can also establish that future work might benefit from targeting first the performance of BAXdem related to Cassandra and Kafka uses (e.g., improving range queries, or perhaps pushing computation to the database) before focusing on improvements to BAXdem's application-side functionality.

5.5 Evaluation Based On a Queueing Theory Model

We evaluate the system's performance based on based on a model that uses queueing theory [4]. This model pertains to the queueing theory (invented by Agner Krarup Erlang), is a mathematical study of queuing systems to analyze the size and latency of queues in a (streaming) system. Our system assumes the architecture as one big queueing system, with a queue per logical process (e.g., scoring, pair-building), as opposed to a network of small queues, in order to evaluate the end-to-end process in terms of queue-load and time efficiency.

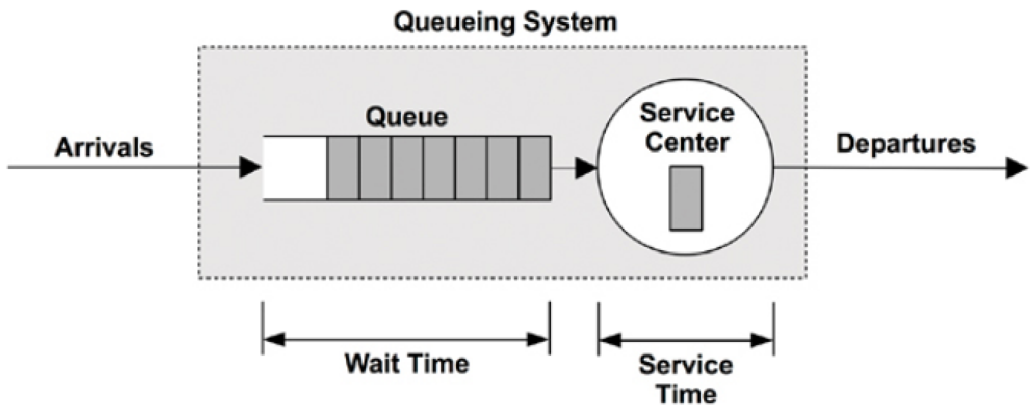

Figure 5.8: A queueing model for performance analysis [4]

Based on Figure 5.8, “Arrivals” are MGF spectra, “Queue” is the queue of Kafka messages for various modules throughout the system, “Service-Center” is the pair-building module as well as the collection of scorers and “Departure” is the score of MGF-FASTA pairs, consumed by the score-collector. Thus we frame BAXdem as a real-time queueing system.

Little’s law is expressed as:

$$L = \lambda W \tag{5.1}$$

where L denotes the average number of pending requests in the system at any time. , λ as the average arrival rate and W as the average service time. For small data irrespective of the scaling factor, our system has a calculated number of requests as

$$L = 102 * 36.09 = 3681.18 \tag{5.2}$$

Little’s law and Kendall’s A/S/m notation (A=arrival rate, S=service time distribution, m=number of service centers) can help us evaluate the effect of load on mean response time, figuring out the number of service centers and distribution of the response times [4] as explained in the following section.

5.5.1 Service Time

Service time is nothing but A-B time [79]. This only highlights the time required for a chosen function to execute. This not only reduces the profiling overhead, but can also identify if pre- and post-processing is more time consuming and appropriate measures can then be taken to reduce this time. On the other hand, if the core function takes more time than others, then the code needs to be more adapted, if optimization is required.

We calculate the service time for calculation of scores. The X-axis shows the scalling factor, and the Y-axis shows the average service time (in seconds). The first chart (a) in Figure 5.9 was created by sorting the service times for small data and observing the time requires. The second chart (b) in Figure 5.9 for big data. The third one (c) shows unsorted service times for small data processing. Similarly the last one (d) shows unsorted service times for big data processing. Figure 5.9(c) and Figure 5.9(d) are log scaled for better readability. They clearly show that there is some startup overhead system. This overhead was reduced with the "kickstarter" script (as described in Chapter 5, a script that runs a small Cassandra query in the background while the system is ready), but not by a very big factor.

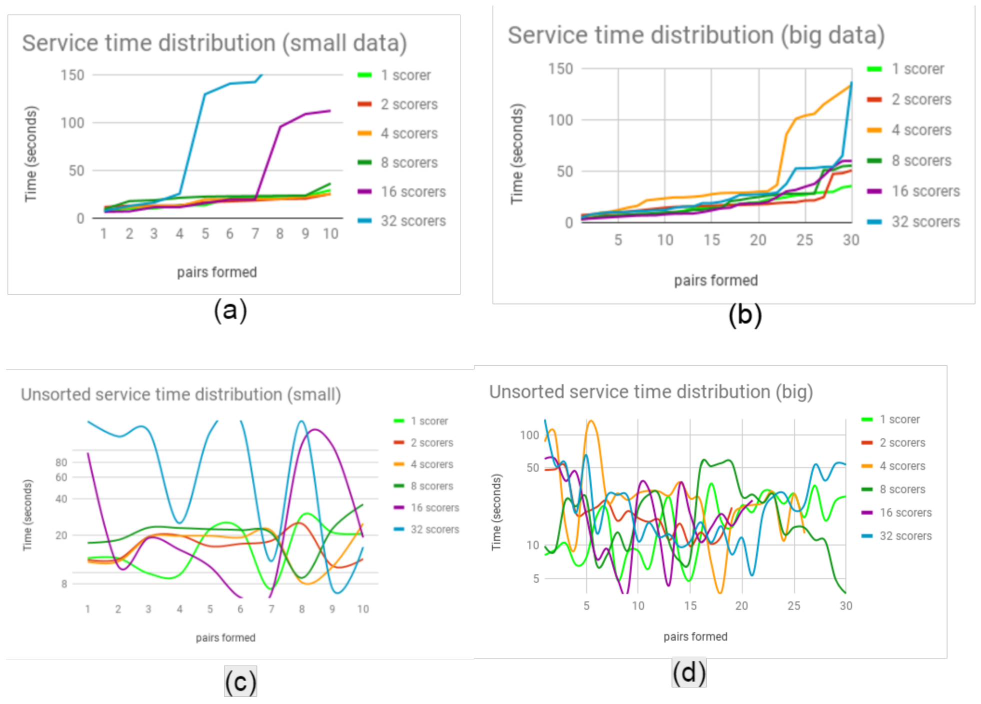

Figure 5.9: Service time plots on the CPU

We run BAXdem on CPU, on a cluster of 3 physical computers and scale the scorers through all the machines. For small data (Figure 5.9(a)), BAXdem running on 16 and 32 scorers appears to be the slowest. This is presumably due to overhead caused by the inequal distribution of data and resources. On the other hand, with big data (Figure 5.9(b)) BAXdem on 4 and 32 nodes requires the most calculation time. Running it on 32 nodes is understandable and similar to the evaluation of small data. However, it seems to perform poorly on 4 nodes.

Thus we see that the service time shows great variability for some cases. Related to this we observe two things: First, that the sharp increases in service time have a big impact on the performance, increasing the wait time for requests in the current queue and the number of active requests. This limits the gains from scaling out. Second, we see that the causes for the changes in service times (as observed in figures c and d), do not follow a temporal pattern (i.e., the increases do not happen at regular intervals, nor can they be explained by start-up overheads), instead the behavior is non-deterministic. This is behavior is notably caused by overheads when making requests to Cassandra for both the pair-building and the score calculation.

We believe that several causes could be responsible for this non-determinism: the fact that the Cassandra requests for pair-building require range searches, an access pattern ill supported by the database (as it includes no range indexes), memory management between the application and the server for requests and results, scheduling issues when dealing with several concurrent requests to the database, and finally network latencies for communication, specially among the internal components in this database.

Looking at the big picture, we see that the behavior of Cassandra plays an unexpectedly important role in the performance of the system. Specifically, the non-deterministic overheads deteriorate significantly the expected benefits from other optimizations in the service time. We believe that finding a solution for guaranteeing acceptable service times when using this system is fundamental for future development in improving the performance of BAXdem and other systems using a similar architecture. One alternative measure that could be considered might be to remove the database interaction from the critical path by either keeping theoretical spectra in an in-memory grid, or employing a database that is a better fit for the query access patterns.

5.5.2 Waiting time

We have established that BAXdem is a real-time queuing system. Since there is some amount of waiting involved in queues. We now measure the waiting time and identify how the waiting time is affected through test changes. Figure 5.10 shows a log scale of the average waiting time across the program and all the runs on a CPU. We observe smooth expected results for small data. The waiting times are calculated as the total time elapsed for a request (i.e. for a given pair, starting from its pair-building up until the score is received in the collector) without considering the service times (i.e. specifically pair-building and score calculation). In contrast to service time, which included Cassandra interactions, the waiting time includes the interactions with Kafka.

The average waiting time decreases for small data, until it reaches 32 nodes after which the overhead increases the waiting time. For big data, results for tests on a scaling factor of 1, 2, 4 and 8 are all high and comparable to each other. The waiting time reduces for 16 nodes and increases slightly for 32 nodes.

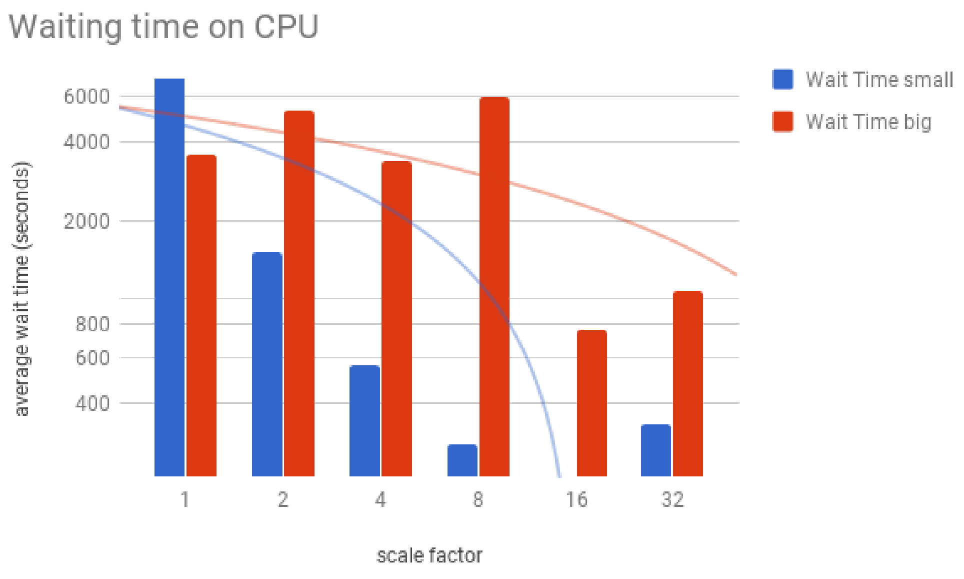

Figure 5.10: Waiting time of data on a CPU

This behavior of big data on 4 and 8 nodes, if not erred due to Kafka, could possibly be due to the large amount of messages passed. It reaches its best potential of resource distribution on 16 nodes and then slightly increases, just like small data. We can see that results for nodes 1 and 2 are heavily skewed due to runtime issues.

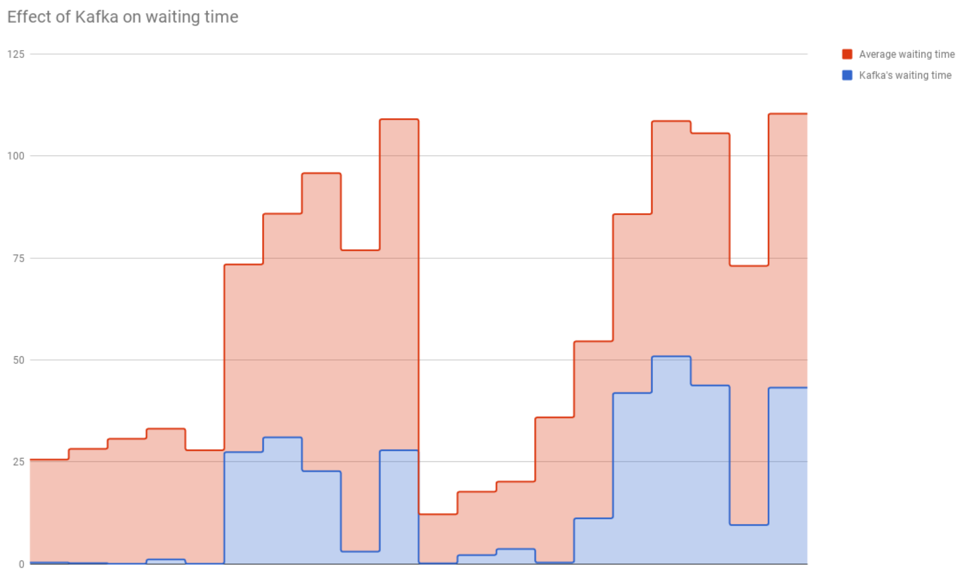

Figure 5.11: Percentage of waiting time required by Kafka

To study the effect of components on the waiting time, some samples were studied. Figure 5.11 has some sample data on the X-axis. The Y-axis shows the amount of time in seconds. The stacking shows the amount of time that Kafka spends in waiting, against the average waiting time. As evident in the chart, it is often the non-deterministic behaviour of Kafka affects the average waiting times.

This suggests that Kafka scales better than Cassandra. Specifically, message passing in Kafka involves establishing consensus on all nodes on message delivery. When the nodes do not consume enough messages, the benefits of the distribution are more limited. When the number of messages grows (e.g., for big data) or the number of consumers grows beyond a point, overheads for message passing and consensus could be observed.

These effects explain the behavior of BAXdem wait times. Another important observation is that the wait times require a much bigger amount of time than the service times, with an important difference of 1 to 3 orders of magnitude (at least 1000s vs 100s of seconds).

5.5.3 Effect of scaling on service and wait times

All the experiments were performed five times, and the mean values were used to plot graphs. The log-scaled wait and service time of the CPU-based system is shown in Figure 5.12. It shows the wait and service times with respect to the total time (Y-axis). Clearly, the performance is poor when the system is not scaled. As the scale factor moves towards 2, the wait time starts to reduce. Both wait and service times drastically improve on scales of 4 and 8. With 16th scale approaching, the wait time still improves, in fact is the best, however the service time begins to increase. Thereafter, overhead is assumed to lower the overall performance. The system at this load performs the best on a scale of 4 and 8, which makes sense as the processors have 4 cores.

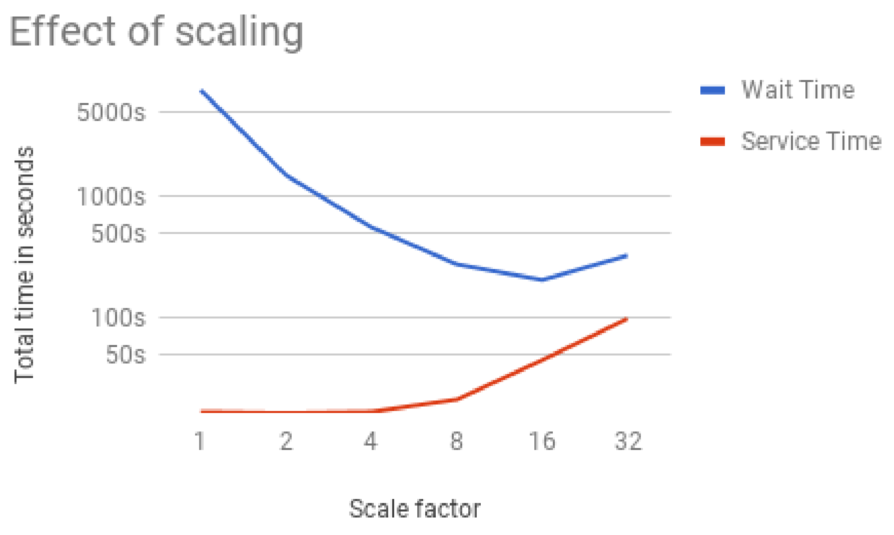

Figure 5.12: Evaluation of small data

It is observed that service time is optimized, until a scale of 8. It is the waiting time, that causes a larger total time. In contrast to wait times, service times increase with growing number of nodes. We ascribe this to the effect to the non-deterministic causes we outlined when discussing service times: limited database support for the query access pattern, memory management, scheduling issues for concurrent requests or network latencies. We recommend future studies in profiling these specific characteristics.

5.6 Concluding Remarks

Thus we evaluate the performance of BAXdem on CPU in terms of response time, scalability, throughput, service and waiting times, answering the questions we established in Section 5.3.

In the next chapter, we will summarize and discuss the performance results. We also propose threats to the validity of our evaluation, hoping that this could inform researchers on issues that need to be considered for continuing our work.

6. DISCUSSION

The tests were successful, and different times were recorded. We now move further, evaluating the system in more detail.

- We start this chapter by summarizing our findings in Section 6.1.
- Next, we propose a simple evaluation of our results in Section 6.2, based on a 5-point qualitative rating.
- We follow by establishing a list of threats to validity in Section 6.3.

6.1 Test Summary

We summarize our results from Chapter 5 as follows:

- Overheads (Section 5.4) and non-deterministic response times (Section 5.5) in the use of our underlying frameworks (Cassandra and Kafka) played a large role in the results of our experimental evaluation. Reducing these aspects needs to be considered for future work.
- Scaling behavior was observed both for CPU and GPU implementations (Section 5.3.1 and Section 5.3.2).
- In the case of CPUs, small data sizes lead to better performance and a predictable scaling behavior, with almost linear increases in throughput when scaling the scorer between 1-16. In contrast, big data sizes showed comparably less increases in throughput, save for the case of 32 scorers, where effects from more parallel Cassandra queries might have played a role in improving the performance. We believe that the difference in scaling behaviors between big and small data was due to the big data having more overheads from message passing and database queries.

- In the case of GPUs, we found first that performance was better than for CPU only implementations, with end-to-end speedups of around 6x to 54x. Part of the gains could be a result of reducing the number of messages through batching. For GPUs we studied scaling behavior for different batch sizes over big data. We find that, save for the outlier case of scale 1, big batch sizes perform better across all scales than small batch sizes. A surprising U- shaped behavior is observed for scaling on big batch sizes, with similar performance on 2-16 scales, and plenty of overheads on 32. We consider that the impact on 32 is related to the scheduling of concurrent kernels on the GPU. In spite of this case, the results for big data are encouraging. They show that GPUs can perform well on large batches until limits of kernel scheduling are reached. On the other hand, small batches display less performance improvements, in part because of message passing overheads, with wait times consistently constituting a large proportion of the total times. In spite of this, there seems to be a reasonable scaling behavior for small batches on GPUs.

- For characterizing the performance of the overall system, we adopted a model based on queueing theory Section 5.5. For this we studied the requests by tracking their service times and wait times. This study was fundamental in understanding the large role that requests with non-deterministic response times play in the overall runtime of BAXdem for our evaluation. Specifically, we established that Cassandra and Kafka operations are part of the overheads. Network communication costs could also be a component in this overhead. This needs to be considered in future work. Another takeaway from the fine-grained studies show that wait times and service times complement their effects as the scale factor grows, with wait time decreasing -in general- with scaling out the scorer, and service times slowly increasing as with the same.

6.2 Performance

In this section we propose a simple rating, for a straight-forward takeaway of our results.

Real-time systems can be categorized as hard and soft [79]. Hard real-time systems are extremely time critical. Their performance is measured in response time, after correctness. If they are not fast enough, they fail in being deployed. For instance, a mission control system or a sensor for accident safety cannot be used if they do not respond to the instructions immediately. On the other hand, soft real-time systems are slightly more tolerant in terms of response time. They can endure slower processing despite being real-time. In other words, they need not be immediate or at zero delay, but just fast enough to suit the application. We foresee BAXdem to be a soft real-time system when integrated with a mass spectrometer, as it should start protein identification as and when a purified protein enters the system. It can tolerate some delays without reaching a critical condition.

By extension of this definition, we propose a 5-point rating to act as an aggregate qualitative performance evaluation measure.

Five point scale performance evaluation

BAXdem can be evaluated on a five point scale with attributes very poor (unacceptable), poor (needs several improvements), average (needs some improvements), good (acceptable) and very good. The performance of a system is supposed to be good if the system responds at a user accepted rate. This criterion is imposed on any real-time system by its environment physically or logically. Once BAXdem was approved by the biotechnology team, we measured its performance. Table 6.1 describes this basic performance evaluation.

Scale Factor	**BAXdem on CPU**	**BAXdem on GPU**
1	Very poor	Poor
2	Poor	Good
4	Average	Very good
8	Good	Good
16	Average	Average
32	Poor	Very poor

Table 6.1: Basic Performance of BAXdem

BAXdem performs best on CPU with a scaling factor of 8. A scaling factor of 4 is acceptable for small data and 16 is acceptable for both small and big data, despite having poor speed. The others scales are either too little or too much for this kind of data and configuration. BAXdem on the GPU works best on a scale of 4. The scale of 1 is acceptable, although all other scale factors are better than it. To process about 10 spectra end-to-end, the CPU requires 19488.45 seconds on average, which is about 5.4 hours. On the GPU, the same amount of data requires about 2.12 minutes to process. Therefore, in comparison to the CPU version, the GPU version's behaviour is notably better.

Now that we've studied the performance of our system we propose a short list of possible threats to the validity of our evaluation using the selected case study and our prototypical implementation, BAXdem.

6.3 Threats To Validity

This section points some factors that could have influenced our evaluation. They are as follows:

- *Possible lack of generalization beyond technology versions, configurations and hardware specifics:*
  As shown in previous sections, our results show some impact from factors external to our core implementation, such as non-deterministic response times for requests and overheads for Kafka and Cassandra, limits in scalability of concurrent GPU kernels, among others. These factors might be very specific to our test configuration and their impact might be different for other component versions, hardware and configurations. From these we believe that network communication could've been a factor that had a big impact in our results and which could be improved in other platforms.

- *Limited representativeness of dataset:*
  We selected a dataset provided by our supporting team from the Chair of Bioprocess Engineering at the University of Magdeburg. Arguably by not using a standard benchmarking solution, our results could be considered to lack in representativeness.

- *Impact of non-determinism:*
  Non-deterministic behavior had an effect in our results. We did not include in our testing any strategy to shield the overall measurements from these effects. We believe that a more thorough statistical analysis of the results, removing outliers could improve the study of BAXdem and the problems that it intends to study.

We now conclude our chapter.

6.4 Concluding Remarks

We have thus evaluated the system to the best of our abilities during the time alloted for our research and development. We will now discuss what work has been done in this field and on the related topics in the following chapter.

7. RELATED WORK

Research shows that some initiatives have already been taken on bridging the gap between HPC and big data. Some of them are discussed in this section. The structure of this chapter is as follows:

- We start with Section 7.1 by explaining the current alternatives for running a big or fast data architecture on modern hardware.
- It is followed by Section 7.2 where we describe the research of databases powered by HPC.
- Finally, Section 7.3 gives additional information about related work on the case study, trying to achieve HPC.

At the end of each section we consider how our research in BAXdem compares to the related work.

7.1 Related Work On HPC And Big Data

1. CUDA on Hadoop[1]
   A CUDA software development Kit is introduced for Java and C/C++developers after Hadoop was tested in a heterogeneous environment [80] and produced interesting results for some workloads. They manipulated the schedulers and hardware accelerators for testing whether running CUDA on Hadoop is possible and if yes, is it cost or time effective. CUDA on Hadoop requires parts of the program to be parallelized in the map phase of map-reduce.

[1]https://wiki.apache.org/hadoop/CUDA%20On20Hadoop

A speedup and power saving of 20x was observed in certain experiments[2]. However, the code is very application specific, only limited to the map phase, requires manual CUDA coding, meaning extensive knowledge of both CUDA and map-reduce. Moreover Hadoop is slightly limited in terms of coding language and speed of execution [10].

2. Kinetica
   Kinetica is a tool that powers itself from GPUs for faster processing of data on a distributed in-memory database and claims to be 10 to 100 times faster than several in-memory database systems[3], given that it runs on millions of GPU cores. It provides Kafka, Storm, NiFi, Kibana and Spark connectors and can be used with AWS.

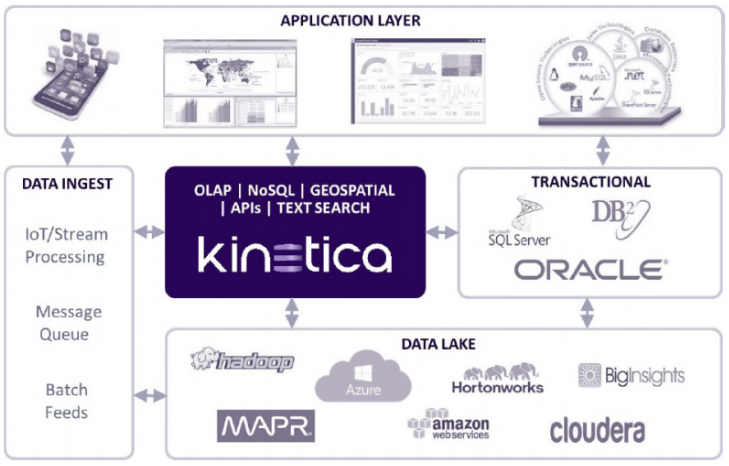

Figure 7.1: Architectural framework with Kinetica [a]

[a]Kinetica:, "Maximizing Data Analytics Price/Performance with GPU acceleration", https://www.kinetica.com/resources/

The architecture of a big data's framework using Kinetica as shown in Figure 7.1 provides high availability and processing speed. It is similar to any other database in operation, but differs in its storage and processing techniques. Kinetica places business and processing logic in a single database. It interacts with data sources (streams, queues, buffers), persists the data (DB2, Oracle,SQL server), processes it (Hadoop, AWS, Azure) and provides data to the application layer (visualization), therefore acting as a speed layer.

However, the downside is that Kinetica is not open source.

3. Spark extensions
   Spark users have the following possibilities of choosing a big data integrated HPC environment:

[2]https://www.slideshare.net/airbots/cuda-29330283
[3]https://www.kinetica.com/docs/

(a) IBM GPU Enabler: GPU and Spark
IBMs GPU Enabler [4] brings GPU awareness to the Spark environment. It runs GPU kernels on Nvidia GPUs cards, and can convert data into column-based format for inputting data into the kernels. It is provided in form of a Spark library, and added as a dependency. It provides Java (only RDDs) and Scala APIs which can be used to run a Spark job on the GPU.

However, the downside is that it is fine-tuned for Spark only and requires Nvidia GPU card with CUDA v7.0+. It is still not completely mature at the moment. It supports only basic matrix operations. For unsupported operations users are encouraged to write their own CUDA codes, compiling it to intermediate results and using it from Spark. Limited guides are offered for this process.

(b) SparkCL
This was an initiative to have compatibility between Spark clusters and special cores like FPGA, GPU, APU [81]. It is implemented using the Java OpenCL framework. It works as follows:
Kernel code for the GPUs is written in Java and passed to the SparkCL module and then passed to the Spark program and the Aparapi framework[5].

Thus HPC can be leveraged using Spark extensions.

4. MARS
GPU-Accelerated Cloud Computing for Data-Intensive Applications [5] implement map-reduce on a GPU cluster. It has an API similar to map-reduce, except that map and reduce phases work as three phases: map, group and reduce.

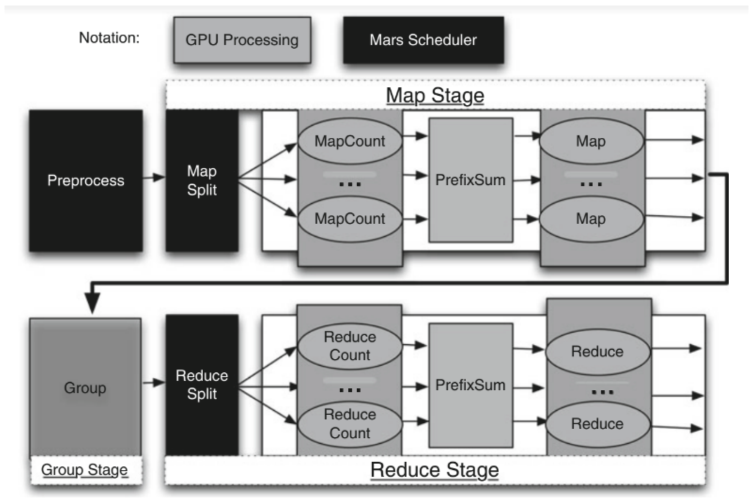

Figure 7.2: Workflow of Mars (Hadoop) on a GPU system [5][6]

[4]https://github.com/IBMSparkGPU/GPUEnabler
[5]Aparapi is a framework which executes native Java code on the GPU https://aparapi.com

As opposed to only one phase being parallelized on GPUs [80], it is optimized for all three phases to have CUDA code underneath. Experiments show that it can improve the speed. It still has a large optimization space, since operations such as hash or sort are not reported to be optimized.

5. Heron
   Apache Heron is a stream processing framework, similar to Apache Storm, motivated by streaming trends and lack of use of HPC interconnects in big data frameworks [7]. Instead of relying on HPC as a processor, it is based on HPC provided by the connection fabric, namely Infiniband [82] and Intel's Omni-Path [83]. It has the following components and can be seen in Figure 7.3:

   (a) A scheduler is responsible for allocating multiple containers which may be physical nodes or virtual machines.
   (b) A master container which is the head manager of the topology (a graph representation for logic and data flow).
   (c) Heron instances which manage the data source and processing logic.
   (d) A stream manager to handle data routing to and from the system.
   (e) A metrics manager which is like a statistics collector that monitors all the containers in the topology.

   All stream managers in the topology are connected by HPC interconnects to ensure efficiency and speed of messages passed between them. As the communication is more frequent between stream managers, than between stream managers and master containers, the latter does not require an HPC enhancement, but only use a normal TCP connection.

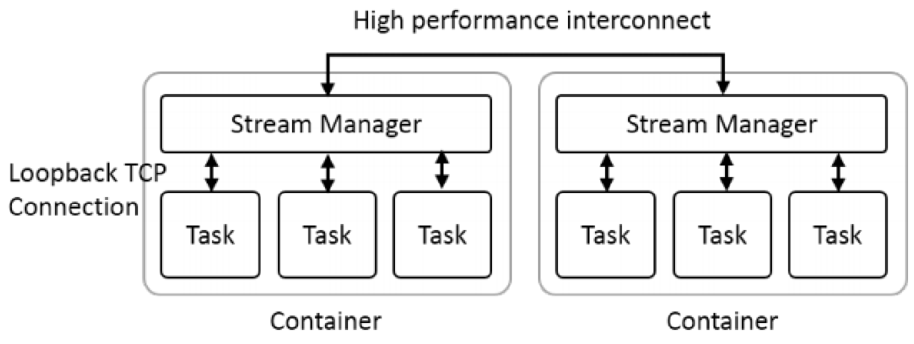

Figure 7.3: Architectural framework of Heron [7]

Experiments performed using this architecture showed that using HPC interconnects yielded more throughput and decreased the latency as compared to a normal Ethernet connection.

From the industrial and academic systems we found that considering bridges between HPC and big data, we observe a distinction between those that optimize for execution over heterogeneous devices, and those that focus on improving the networking subsystem. BAXdem is similar to items in the first category. Unlike most systems BAXdem does not consider an established big data framework like Spark or Hadoop, instead we focused on exploiting a message bus framework like Kafka. This allows the observations in BAXdem to be independent from code optimizations provided by query optimizers in systems like Spark, and independent of the scheduling configurations of a framework. Through its design choices, BAXdem allows a clear evaluation of how specific components of a big data framework, like the streaming and persistence services, interact on the architecture with GPU acceleration. We believe that future work in BAXdem focusing on these specific components could be beneficial.

Similar to the work in CUDA on Hadoop, Mars and the Spark extensions, BAXdem provides GPU acceleration for jobs and not for underlying processes. Unlike the reported work, BAXdem adopts OpenCL.

There is a large body of work in providing GPU acceleration of diverse fine-grained data processing operations. Libraries like CUBLAS and cuDNN are example of them, providing accelerated operators for linear algebra and neural network training, respectively. The work in BAXdem shares with them a focus on adopting GPU acceleration through operators which could be standardized. In our current version of BAXdem we only provide one operator for batch-wise similarity scoring of proteins. Further standardization would be necessary for BAXdem to offer fully fleshed library for GPU acceleration of protein identification operations. The current focus of BAXdem was only on how these operators interact with the entire framework.

Work considering HPC for the networking layer could be beneficial for BAXdem, since this layer could be partly responsible for the non-deterministic response times in using the database and messaging systems in BAXdem.

7.2 Related Work On HPC Powered Databases

For a complete HPC powered big data system, we need databases to be both OLTP (OnLine Transactional Processing) or OLAP (OnLine Analytical Processing) optimized, depending on the project. Research suggests that current CPU and GPU systems are not yet fit for HTAP (Hybrid Transactional/Analytical Processing) [84]. However a lot of research has happened in using GPU accelerated databases and some of the examples are mentioned in this section.

1. GPUTx:
   GPUTx is a GPU based database optimized for OLTP queries [41]. It groups transactions together, and forms a “bulk”. Each transaction belongs to a certain

transaction type and is an instance of it. They are programmed as a kernel and parallelly executed on the GPU. This proves to be efficient as it avoids user interaction, improves utilization by grouping transactions as compared to sequential execution of transactions and are also easier to manage as they are programmed in one kernel. Although it has its own limitations, GPUTx proved that GPUs can definitely influence the speed of an OLTP database.

2. CoGaDB:
   CoGaDB is a CPU / GPU database system, specially designed for analytic processing [85]. The specialty of this database is its self-adaptive, scalable query optimizer called HYPE. It is responsible for learning and adapting the cost model as well as load balancing on the involved devices.

   This database handles the bottleneck to and from the device memory using distributed data locality, thereby allowing fragments of database records to exist in both memories, or in either one. The central GPU buffer manager handles the data flow and data placement. Since GPU memory is limited and CoGaDB allocates memory as needed, it frees GPU space by removing cache columns. The query processor benefits from the GPUs by parallelizing the operations and increasing the speed and efficiency of the system. It also enhances the join operations as joins are the most time-consuming operations compared to others.

3. MegaKV:
   MegaKV is a GPU accelerated database for big data applications [86]. It relies on its in-memory key-value properties and optimizes it for processing large volumes of data. MegaKV reduces the load on CPU by using GPUs for indexing purposes.

   The algorithm works in three phases: Receive, Schedule and Send. In the first phase, jobs are received, separated on the basis of their operation (search, insert, delete) into different buffers. Schedule launches GPU kernels at fixed time intervals. At the end of a certain interval, the queried data accumulated until then is processed. The throughput is enhanced by leveraging GPUs parallelism properties for cuckoo hash [87] function, which has fixed size buckets, and each key can have multiple locations for faster access. It deals with the memory size and access latency constraints by using two hash functions and a large bucket size. Finally the MegaKV sender looks up key-value references using the indices and sends responses back.

   Thus, MegaKV uses GPUs to make itself an efficient in-memory key-value database.

4. Blazegraph:
   This is a graph based database [88] mainly used for IOT(Internet of things) and data discovery. It directly translates SPARQL queries and transparently makes them GPU compatible with a functional, domain specific language DASL. DASL handles the conversion from Apache Spark or Scala to CUDA, thereby alleviating the pressure of GPU programming from the developers.

In contrast to research in HPC databases, BAXdem is focused on large scale applications where the data access is not required to follow relational semantics. Considerations like transactions and buffer management seem are outside the scope of BAXdem.

Similar to the query processing in most HPC databases, BAXdem required to consider batching of operations in order to gain benefits from GPU processing. Within the architecture of BAXdem batching has also the benefit of reducing the number of Kafka messages, thus reducing wait times and improving the use of the messaging component of our big data architecture. Based on these facts, research in batching could be informative for future work in BAXdem. Batching in the database could also influence the database overheads we observe, thus they could be worthwhile to consider as well.

A GPU accelerated database capable of good support for range queries (which was a limitation observed for Cassandra) could be a good fit for using with BAXdem.

7.3 Related Work On The Case Study

Researchers and developers have come up with variations of the X!Tandem algorithm to make them compliant with big data frameworks and/or HPC systems.

1. X!!Tandem
   X!!Tandem [72] uses the threaded model of X!Tandem and spreads it across a network, thereby distributing the search task. It passes search results via ssh and works asynchronously. It relies on MPI (Message Passing Interface) framework, making it slightly complicated and prone to possible latency and high failure chances. This is so because with MPI, any node could be a single point of failure, and so can a network delay. However, it is fast and produces the same results as X!Tandem and is efficient that way.

2. Parallel Tandem
   Parallel Tandem [89] divides a search task into smaller parts and then distributes them across a Linux cluster with a parallel computing environment. A large spectra file is equally divided into smaller parts and parallelly matched against protein databases using PVM or MPI frameworks. Finally results from all nodes are collected and aggregated. Results show it to be 90% scalable, meaning that 10 processors make it 9% faster.

3. MR-Tandem
   MR-Tandem [90] adapts X!Tandem to get it to work on Hadoop's map reduce using Python scripts and AWS. It is very similar to X!!Tandem in terms of its concept, except MPI is replaced by Hadoop. MR-Tandem starts the Hadoop cluster, transfers the data to the cloud (Amazon S3), performs the search on the distributed cloud cluster and sends the results back to the local calling machine. As expected, Hadoop induced latency (disk I/O and wait for successful acknowledgements) and made the process about 20% slower than X!!Tandem. However, they did succeed in making the system more fault tolerant.

X!Tandem tackles the problem of its incapability to cope up with fast data generation by running a variant of the algorithm on GPUs, but only for a single machine. This comes at a cost of non-portability and is thus less flexible.

Thus, research has been done to parallelize metaproteomics algorithms such as X!Tandem, and to run it on Hadoop clusters. Unlike X!!Tandem, BAXdem assumes a scalable message passing system like Kafka instead of point-to-point communication. Unlike MR-Tandem BAXdem selects a different big data framework, removing from the path query optimizations and allowing our study to focus on the big data components.

BAXdem is most similar to parallel tandem, but is distinguished from it by using big data tools and GPU acceleration.

Finally, the GPU work in X!Tandem is related to BAXdem but is different since it was not done while using scale-out techniques.

Furthermore, all these tools offer complete end-to-end support for the protein identification process, in contrast our work in BAXdem approaches this task as a case study for evaluating, on a real-world scenario, how GPUs integrate with big data frameworks and what should be considered for improving such integration.

7.4 Concluding Remarks

In this chapter we covered related work to BAXdem, discussing specifically research on using HPC with big data frameworks, HPC powered databases and parallel versions of X!Tandem. For each case we discussed how our work in BAXdem compares to the reviewed research.

In the next chapter we conclude the thesis by summarizing our findings and suggesting future research directions.

8. CONCLUSION

In this chapter we provide a summary of our work and findings. We conclude by discussing possible future directions in our research.

8.1 Summary

In this thesis, an overview of concepts, examples, advantages and challenges of HPC and fast/big data is provided. The use of these technologies in a streaming system is also discussed.

Chapter 2 explains that scientific applications often have to choose between big/fast data applications and HPC, and, mentions the challenges encountered if both technologies need to be leveraged. As the first accomplishment, this thesis provides a proof-of-concept through a prototype implementation, that protein identification algorithms can be run on a hardware accelerated platform capable of handling big or fast data. Details of the case study are given in Chapter 3. Secondly, to support the research questions, a prototype called BAXdem (introduced in Chapter 4) is implemented on the CPU and GPU, on different scales, different datasets and different batch sizes.

It is a fast data framework made up of Python, Cassandra and Kafka; with parallel programming for data- and calculation-intensive tasks. This framework is compatible with GPUs, and hence is an HPC powered fast data framework. To leverage HPC, we use openCL as a Python library. For that, we create a context and a program (with accelerated/parallel code) in it. Then, we create a kernel int he program, which performs the operations for each selected pair of theoretical and experimental spectra, and build it. Internally, it creates threads that run simultaneously, and each comparison is concurrently performed by different threads. Finally, the output of the kernel is copied to the host memory. In contrast, this happens sequentially on the CPU version, with the number of comparisons ranging from 75 to 6000.

The tests in Chapter 5 and its discussion in Chapter 6 allows us to answer the research questions from Chapter 1 as follows:

1. Does an HPC-based fast data architecture bring benefits to a streaming system? Or, is it worthwhile for scientific applications to move from high performance computing to big data frameworks if the gap between the two technologies is reduced?

    - We see that optimizations for parallel processing yield good results. To be precise, a speed up of a factor of 53 is observed on a GPU. However, the non-deterministic behavior of network components like Kafka and Cassandra undermine the benefits, by slowing it down. We see in Section 5.4, that Kafka takes nearly 43.7% of the total time, and Cassandra requires about 47.7%. Other calculations, which were the focus of this thesis, were optimized and require only 8.6% of the total time, on an average. Different configurations of Kafka and Cassandra were tested, and, Cassandra was scanned in parallel using 3 physical nodes. Nonetheless, the results did not improve much.
    - It is also noteworthy, that our implementation can, with minimal enhancements, be moved to a cloud computing environment. More importantly, in doing so, we don't lose the advantages of HPC, such as GPU processing. This further strengthens the support of our research question, that an HPC-based fast data architecture can bring benefits such as cloud compatibility, to a streaming system.
    - In conclusion, we can say that this thesis proves that it can be worthwhile for scientific applications to move from HPC to big data frameworks. However, it is more beneficial after the right alternatives, or, optimizations for the current configurations of the supporting components (in our case Kafka and Cassandra are chosen). This conclusion can be complemented with the takeaway that speeding-up supporting components can be expected to have significant impact in speeding-up applications built over them. Thus we strongly believe that hardware-acceleration for big data is a holistic challenge requiring strategic collaboration between large scale processing frameworks, distributed databases and applications.

2. Which technique among scaling out and scaling up is more powerful for hardware accelerated big data systems?

    - We see that the CPU version of the prototype is the fastest with a scale of 8 on a 4-core processor, whereas the GPU version is the fastest on a scale of 4. However scaling up the system (i.e., adopting GPUs), provided a higher gain than scaling out. A slightly lesser but still interesting gain can be seen when we scale out the system, especially with the first scale, which is scale factor of two.

- In conclusion, we can say that for this application, scaling up was more beneficial than scaling out. As a result, our experience and study yield an interesting finding: hardware-acceleration can win over scaling-out for scientific applications, and without significant efforts in its adoption.

Thus, this thesis analyses the current streaming trends and proves that such a system can benefit from hardware as well as software. Finally, an overview of related research was provided in Chapter 7. It describes the past and current work in bridging the gap between HPC and big data, using HPC to optimize databases and work on the case study benefiting from modern hardware. In the next section, we describe some possibilities that can enhance this research.

8.2 Future Work

BAXdem is a confluence of big data tools and hardware acceleration. It was developed to to support the scaling-out and up (exploiting hardware) of the specific use case of protein identification. Though the use case was specific, BAXdem allowed us to evaluate practical limitations and opportunities that developers might face when attempting to combine HPC and big data. As a result, future work in BAXdem is mix of research opportunities in both the specific use case, and in the general goals of combining hardware acceleration with big data. We categorize potential improvements on BAXdem as follows:

- Improvements on current version of BAXdem:
  1. A lot of research has been done on database applications [27][28][25][26]. BAXdem can run faster by moving the scorer logic to the database. This way we retrieve lesser data from Cassandra and thereby speed up the process. 2. Section 5.4 shows Kafka and Cassandra slow the system down. Alternatives for these technologies can be researched. As BAXdem is a general design, other tools can be used, exploring alternatives in the development space. Some interesting alternatives could be the use of in-memory grids and distributed file systems in contrast to single-system databases, or the use of a large-scale relational database, with reasonably good support for range queries, like CockroachDB or NuoDB.

- Improvements for a next version of BAXdem:
  BAXdem could potentially become BRAXdem (Big Real-time Accelerated X!tandem) when incorporated into a mass spectrometer. That way, it will be a truly real-time fast data system accelerated by modern hardware.
  BAXdem as a library for accelerated protein identification:
  Section 4.1.3 mentioned that for BAXdem we wrote operators as openCL code, to perform higher-granularity function that were not provided by Tensorflow. The developed operator could be imported into Tensorflow as an extension to make it reusable for other users who would like to have the same functionality. Further operations could be implemented and standardized, such that BAXdem

contributes to the acceleration of protein identification tasks and the development of a hardware-accelerated library for this domain.
Extending protein identification functionality:
For a real use in protein identification, BAXdem needs to be extended with more functionality. For example, as mentioned, X!Tandem is able to reduce the number of pairs considered through filtering criteria applied to the theoretical spectra. This is not entirely adopted in BAXdem, making our prototype perform much more comparisons than X!Tandem. One grand goal could be for BAXdem to grow such that it can become a fully-fledged system comparable in number of features to X!Tandem.

- Further improvements related to the general HPC-Big Data concerns:
  Study of gains from HPC in the network system:
  Research on tools like Heron has found good performance in big data frameworks by leveraging HPC for the networking system. Future work in BAXdem could explore this potential. We believe that possible overheads from network communication affecting our results would be evened out with similar tools.
  Research in further enhancing Big Data supporting components with HPC:
  From our work we find that expected gains from scaling-out applications can be toppled by hardware-acceleration with little effort. However we also observe that the supporting components (large scale processing frameworks and distributed databases) can add overheads and limit the gains from hardware-acceleration. As a result we believe that more effort is needed in adding hardware-acceleration to distributed databases and to large-scale processing frameworks. This should be done in a holistic manner that does not impede the possibility of also adding acceleration at the application level.

- Further evaluation:
  In BAXdem we report that the effects of non-deterministic behavior from Kafka and Cassandra influence the performance on the tests we carried out. Future work needs to consider this carefully, first, detecting the precise causes of the behavior (e.g. scheduling, networking, memory management) and second, including improvements into BAXdem. This would be a very valuable follow-up from our research.

Thus, we conclude the thesis by summarizing our work and findings, and, by providing directions to possible future work.

Bibliography

[1] H. Anzt, J. Dongarra, M. Gates, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, *Handbook of Big Data Technologies.* Springer International Publishing, 2017. (cited on Page xiii, 3, 16, and 17)

[2] E. Hewitt and J. Carpenter, *Cassandra-The Definitive Guide: Distributed Data at Web Scale. Definitive Guide Series.* O'Reilly, 2 ed., 2016. (cited on Page xiii, 32, and 59)

[3] S. Banerjee and S. Mazumdar, "Electrospray ionization mass spectrometry: A technique to access the information beyond the molecular weight of the analyte," *International Journal of Analytical Chemistry*, vol. 2012, 2012. (cited on Page xiii and 41)

[4] B. Gregg, *Systems Performance: Enterprise and the Cloud.* Upper Saddle River, NJ, USA: Prentice Hall Press, 1 ed., 2013. (cited on Page xiv, 69, 74, 75, and 76)

[5] B. Zhao, J. Zhong, B. He, Q. Luo, W. Fang, and N. K. Govindaraju, "Gpu-accelerated cloud computing for data-intensive applications," in *Cloud Computing for Data-Intensive Applications*, pp. 105–129, Springer, 2014. (cited on Page xiv and 89)

[6] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 260–269, ACM, 2008. (cited on Page xiv and 89)

[7] S. Kamburugamuve, K. Ramasamy, M. Swany, and G. rey Fox, "Low latency stream processing: Apache heron with infiniband & intel omni-path," 2017. (cited on Page xiv and 90)

[8] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO white paper*, vol. 1, no. 2011, pp. 1–11, 2011. (cited on Page 1)

[9] A. Kipf, V. Pandey, J. Boettcher, L. Braun, T. Neumann, and A. Kemper, "Analytics on fast data: Main-memory database systems versus modern streaming systems," 2017. (cited on Page 1 and 10)

[10] D. Wampler, “Fast data: Big data evolved,” *Lightbend*, 2015. (cited on Page 1, 2, 3, 14, and 88)

[11] Y. Ahmad and U. Çetintemel, “Streaming applications,” *Encyclopedia of Database Systems*, pp. 2847–2848, 2009. (cited on Page 1, 2, and 10)

[12] M. Kleppmann, *Making Sense of Stream Processing.* O’Reilly Media, Inc, 1 ed., 2016. (cited on Page 1)

[13] E. Curry, “The big data value chain: Definitions, concepts, and theoretical approaches,” *New Horizons for a Data-Driven Economy: A Roadmap for Usage and Exploitation of Big Data in Europe*, pp. 29–37, 2016. (cited on Page 1)

[14] Z. Zheng, P. Wang, J. Liu, and S. Sun, “Real-time big data processing framework: challenges and solutions,” *Applied Mathematics & Information Sciences*, vol. 9, no. 6, 2015. (cited on Page 2)

[15] M. Stonebraker, U. Çetintemel, and S. Zdonik, “The 8 requirements of real-time stream processing,” *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005. (cited on Page 2)

[16] R. Ledyayev and H. Richter, “High performance computing in a cloud using openstack,” *CLOUD COMPUTING 2014 : The Fifth International Conference on Cloud Computing, GRIDs, and Virtualization*, 2014. (cited on Page 3)

[17] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, “The rise of “big data” on cloud computing: Review and open research issues,” *Information Systems*, vol. 47, no. Supplement C, pp. 98 – 115, 2015. (cited on Page 3)

[18] C. Vecchiola, S. Pandey, and R. Buyya, “High-performance cloud computing: A view of scientific applications,” *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pp. 4–16, 2009. (cited on Page 3)

[19] R. Heyer, F. Kohrs, U. Reichl, and D. Benndorf, “Metaproteomics of complex microbial communities in biogas plants,” *Microbial Technology*, vol. 8, 04 2015. (cited on Page 4, 5, and 41)

[20] A. Schlüter, T. Bekel, N. N. Diaz, M. Dondrup, R. Eichenlaub, K.-H. Gartemann, I. Krahn, L. Krause, H. Krömeke, O. Kruse, *et al.*, “The metagenome of a biogas-producing microbial community of a production-scale biogas plant fermenter analysed by the 454-pyrosequencing technology,” *Journal of biotechnology*, vol. 136, no. 1, pp. 77–90, 2008. (cited on Page 5)

[21] R. Heyer, K. Schallert, R. Zoun, B. Becher, G. Saake, and D. Benndorf, “Challenges and perspectives of metaproteomic data analysis,” *Journal of Biotechnology*, vol. 261, no. Supplement C, pp. 24 – 36, 2017. (cited on Page 5 and 38)

[22] R. Craig and R. C. Beavis, "A method for reducing the time required to match protein sequences with tandem mass spectra," *Rapid Communications in Mass Spectrometry*, vol. 17, no. 20, pp. 2310–2316, 2003. (cited on Page 5, 48, 49, 55, and 56)

[23] J. K. Eng, A. L. McCormack, and J. R. Yates, "An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database," *Journal of the American Society for Mass Spectrometry*, vol. 5, no. 11, pp. 976–989, 1994. (cited on Page 5 and 48)

[24] J. S. Cottrell and U. London, "Probability-based protein identification by searching sequence databases using mass spectrometry data," *electrophoresis*, vol. 20, no. 18, pp. 3551–3567, 1999. (cited on Page 5 and 48)

[25] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012. (cited on Page 6 and 97)

[26] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann, "Sql-and operator-centric data analytics in relational main-memory databases.," pp. 84–95, 2017. (cited on Page 6 and 97)

[27] I. Arefyeva, M. Bhatnagar, D. Broneske, M. Pinnecke, and G. Saake, "Column vs. row stores for data manipulation in hardware oblivious cpu/gpu database systems," *29th GI-workshop on Foundation of Databases*. (cited on Page 6 and 97)

[28] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, "Gpu-accelerated database systems: Survey and open challenges," *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV*, 2014. (cited on Page 6 and 97)

[29] J. Kreps, N. Narkhede, and J. Rao, "Kafka: a distributed messaging system for log processing," *Proceedings of 6th International Workshop on Networking Meets Databases(NetDB)*, 2011. (cited on Page 11, 16, 29, and 30)

[30] M. Saecker and V. Markl, *Big Data Analytics on Modern Hardware Architectures: A Technology Survey.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. (cited on Page 17, 18, 20, and 21)

[31] D. J. DeWitt, "Direct - a multiprocessor organization for supporting relational data base management systems," *Proceedings of the 5th Annual Symposium on Computer Architecture*, 1978. (cited on Page 17)

[32] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays.* Springer-Verlag Berlin Heidelberg, 2007. (cited on Page 18)

[33] J. Nikolic, J. Rehder, M. Burri, P. Gohl, S. S. Leutenegger, P. T. Furgale, and R. Siegwart, "A synchronized visual-inertial sensor system with fpga pre-processing

for accurate real-time slam," *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 431–437, May 2014. (cited on Page 18)

[34] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 25–28, May 2014. (cited on Page 18)

[35] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, 2015. (cited on Page 18)

[36] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W.-m. Hwu, *Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)*. Morgan & Claypool Publishers, 2012. (cited on Page 18, 19, 23, 25, and 26)

[37] R. Rui and Y.-C. Tu, "Fast equi-join algorithms on gpus: Design and implementation," *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, 2017. (cited on Page 19)

[38] M. Yabuta, A. Nguyen, K. Shinpei, M. Edahiro, and H. Kawashima, "Relational joins on gpus: A closer look," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2663–2673, 2017. (cited on Page 19)

[39] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, pp. 777–786, Aug. 2004. (cited on Page 19)

[40] A. Meister, S. Breß, and G. Saake, "Toward gpu-accelerated database optimization," *Datenbank-Spektrum*, vol. 15, pp. 131–140, Jul 2015. (cited on Page 19)

[41] B. He and J. X. Yu, "High-throughput transaction executions on graphics processors," *Proceedings of the VLDB Endowment*, vol. 4, no. 5, pp. 314–325, 2011. (cited on Page 19 and 91)

[42] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pp. 215–226, 2004. (cited on Page 19)

[43] N. Brookwood, "Amd fusion^{TM} family of apus: Enabling a superior, immersive pc experience," 2010. (cited on Page 20)

[44] A. Duran and M. Klemm, "The intel many integrated core architecture," *2012 International Conference on High Performance Computing Simulation (HPCS)*, pp. 365–366, July 2012. (cited on Page 21)

[45] A. Heinecke, M. Klemm, and H.-J. Bungartz, "From gpgpu to many-core: Nvidia fermi and intel many integrated core architecture," March 2012. (cited on Page 21)

[46] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, vol. 28, no. 2, 2008. (cited on Page 23 and 26)

[47] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010. (cited on Page 25)

[48] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011. (cited on Page 25)

[49] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *Computing Research Repository - CORR*, vol. arXiv preprint arXiv:1005.2581, 05 2010. (cited on Page 25)

[50] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990. (cited on Page 28)

[51] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, "Trends in big data analytics," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2561–2573, 2014. (cited on Page 28)

[52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica, "Fast and interactive analytics over hadoop data with spark," *Usenix Login*, vol. 37, no. 4, pp. 45–51, 2012. (cited on Page 28)

[53] J. M. Pampliega, "Towards an architecture for real-time event processing," 2016. (cited on Page 28)

[54] D. Zhu, "Large scale etl design, optimization and implementation based on spark and aws platform," 2017. (cited on Page 29)

[55] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide*. O'Reilly Media, Inc, 2016. (cited on Page 29 and 30)

[56] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, Apr. 2009. (cited on Page 31 and 32)

[57] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983. (cited on Page 31)

[58] D. Pritchett, "Base: An acid alternative," *Queue*, vol. 6, no. 3, pp. 48–55, 2008. (cited on Page 31)

[59] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, pp. 654–663, 1997. (cited on Page 32)

[60] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014. (cited on Page 32 and 33)

[61] M. Hailperin, "Load balancing for massively-parallel soft real-time systems," *Proceedings of 2nd Symposium on the Frontiers of Massively Parallel Computation*, pp. 159–163, 10 1988. (cited on Page 32)

[62] M. D. Da Silva and H. L. Tavares, *Redis Essentials.* Packt Publishing Ltd, 2015. (cited on Page 33)

[63] P. Membrey, E. Plugge, and D. Hawkins, *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing.* Apress, 2011. (cited on Page 34)

[64] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016. (cited on Page 35)

[65] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, pp. 1–12, June 2017. (cited on Page 35)

[66] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012. (cited on Page 36)

[67] M. Wilkins, R. Appel, K. Williams, and D. Hochstrasser, *Proteome Research: Concepts, Technology and Application.* Berlin, Heidelberg, New York: Springer, 2 ed., 2005. (cited on Page 38, 39, and 40)

[68] D. Whitford, *Proteins: Structure and Function.* John Wiley and Sons Inc., 2 ed., 2005. (cited on Page 38, 39, and 40)

[69] J. Rodriguez, N. Gupta, R. D. Smith, and P. A. Pevzner, "Does trypsin cut before proline?," *Journal of proteome research*, vol. 7, no. 01, pp. 300–305, 2007. (cited on Page 41)

[70] E. W. Deutsch, "File formats commonly used in mass spectrometry proteomics," *Molecular & Cellular Proteomics*, vol. 11, no. 12, pp. 1612–1621, 2012. (cited on Page 43)

[71] W. H. McDonald, D. L. Tabb, R. G. Sadygov, M. J. MacCoss, J. Venable, J. Graumann, J. R. Johnson, D. Cociorva, and J. R. Yates, "Ms1, ms2, and sqt—three unified, compact, and easily parsed file formats for the storage of shotgun proteomic spectra and identifications," *Rapid Communications in Mass Spectrometry*, vol. 18, no. 18, pp. 2162–2168, 2004. (cited on Page 44)

[72] R. D. Bjornson, N. J. Carriero, C. Colangelo, M. Shifman, K.-H. Cheung, P. L. Miller, and K. Williams, "X!! tandem, an improved method for running x! tandem in parallel on collections of commodity computers," *The Journal of Proteome Research*, vol. 7, no. 1, pp. 293–299, 2007. (cited on Page 48 and 93)

[73] R. Craig and R. C. Beavis, "Tandem: matching proteins with tandem mass spectra," *Bioinformatics*, vol. 20, no. 9, pp. 1466–1467, 2004. (cited on Page 49 and 56)

[74] GPM, "X! tandem spectrum modeler." (cited on Page 49, 52, 55, and 56)

[75] V. Abramova and J. Bernardino, "Nosql databases: Mongodb vs cassandra," in *Proceedings of the international C* conference on computer science and software engineering*, pp. 14–22, ACM, 2013. (cited on Page 59)

[76] L. Prechelt, "An empirical comparison of c, c++, java, perl, python, rexx and tcl," *IEEE Computer*, vol. 33, no. 10, pp. 23–29, 2000. (cited on Page 63)

[77] L. Prechelt, "Are scripting languages any good? a validation of perl, python, rexx, and tcl against c, c++, and java," *Advances in Computers*, vol. 57, pp. 205–270, 2003. (cited on Page 63)

[78] X. Shen and B. Wu, "Data placement on gpus," in *Advances in GPU Research and Practice*, Morgan Kaufmann, 2016. (cited on Page 69)

[79] N. Hillary, "Measuring performance for real-time systems," *Freescale Semiconductor, November*, 2005. (cited on Page 74, 76, and 84)

[80] J. Polo, D. Carrera, Y. Becerra, V. Beltran, J. Torres, and E. Ayguadé, "Performance management of accelerated mapreduce workloads in heterogeneous clusters," *Parallel Processing (ICPP), 2010 39th International Conference on*, pp. 653–662, 2010. (cited on Page 87 and 90)

[81] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala, "Sparkcl: A unified programming framework for accelerators on heterogeneous clusters," *arXiv preprint arXiv:1505.01120*, 2015. (cited on Page 89)

[82] G. F. Pfister, "An introduction to the infiniband architecture," *High Performance Mass Storage and Parallel I/O*, vol. 42, pp. 617–632, 2001. (cited on Page 90)

[83] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® omni-path architecture: Enabling scalable, high performance fabrics," pp. 1–9, 2015. (cited on Page 90)

[84] M. Pinnecke, D. Broneske, G. C. Durand, and G. Saake, "Are databases fit for hybrid workloads on gpus? a storage engine's perspective.," pp. 1599–1606, 2017. (cited on Page 91)

[85] S. Breß, "The design and implementation of cogadb: A column-oriented gpu-accelerated dbms," *Datenbank-Spektrum*, vol. 14, no. 3, pp. 199–209, 2014. (cited on Page 92)

[86] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015. (cited on Page 92)

[87] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004. (cited on Page 92)

[88] P. Howard, "Blazegraph gpu," 2015. (cited on Page 92)

[89] D. T. Duncan, R. Craig, and A. J. Link, "Parallel tandem: a program for parallel processing of tandem mass spectra using pvm or mpi and x! tandem," *Journal of proteome research*, vol. 4, no. 5, pp. 1842–1847, 2005. (cited on Page 93)

[90] B. Pratt, J. J. Howbert, N. I. Tasman, and E. J. Nilsson, "Mr-tandem: parallel x! tandem using hadoop mapreduce on amazon web services," *Bioinformatics*, vol. 28, no. 1, pp. 136–137, 2011. (cited on Page 93)