ACCELERATED COMPLEX EVENT PROCESSING WITH GRAPHICS PROCESSING UNITS

Prabodha Srimal Rodrigo

Registration No. : 138230V

Degree of Master of Science

Department of Computer Science & Engineering

University of Moratuwa

Sri Lanka

April 2015

ACCELERATED COMPLEX EVENT PROCESSING WITH GRAPHICS PROCESSING UNITS

Prabodha Srimal Rodrigo

Registration No. : 138230V

Dissertation submitted in partial fulfillment of the requirements for the degree Master of Science in Computer Science

Department of Computer Science & Engineering

University of Moratuwa

Sri Lanka

April 2015

Declaration

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Candidate

....

Date *U.P.S. Rodrigo*

The above candidate has carried out research for the Masters Dissertation under my supervision.

Supervisors

....

Dr. H.M.N. Dilum Bandara *Dr. Srinath Perera*

....

Date Date

Acknowledgments

This project would not have been possible without the support of many people. Specially thanks to my supervisors, Dr. Dilum Bandara and Dr. Srinath Perera, for their valuable guidance. Many thanks to our MSc research project coordinator, Dr. Malaka Silva, for his dedication and support. Thanks to all the lecturers at the Faculty of Computer Science and Engineering, University of Moratuwa, for their valuable advice. Also thanks to my wife, Malmi Amadoru and my parents always offering support and love. Finally, I thank to my numerous friends who endured this long process with me.

Abstract

As Big Data scenarios increasingly become common, a large number of distributed data processing systems require timely processing of high volumes of real-time data streams. Detecting complex correlations between incoming data streams in near real-time is at the heart of these data processing systems. Complex Event Processing (CEP) have been dominating in this domain since inception a decade back. But, growth of Big Data volumes demands for more performance and faster processing. CEP operators like stream join and event patterns require considerable processing power and have huge impact on the overall query processing performance. In some use cases these operators have to operate on lots of events simultaneously. Making parallel algorithms for these operators is a common approach for improving the individual operator performance.

A Graphics Processing Unit (GPU) provides a vast number of parallel computing cores and leverage new parallel algorithms which enables novel problem solving approaches for existing problems. But the challenge is combining complex event processing and GPUs in the right way to get the maximum performance out of the this parallel hardware. There had been attempts to use parallel hardware in improving CEP performance in both commercial and academic implementations, and most of them uses multi-core approach. Only a very few researches had used GPUs for CEP. We believe the lack of GPU related CEP researches is that they are not designed to benefit from parallel processing in GPUs.

In this research we investigate how and when GPUs can be used to improve the query processing performance of a popular open source CEP implementation, Siddhi CEP. Siddhi, by design, supports for parallel query processing in multi-core CPUs. This work propose a novel approach for parallel event processing in GPUs with several GPU event processing algorithms. Performance evaluation on our implemented algorithms shows, for a mix of complex queries, parallel event processing on GPUs achieve more than ten times event processing throughput than the sequential processing in CPUs. Moreover, our approach helped to reduce event queuing at the incoming event queue when there are high frequent input event stream and several complex queries.

Keywords. Complex Event Processing, Parallel Hardware, GPGPU, Siddhi CEP.

Contents

List of Figures

List of Tables

List of Abbreviations

CEP	Complex Event Processing
CMP	Chip Multi-processors
CUDA	Compute Unified Device Architecture
DBMS	Database Management Systems
DSMS	Data Stream Management System
EDA	Event Driven Architecture
FPGA	Field-Programmable Gate Array
GPGPU	General Purpose Computing on the Graphics Processing Unit
GPU	Graphic Processing Units
HPC	High Performance Computing
JSON	Javascript Object Notation
MIMD	Multiple Instruction, Multiple Data streams
MISD	Multiple Instruction, Single Data stream
POJO	Plain Old Java Object
SIMD	Single Instruction, Multiple Data stream
SISD	Single Instruction, Single Data stream
SMP	Symmetric Multiprocessing

Chapter 1

Introduction

Complex Event Processing (CEP) is a technology designed to infer complex patterns in streams of events. CEP is a widely used computing paradigm in today's IT enterprise systems. Most of the CEP use cases demands for faster and efficient processing of incoming events as soon as they are available. This is the major challenge that most of the CEP implementers are facing today. This chapter gives an introduction to the project while describing the problem domain and the problems associated with that domain, which this research is going to tackle. The Section 1.1 gives an overview of the limitations we face in today's enterprise IT systems with the high volumes of data. Section 1.2 and 1.3 describe in brief the complex event processing and emerging parallel hardware technologies and in Section 1.4 we describe how those two concepts used in synergy to improve the CEP performance. Brief overview of the project and problem statements follows in Section 1.5. The contribution of this work is described in Section 1.6.

1.1 Big Data and Data at Move

Today, there are so many data sources available around us generating unimaginable volumes of data per second [1]. These include social media inspired web activities like Twitter and Facebook data streams, sensor data from different sensor networks, stock market activity data, weather data, among others. These are so called *Big Data* and often called *data at move* since there are large volume of real-time data generating and flowing from various data sources to various data sinks. So it is very important to process these data streams as and when they are generated and derive interested information from these data streams in order to exploit the opportunities and maximize profits.

For example, consider the capital markets, where market activities happen very fast.

Market players who react fast to the changing market activities win the game in capital markets. The ability to react instantly to market movements is a competitive advantage and way to maximize profit and minimize risk [2]. Whether it is a sophisticated algorithmic trading application or decision support system for risk managers, it all boils down to the ability to intercept fast moving data from multiple sources and process them in real-time.

This is often the case for many other industries and use cases where distributed systems are involved. For example, consider following use cases; a fraud detection tools observing stream of credit card transactions to detect fraudulent usage patterns of credit cards in real-time [3]; RFID based inventory management systems, analyzing real-time sensor data to track RFID objects and detect irregularities [4]; a monitoring and controlling systems for a power grid that analyze sensor data from millions of distributed sensors and detect complex patterns in all the sensor data streams combined to optimize the power usage [5]. All of these use cases are challenged by the real-time process of fast moving, large volume of data.

In fact, a study done on digital data [1] says that, in 2012, 23% of the total data generated in the digital universe, or 643 Exabytes of digital data, would be useful for Big Data, if it were duly processed and analyzed. However, technology is far from where it needs to be, and in reality, it is only 3% of the potentially useful data that is processed and tagged. This shows how hard it is to analyze the Big Data and deliver useful information in real-time.

1.2 Complex Event Processing

The Complex Event Processing (CEP) technology emerged as a solution for analyzing fast moving Big Data and continue to grow its usage in event processing domain. Most of the complex systems are event driven. Most of the above mentioned data streams are comprised of continuous stream of *primitive events*. Event driven systems operate by observing one or many streams of these primitive events from external data sources, detect interesting combination of primitive events, and react for them once detected. These systems are build around a core event processing component called *Complex Event Processing Engine* [6].

CEP engines detect meaningful patterns in the input event streams (*event sources*), according to the user provided pattern definition (*event query definition*) and alert user about the detected patterns (*event sink*). There can be several event sources for a particular event query and these event streams can be differ from each other. The detected events are also output as a stream of events.

The CEP domain has been evolving throughout the last decade and it has gain attention from both academia and industry alike, because of its practical usage in solving real-world problems [7], [8]. There are many approaches and implementations of CEP engines proposed by both academia [9]–[12] and industry [13]–[15].

Current implementations of CEP engines in general features several characteristics like support for unrelated event streams, support for parametrized combination of event rules, processing events in temporal window, joining event streams, basic query language for defining complex event patterns and optimization of event queries. But above all, the most crucial characteristic of CEP is the low latency of event processing in the face of high volume of incoming events and emit output notifications as soon as an interesting pattern is detected. To this end, there had been lot of research work done to improve the event processing performance of CEP engines [16]–[19]. Some approaches took the path of changing the event processing core of existing CEP engines and some took the path of proposing completely novel architecture to event processing.

The *Siddhi CEP engine* [20], [21] is one of the CEP engines that took the novel approach of using stream processing techniques like multi-threading and pipeline architecture to make the event processing faster. Further, the architecture of Siddhi CEP closely follows the stream processing systems so it utilizes the current multi-core processors through multi-threading and producer-consumer architecture. Event processing performance of the Siddhi is one of its major advantages. The authors of Siddhi have shown that the engine has considerable higher performance than other existing CEP implementations.

1.3 Parallel Hardware

While the world is moving to the *data avalanche*, the computational processing power has also increased to help solving problem arised by this huge volume of data. This increased computational power is available through computational grids, clouds, multi and many core processors. The recently addition to the many-core processors are *Graphic Processing Units* (GPU).

GPUs are originally developed as a fixed function processor built around graphics pipeline to process 3D graphics. But over the past few years increasing number of communities have identified that GPUs can be used in other applications which need high computation requirements [22]. GPUs are designed to increase the throughput rather decrease the latency. Now the General Purpose Computing on the Graphics Processing Unit (GPGPU) has become a mainstream technology and used in growing number of use-cases which need High Performance Computing (HPC).

Compute Unified Device Architecture (CUDA) is a wide spread architecture for GPGPUs, invented by Nvidia [23]. CUDA provides a C-like programming infrastructure for GPUs with two levels of parallelism; data parallelism and multi-threading.

The GPGPU is closely bound to the underlying GPU hardware architecture. Programming to a particular GPU needs proper understanding about the key characteristics provide by the implementations of that GPU. The data structures and algorithms should be carefully designed to gain the right performance from the GPU computations [24].

1.4 CEP on Parallel Hardware

Performance of event processing is one of the core features and a major selling point of any CEP implementation. Because most of the CEP implementations are used in diverse mission-critical and time-critical scenarios with demanding performance requirements. This is why most of the commercial CEP implementations are trying to constantly improve their performance.

The definition of term *performance* vary widely in different CEP implementations. Some say its the number of incoming events per second handled by the CEP engine, while others say it is the number of concurrent queries handled by the CEP engine. Both metrics are important but none of them are complete as an individual metric for performance. Besides there are lot of other metrics to consider when measuring performance of a CEP engine [25].

The basic common operations supported by many CEP implementations are; pattern detection, filtering, transformation, windowing, aggregation, sorting, correlation (stream join) and merging (stream union). These operations will be described further in Section 2.2.2. The performance of a CEP engine directly depends on:

- the internal data structures and algorithms used for basic operations;
- basic operation parameters such as window type and size, and number of filtering events;
- speed and size of incoming data;
- number and type of queries, the complexity of the queries;
- external parameters such as available resources.

Performance is one of the key aspects in any CEP engine and parallel hardware are increasingly become available and common, but there had been only a few attempts

to harness the performance advantages offered by parallel hardware to increase the performance of CEP engines [26]–[28]. As we will describe in the Chapter 2, this is mainly because of the architecture and implementation of most of the current CEP engines are unable to take advantage of parallel hardware. Some CEP engine implementations are inherently single threaded and some uses data structures that are hard to parallelize. There are some CEP operations, such as "joining two streams with time window events", which require sequential processing and parallelizing them may require locking and waiting which negatively affect their performance.

1.5 Problem Statement

In this research our main goal is to investigate how and when GPGPUs can be used to improve the query processing performance of Siddhi CEP engine. As we have described above, using GPGPUs for accelerating event processing performance in CEP systems is a novel approach in event processing domain.

Current Siddhi architecture supports parallelism and it already uses the multi-core CPUs to improve query processing performance using multiple threads. But current implementation of parallel algorithms needs to be improved to gain high performance in processing complex rules with high volumes of incoming data. For example, some query operators currently require locking and processing of stored events one-by-one sequentially, which can negatively effect performance, if there are lot of stored events (time windows size is long) or there are high frequent incoming data. GPUs provides performance advantages over this kind of scenarios by processing individual events in parallel in multiple threads. GPUs may not provide low-latency processing for individual events, but collectively it yields high throughput, which is more beneficial to high frequent data processing scenarios.

We chose Siddhi as the CEP engine for this research, because it is an openly available, actively developed, emerging CEP engine that has used in many production systems and its architecture closely resembles stream processing systems and supports multi-threading. So we suppose the impact of this research will be higher on both research community and the industry.

1.6 Contributions

To achieve the main goals of this research, we have introduced a novel approach of off-loading event processing tasks to GPUs. Our main contributions are as follows:

- Through profiling existing Siddhi implementation using a real-time data load we have identified the most frequently used complex event processor operators that have high impact on event processing performance. We have designed and implemented several GPGPU-based parallel event processing algorithms for these highly used CEP event processors like filter event processor, window event processor and event stream join processor.
- In designing GPGPU-based event processing mechanism for Siddhi CEP, we identified the requirement of having a general purpose event processing framework for parallel hardware technologies. Thus, we have present the design and implementation of a general purpose event processing library for GPGPUs.
- We also present the implementation of a new Siddhi Query Processing Runtime for GPGPUs which follows the same interface as existing Siddhi Query Processing Runtimes and internally uses our general purpose event processing library for communicate with GPGPUs. Using this new Query Processing Runtime, we have implemented GPU event processors for filter operator, event window operator and event stream operator.
- Finally, in evaluating CEP systems, we propose the use of publicly available, real-world dataset and a set of standard queries to evaluate and compare CEP system's query processing performance. Using this data set we have evaluated our proposed GPU event processing algorithms and we have achieved, compared to sequential event processing, more than ten times event processing throughput for a mix of complex event queries.

Our experimental evaluation shows with the use of GPGPUs for event processing we can gain two times speedup of event processing throughput for complex operators like stream join operators. Moreover, our approach helped to reduce event queuing at the incoming event queue when there are high frequent input event stream and several complex queries.

1.7 Challenges

As we have mentioned earlier, there had not been much research work done in CEP domain to use GPGPUs for parallelization. According to our knowledge, this work is the first to use Java based CEP engine with GPGPU hardware. So we had to overcome lot of technical and design challenges during this research. Some of these challenges are listed below.

- CUDA-C is extension to C programming language. Siddhi is implemented using Java. So we had to design a Java to CUDA-C/C++ communication mechanism

that does not negatively impact on data transfer performance.

- Siddhi internally represent Event attribute data as a flat generic data structure (i.e. array of Java Objects). It does not use data structure that represent event schema to represent actual events. Hence event serialization and de-serialization takes considerable processing power.
- CUDA is generally used to solve data parallel computations problems, and it is not generally recommended for event-based parallelism. So stream-based processing are generally not suitable for CUDA. It was challenging to identify which part of the processing should off-load to GPUs and which part of the processing should be done in CPUs.

1.8 Organization of the Thesis

The reminder of this thesis is organize as follows. Chapter 2 presents the background of this research, which covers CEP systems, parallel hardware systems (multi-core, GPU and cell processors) and GPU Programming Environments. In Section 2.2, we presents the current architecture and main implementation concerns of the Siddhi CEP engine. In Chapter 3 and Chapter 4, we present our proposed approach to the improving performance of Siddhi CEP engine with use of parallel hardware. Chapter 3 describes the changes we done in Siddhi to accommodate GPU event processing while Chapter 4 describes the implementation of general purpose GPU event processing library that we have implemented. Chapter 5 presents evaluation of our proposed approach using real-world workload and Chapter 6 summarizes our work and suggest future works.

Chapter 2

Literature Review

Complex event processing demands for high performance event processing while parallel hardware providing a platform for developing high performance application systems. But the challenge is combining these two domains to work together in order to improve event processing performance. Understanding both domains and their internals helps the challenge solving process.

This chapter first gives a overview of problem domain and then expands a thorough literature of current state of the art, while differentiating each related work with this research work. Since this research resolves around two major domains – Complex Event Processing and Parallel Hardware, this chapter has two parts describing each domain. Section 2.1 presents overview of Complex Event Processing while Section 2.2 describe Siddhi CEP and its internal implementation details in higher level. In Section 2.3 we describe current parallel hardware technologies including GPGPUs. Finally, in Section 2.4 we present related researches.

2.1 Event Processing

In computing, an *event* is defined as *an occurrence which happened in past, currently happening or considered as happened within a particular system or domain (event occurrence)* [29]. In the programming context, the word *event* is also used to denote the computational representation of the particular occurrence (*event entity*). The computational representation usually in the form of a state change of one or more attributes related to that particular occurrence. These state changes are encoded in programming entity called *event object* or *event tuple* for ease of computer processing. For example, following is a set of interesting event occurrences and their event entities;

- a weather sensor outputs its readings as message,

- a stock is traded and reported as a stock tick message,
- a monitoring system detects an application server crash and send a alert message,
- a tsunami wave is detected by a detection buoy and sends an alert signal,
- a twitter post is published to twitter feed,

It is usual that a single event occurrence represented by many event entities, and a single event entity may not represent the all attributes of particular event occurrence. As stated in the event definition the event entity can represent an occurrence that has already happened or currently happening. It is also possible to represent an occurrence that considered as happened but did not actually happened. For example a network intrusion detection system can output a false positive detection event of intrusion, where the detection system contemplated an intrusion has happened, when it was not.

There are systems which generate and disseminate events, such as applications, data feeds and data stores. These are called *event sources*. There are two methods of disseminating data; by pushing as in data feeds and by pulling as in data stores.

An *event stream* is a collection of associated, temporally ordered events. The events are ordered based on a timestamp in each and every event in the stream. Usually in an event stream we encounter events with same semantic meaning and structure. For example an event stream from a stock exchange trading engine consist of events describing trading activities (i.e. trade report messages). These events with same semantic meaning are said to have same *Event Type*. Although an event stream can contain events in same type, which it called homogeneous event stream, it is also possible to contain events of different types, which it called heterogeneous event stream.

Each of the example events described above have one or more attributes that change when the event occur. These attributes often called *data* items and as already described an event contains one or may data items. For example, the weather sensor output may contain wind speed, wind direction and humidity level as its attributes. The stock trade has the trading price, trading quantity and trade time as its attributes.

Data is not much useful in its original nature; they are nothing but unorganized facts. The wind speed is not a much useful data by its own, but if we combine and analyze series of wind speed readings with humidity level readings and how they correlate, we can derive useful information about future weather conditions. So the data needs to be converted, analyzed, stored, aggregated or summarized to yield useful information out of them. *Data processing* is one of the main computing task which interprets input data and generate useful information as output. Data processing systems are responsible for data processing and comprised of either hardware, software or liveware (i.e. people).

While the Data Processing Systems operate on data, the *Event Driven Systems* or *Event Driven Architecture (EDA)* operate on streams of events. Event driven systems

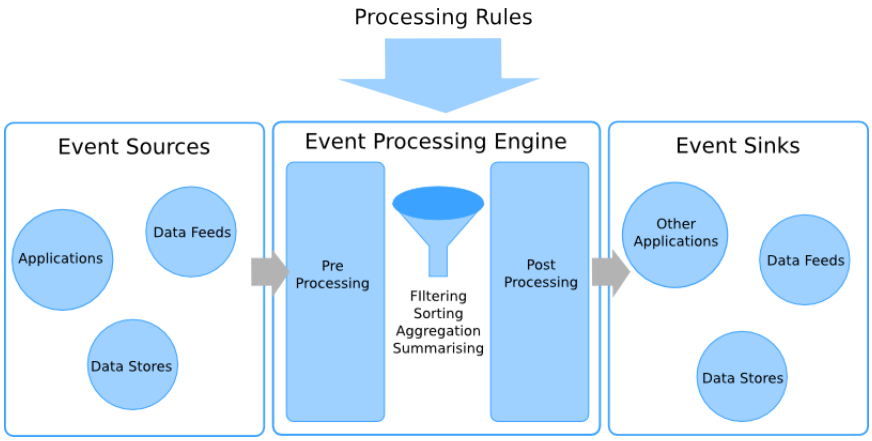


Figure 2.1: Event Processing Architecture.
Event processing systems architecture consist of event sources, event processing engine and event sinks.

process event streams to convert data inside events into usable information stream. Event driven architecture is increasingly become useful in mission critical domains where high performance data processing is the utmost important requirement. EDA has four main sub-systems (see Figure 2.1); i) event sources, who generate and disseminate events as streams, ii) an event processing engine, who convert and process incoming events and derive useful information and output them as a stream, iii) event sinks, who consume output event streams from event processing engine, and iv) event processing rules, which define how the event processing engine should operate on incoming data. Apart from this, an EDA is also associated with a messaging infrastructure which helps to deliver events from its sources to the center of the EDA. Messaging infrastructure should be capable of doing one-to-one, one-to-many, many-to-one, and many-to-many communications, whichever is used by event processing engine. Event processing engine is the main focus of this research, this can be either a complex event processing engine, high-speed rules engines, neural networks, Bayesian networks, or other analytical models.

The event processing rules are added and removed from the event processing engine by users of the system. Users define how to filter, aggregate, combine and summarize incoming event streams according to what information they need to be output from the system. Usually the implementation of event processing engine defines a *query language* to express event processing rules to the system. Query languages can be categorized into broadly two categories; rule-oriented languages and stream-oriented languages [29]. Most of the current implementations of stream-oriented languages are extensions of SQL [30]. Event sinks can be another data processing system, a data store or sometimes can be feedback to the same event processing engine.

There are several reasons why event processing approach is used in systems and applications, such as:

- When the application is mainly dealing with events, like the example scenarios we explained in above sections, and the main objective of the application is to analyze those events and react to them as soon as possible.
- When the main objective of the application is to detect certain *situations* in incoming event streams as they occur and react to them in timely fashion. This is the opposite of batch processing method.
- When the application has to process high volumes of incoming events and output derived information to another system or user. This high volumes of data can be send to distributed event processing applications and process them parallelly.
- When the decoupling is necessary between information sources, information processors and information users.

Event processing use cases have two unique requirements that are not catered by any other existing systems: *stream processing* and *timeliness*. These new requirements paved the way for creating new class of systems. Researchers from different background and domains came up with different solutions for this problem of event processing and most of these solutions are evolution or adaptation of these existing systems or domains. Currently there are two major models designed for event processing use cases: the *data stream processing* model [31] and *complex event processing* model [32].

2.1.1 Data Stream Processing

The data stream processing model is an evolution of traditional data processing systems supported by Database Management Systems (DBMS). Traditional DBMSs are designed for use cases where data is first stored and indexed before they are used for processing. The actual data processing is happened when a particular user explicitly query for data which is already stored. In contrast, event processing use cases have complete opposite requirements of these. For example, take a network intrusion detection system which dose processing of network events to detect possible intrusions to the network system. There the main requirement is to analyze network events like traffic flow, network device connection and disconnection, in real-time to detect predefined patterns of possible malicious activities in the network and alert required parties as soon as an intrusion is detected. There is no real requirement of storing all the network events in a data store before they are processed. If all the network events are stored in a database at the rate of event occur, it will require huge storage capacity and high performance storage system. So storing of incoming events are not required unless they are related to detection of particular incident; the malicious activity in

intrusion detection use case. All stored events can be discarded once the particular incident is detected where the output event (intrusion alert in this case) includes all the necessary attributes to identify the incident.

The *Data Stream Management System (DSMS)* was designed following the DBMS principles and adopting to event processing requirements. The DBMS and DSMS systems have similarities and substantial differences. Both DBMSs and DSMSs have common way of processing incoming data using processing logics defined by SQL operators like selecting, aggregating and joins. DBMSs are designed to work with less updating persistent data while DSMSs are designed to work with frequently updating transient data. In DBMSs, queries are executed on persistent data set only when the query is invoked by a user. In contrast, DSMSs process the data as a flow as opposed to a dataset and execute standing queries over flowing data stream. In summary, DBMSs have static data and dynamic queries, where as DSMSs have dynamic data and static queries. DSMSs are incomplete in terms of covering the requirements of event processing. They are only capable of answering to the user defined queries, but cannot do most primitive operators in event stream processing like sequencing and ordering of events.

2.1.2 Complex Event Processing

The DSMSs are limited in their capabilities since they are generic in terms of incoming data processing and clients are expected to associate the semantics to incoming data with queries. The CEP model gives the incoming data streams correct semantics by treating them as event occurrences happened outside the system. These event occurrences are detected or produced by event sources and fed into CEP system. The CEP engine do sequencing, combining and filtering of incoming events to derive composite, higher-level events out of incoming events.

2.1.2.1 Primitive Events and Complex Events

Before defining the Complex Event Processing, it is important to understand the difference between primitive event and *Complex Event*. Complex event describes an incident which was derived with complex correlations from set of other primitive or complex events. Complex event encompasses attributes of that set of other related events as a summary. Moreover, it can contain additional attributes that did not present in events which it summarized [6]. Even though all derived events are complex events, not all complex events are derived events as they can be generated from event sources. Usually the definition of complex event is tightly coupled with the context

of processing. A complex event for one processing context may be a simple event for another context. Set of complex events are listed below.

- A Stock trade - Summarizes the events happened during the purchase of a Stock, such as matched bid and offer prices, time of the purchase, how much quantity was traded, etc.
- A credit card fraud detected - Summarizes the events happened during the detection process and their correlation that lead to detection of the fraud, such as a credit card with historically low transactions rate suddenly did series of high value transactions within few hours.
- Detection of fire - Summarizes a series of smoke and heat sensor readings and their relations, such as smoke sensor detected a smoke in a particular area and the temperature of that location was above 60 degrees for more than 15 minutes.

2.1.2.2 Event Stream Processing

The complex event processing provides a software infrastructure to detect patterns in event streams from multiple live data sources by filtering, combining, correlating, contextualizing and analyzing, and respond to its environment as defined by the processing rules. CEP combines events from multiple sources to derive event patterns that describe more complex situations, more meaningful events or situational knowledge and respond in timely manner. The input to complex event processing can also include simple/raw event streams.

Complex event processing concept is different from event processing paradigms in terms of its support for temporal queries. With temporal queries event subscribers are able to exploit time based relationship between events in event streams like “time windows” and “before and after relationship”. Usually in CEP terminology event processing is the preprocessing phase of the CEP engine, where input events are normalized and prepared for processing in higher level CEP processors. These concepts are further explained in Section 2.2.2.

The best way to convey the idea of complex event processing and its relation to the EDA is to compare it with human body and its functionalities [33] (see table 2.1).

Complex event processing systems has its roots in message oriented middleware systems like publisher-subscriber systems [34], [35]. In publisher-subscriber systems, users (i.e. subscribers) are subscribed for channels of their interest. These channels are equal to event streams with homogeneous events (events with same type or class). The publishers publish data to channels without directly addressing subscribers. The event dispatcher in between do the hard work of filtering events based on their content

Table 2.1: Similarities between Event Driven Architecture, Complex Event Processing and Human Body.

Human Body	**Complex Event Processing**	**Functionality**
Senses	Transactions, log files, edge processing, edge detection algorithms, sensors	Direct interaction with environment, provides information about environment
Nervous System	Enterprise service bus (ESB), information bus, digital nervous system	Transmits information between sensors and processors
Brain	Rules engines, neural networks, Bayesian networks, analytics, data and semantic rules	Processes sensory information, "makes sense" of environment, formulates situational context, relates current situation to historical information and past experiences, formulates responses and actions

or topic and forward them to relevant subscribed users. There are two types of publisher-subscriber systems: topic-based and content-based. Topic-based publisher-subscriber system allows subscribers to subscribe for a particular topic, which is equal to a channel. In content-based publisher-subscriber system users are subscribed to messages with particular attribute value. The publisher-subscriber engine do the filtering of incoming channel messages based on the received subscriptions and do the dispatching of data. The subscriptions are described using specialized languages varying from simple attribute-value pairs to XML-based filter languages. Publisher-subscriber systems only support subscriptions for events with same type and do not consider complex relations between events and their history [36]. This basic idea of filtering data based on subscription has evolved into Complex Event Processing engines which added facilities to extract complex relationships between events in one or more event streams.

Similar to the event driven architecture, complex event processing system has four main components: event sources, event processing engine, event sinks and event processing rules. The event processing engine, called *Complex Event Processing Engine*, dose the detecting of patterns of complex and primitive events, generate new composite events by aggregating and combining matched incoming events. The CEP engine interprets the detection rules defined by rule language and configure its processing logics according to those rules. The expressiveness of the rule language is utmost important in a CEP system, because it decides the capabilities of the event processing engine exposed to users.

2.1.2.3 CEP Use Cases

Early days use of CEP was limited to set of use cases that require quick response to changes around it environment like algorithmic trading, pattern recognition in sensor data. But today, CEP has extended its use in vast number of use cases covering different domains. This is due to the fact that in today's IT enterprises events are more frequent and enterprises architectures increasingly follow event driven architecture. Majority of CEP use cases can be broadly categorized in to following categories.

- **Situation Detection**
  Analyzing incoming events to detect event occurrence patterns that would show existence of new opportunities or problems in the event context. In this use cases incoming events are filtered using specific event attribute or complex event correlations while checking existence or absence of event attributes. Once the existence or absence of interested situation is detected, a high-level event is emitted as a result.

- **Data Aggregation and Analysis**
  In this continuous computation use cases, incoming data is correlated, grouped, aggregated and combined, and then computations such as averages are applied to aggregated data to generate novel information. Most of the time these use cases output summarized and higher level statistics information of incoming data. Example real-world use cases are:
  - continuously updated key performance indicators (KPIs)
  - continuous aggregation of data from multiple sources to show *the big picture*
  - continuous price adjustment based on market movement

- **Data Collection**
  While CEP analyze events in real-time, in some use cases, incoming raw events and CEP resulting higher level summary data are stored in separate data store for offline analysis or for recording purpose. Sometimes these stored events are used as the context of processing newly arriving data.

- **Application Integration and Intelligent Event Handling**
  In an event driven architecture, integrating different application systems by facilitating communication between those systems is a common practice. CEP can provide intelligence within an event driven architecture to analyze events in the context of other events and a knowledge of the state of various integrating systems to determine the routing of events between systems and determine the action to be taken based on an event.

2.2 Siddhi CEP Engine

As we described in the introduction chapter, in this research our main focus is on to improve the performance of CEP implementation called *Siddhi CEP* [20]. Siddhi CEP engine is an open source CEP implementation available for free to general public for commercial and non-commercial use under Apache Software License v2.0. Similar to most of other CEP implementations, Siddhi was started as an undergraduate project at University of Moratuwa, Sri Lanka, and soon became an internal project at WSO2, where it is now being improved*. Siddhi is implemented using Java as a "jar library", which enables the easy use of CEP functionalities in any Java application. It is currently being used in many production systems covering many real-time use cases. Siddhi's architecture closely resembles stream processing systems and supports multi-threading.

2.2.1 Siddhi Architecture

The designers of Siddhi has mainly differentiate it from other CEP implementations by its performance. Siddhi's architectural design and implementation is done with performance in mind. The high-level architecture of Siddhi CEP engine as shown in Figure 2.2 is consist of four main components: i) input adapters, ii) Siddhi-Core, iii) output adapters, and iv) query compiler. Each of the modules are described below.

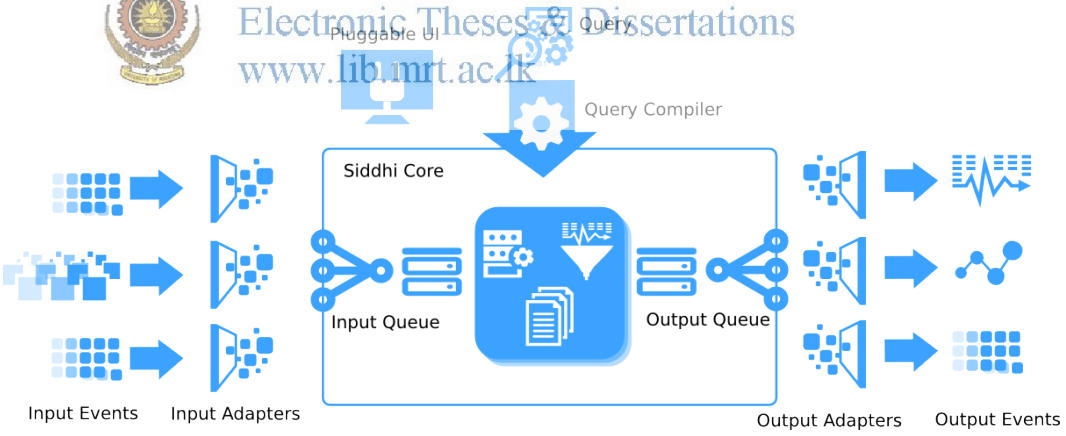


Figure 2.2: Siddhi High Level Architecture.
Siddhi architecture is consist of four main components: input adapters, Siddhi-Core, output adapters, and a query compiler.

- **Input Adapters**
  Input event streams to the Sidhhi engine are handled by input handlers. Usually in practical scenarios, there can be several input event streams to the event processing engine from different event sources. These different event streams can

*http://wso2.com/products/complex-event-processor

be in different forms or wrappers like XML messages, JSON messages, POJOs, emails, or proprietary binary messages. The input adapters provide an interface to these different event streams and convert them into a common easy to process representation, which is currently a *tuple* data structure. There are several input handler implementations to handle different event forms.

- **Siddhi-Core**
  The most important part of the Siddhi is its rule processing engine called *Siddhi Core*. Input events are processed according to the constructs defined by the input queries and emit detected event pattern as an output event. Siddhi core is consist of several sub components such as executors, event queues, processors and callback handlers. Normalized input events from input adapters are appended into input queues where processors fetch them from there and append resulting events to output queues. Each sub component of Siddhi-core will be described later.

- **Output Adapters**
  Output adapter dose the reverse operation of input adapter. Once a complex event pattern is detected by the event processing engine, the resulting event is converted to a representation suitable format and notified to event subscriber (i.e. event sink) by the output adapter. There can be several event sinks who accept resulting event stream in different formats like XML messages, JSON messages, emails, SMS, Database update, etc. There are separate output adapters for each of these different formats.

- **Query Compiler**
  Siddhi supports a SQL like query language called *SiddhiQL*[†] to provide user queries to processing engine. The query compiler dose the validation and interpretation of SiddhiQL using ANTLR language recognizer. Validated queries are compiled into query object model, which is used by the Siddhi-core to drive its processing. Siddhi's internal data object model will be explained in next few sections.

Apart form the above main modules, Siddhi has a pluggable user interface module which can be used to display useful statistics and monitoring tasks.

2.2.2 Siddhi Internal Data Model

Siddhi's internal data model is one of its critical design component. Since our main goal of this research is to improve the performance of Siddhi engine, it is very important that we study these internal data model and how they are used. In this chapter we will explain in higher-level the current data model and in a separate chapter we will

[†]http://docs.wso2.org/display/CEP300/Introduction+to+Siddhi+Query+Language

critically evaluate the design choices and use of data model in Siddhi architecture. We will further explain in implementation chapter the internals of each event processing constructs we have improved preformance using parallel hardware.

Users of Siddhi do not need to manually create any of the internal data structures. It is all automatically created once a user provided Query is compiled. In other words this means the internal data model of Siddhi is tightly coupled with the SiddhiQL. So when we are describing each data model and its components we will associate it with the relevant SiddhiQL definition.

2.2.2.1 Event and Event Streams

As explained earlier Siddhi uses Tuple data structure to internally represent an Event and its attributes (see Figure 2.3). Since this event representation is used almost every other part of the Siddhi-core, and tuple is a simple data structure which allow basic operations like creation, copying and retrial of attribute data very effectively, it suites the event representation perfectly. Tuples can associate a schema where it can represent the class of events and event structure, but this is not mandatory in every use case.

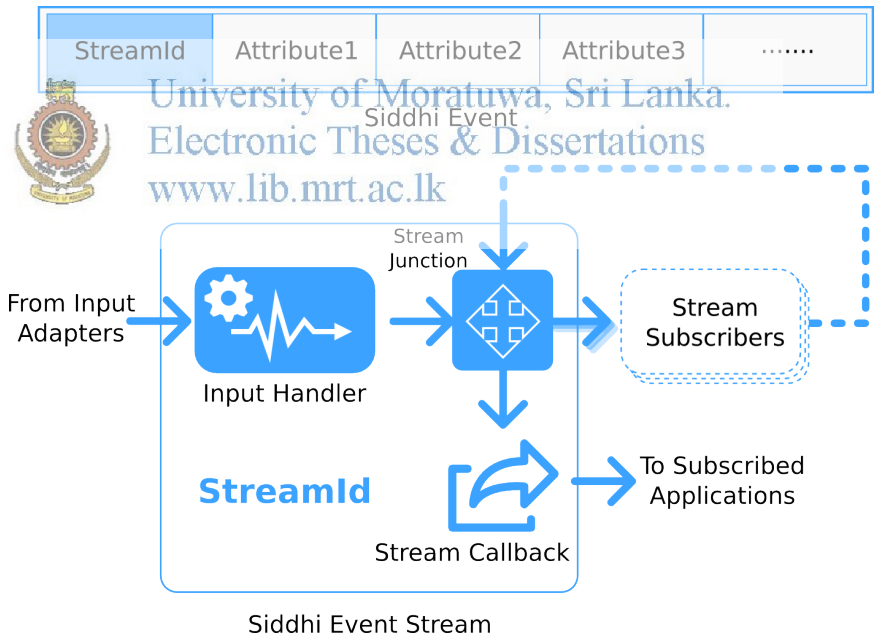


Figure 2.3: Siddhi Event and Event Stream representation.
Event object is represented using a Tuple data structure. Event streams representation has few data structures like input handler, stream junction and stream callback.

An Event Stream is represented with unique *Stream Id* and a set of data structures: input handler, stream junction and stream callback. All the incoming events from outside the CEP engine received through input handler. Event received through input handler are sent to its associated steam junction. Stream junction act as a hub for

events going out and coming into stream. Other components (i.e. Query handlers) who wants to get events from this event stream need to first register with the event junction of that event stream. Once they do, all the events received to stream junction is published to all the subscribers. Each subscriber gets a copy of the events. Apart from the event received via input handler, stream junction can receive events feedback from event subscribers, once they have processed events and emit a resulting composite event to their output stream. These feedback composite events are published to all the subscribers similar to incoming events. The stream callback is a special kind of stream subscriber which is used to notify about an event occurrences on the event stream. Usually the applications who are interested in getting notifications on event occurrences subscribe to Siddhi stream callbacks, while internal components like query handlers directly subscribe to event junction to get events from stream.

A stream is defined using SiddhiQL by specifying its unique id and set of attributes (see Listing 2.1). Each attribute is a pair of attribute id and its type.

```
1 define stream StockStream (symbol string, price float, volume
    int);
```

Listing 2.1: SiddhiQL for define a stream.

2.2.2.2 Query Data Model

Once a stream is defined, a query can be defined to get required information out of that stream. Queries can be vary from its complexity by the type and number of processing stages used. The following processor are supported in Siddhi.

1. Filter processor
2. Window processor
3. Join processor
4. Sequence processor
5. Patterns processor

A typical SiddhiQL query will contain one or many of above query types and a projection (see Figure 2.4). Projection defines the output event stream and selects which attributes should include in the output stream. A user can select all attributes from incoming events to include in output events or she can select set of specific attributes from incoming events to include in output events. Additionally, she can define summary attributes which are calculated using aggregated incoming event attribute values like average, summation and count.

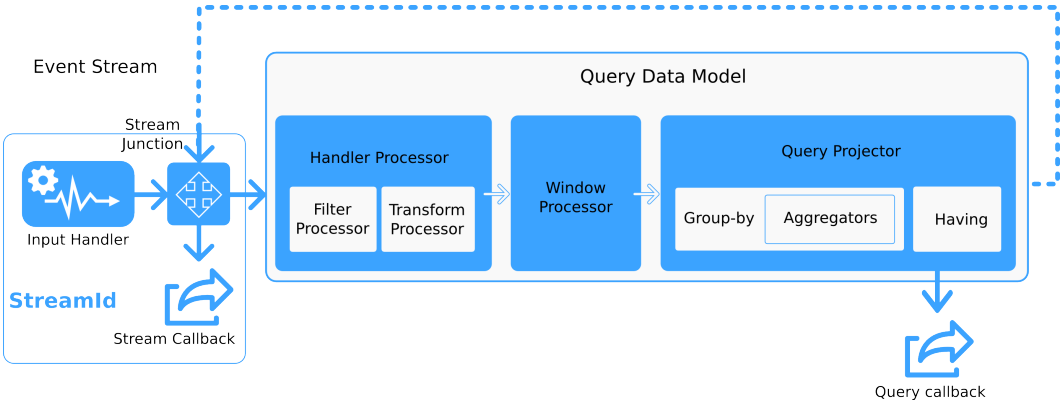


Figure 2.4: High level Siddhi query architecture.
At high level, Sidhhi query consist of several processing stages in between incoming event stream and output event stream.

Filters

Filters are the basic processing method and the first one to execute on the incoming events in the stream. In *filters* events are filtered out based on specific values for specific attributes. Only the successfully matched events are forward to next stage of processing, others are dropped out. Following condition operators are supported in Siddhi for attribute filtering.

1. `>`, `<`, `==`, `>=`, `<=`, `!=`
2. `contains`, `instanceof`

There can be several attribute value filters combined with binary operators like `and`, `or` and `not`. It is also possible to not have any filters on incoming event stream and just output only selected attributes to output stream. These type of queries are called *Pass-through* queries.

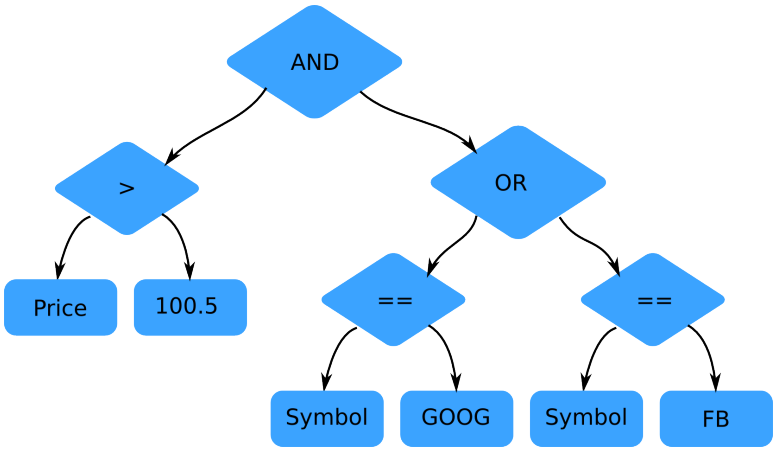


Figure 2.5: Siddhi filter condition data model.
Inside filters, the conditions are hierarchically structured as a tree of operands and operators.

In a filter there can be one or more condition parameters. For ease of processing, inside the filter processor, these conditions are hierarchically stored as a tree of operands and operators (see Figure 2.5). The evaluation of set of filters structures as this tree is started at the root of the tree and follows the depth first search algorithm. This way it is possible to achieve high performance of filter processing as the tree is evaluated to false when there are sufficient conditions are evaluated to false. So processing can stop as soon as possible. But this logic is totally depend on the order of condition definition in SiddhiQL, which is done by the users. So users are advised to order conditions with decreasing order of least success probability.

Windows

After the filter stage, events are processed in window stage if it is defined. In this stage some of the filtered events are sustained for a certain period of time for the purpose of temporal processing, like aggregation data calculations. The windowing policy can be defined in amount of time to wait or number of events in the window. Following windowing policies are supported in Siddhi.

1. **Length window** - a sliding window that keeps last N events.
2. **Time window** - a sliding window that keeps events arrived within the last T time period.
3. **Time batch window** - a time window that processes events in batches. A loop collects the incoming events arrived within last T time period, and outputs them as a batch.
4. **Length batch window** - a length window that outputs events as a batch only at the n^{th} event arrival.
5. **Unique window** - keeps only the latest events that are unique according to the given unique attribute.
6. **First unique window** - keeps the first events that are unique according to the given unique attribute.
7. **External Time Window** - a sliding window that processes according to timestamps defined externally (Defined as an attribute in the incoming stream)

There are two types of event outputs from a window: in-events and expired-events. When a new event arrives to the window processor, it creates a new expired-event and append it to the event window, and at the same time forward the original event to the next processing stage, which is "Query Projector" (see Figure 2.6). At the event window, when an event arrives an expired-event is created with the event timestamp value set to the original event time plus event window expiry time. This expired-event is stored in event window, which is a queue of events, and window processor schedule the expiry of event. Window processor continuously monitor events in event window for expiry using their timestamp. This is easy to implement since events are stored in the

event window in FIFO order. When an actual event expiry occur, that expired-event is removed from the event window and forward to query projector.

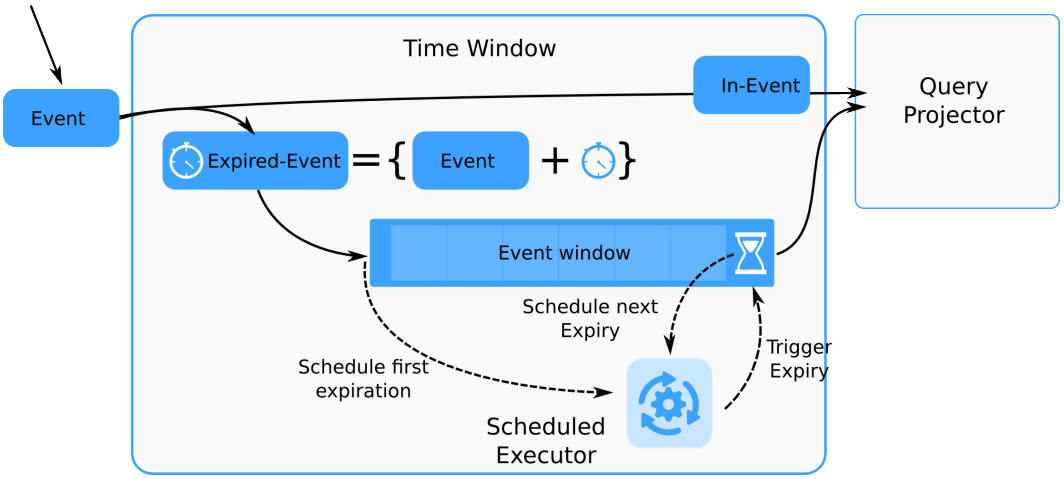


Figure 2.6: Siddhi time window architecture.
Temporal event processing using a time window.

Two things happens in aggregation processor at the query projector when events arrive. If the incoming event is an in-event, it is used to increase the aggregation and if the incoming event is an expired-event it is used to decrease the aggregation. For an example, if the aggregation function is to calculate the `count` of events for past five minutes, the time window is defined to 5 minutes. In-events to aggregate processor increase the count value, while expired-events to aggregate processor decrease the count value. Following aggregate functions are supported in Siddhi.

- Sum - Summation of event attribute values
- Avg - Average of event attribute values
- Max - Maximum value of event attribute values
- Min - Minimum value of event attribute values
- Count - Number of events in windows

The following query definition (see Listing 2.2) filters out events from stream `StockStream`, whose attribute `symbol` is having value "GOOG" or "FB" and creates a output event stream called `TopStockStream` with event attributes `price`, `volume` and number of events matching to above filter in past one minute period as `stockCount`. Output stream event attributes are filled from matching events from input stream.

```
from StockStream[price > 100.5 AND symbol == "GOOG"
        OR symbol == "FB"]#window.time(5 min)
insert into TopStockStream
     price, volume, count(price) stockCount;
```

Listing 2.2: SiddhiQL for define a simple query.

Stream Joining

Joining two streams using a set of join conditions and output event attributes from each input streams is a common form of query. For each input event stream there is a handler process created internally in Siddhi. To perform joining of two streams , each stream must have an associated time window defined. At the stream join processor event streams are considered pair wise and there are two join stream processors, called "In-Stream Join Processor" and "Remove-Stream Join Processor", before and after window processors. When an event arrives to the in-stream join processor, the event join condition is evaluated against all the stored events in other stream's Window Processor. If a match is found that event is forward to the Query Projector as in-event, and at the same time a expired-event is created and added to streams Window Processor. When an event is expired and removed from a window processor, it is sent to the remove-stream join processor where the expired-event is again matched with all the available events in other streams window processor. If a match is found, the expired-event is sent to Query Projector as expired-event.

A sample join query is shown in the Listing 2.3 and the internal data model is shown in Figure 2.7.

```
from StockQuotesStream#window.time(5 min) as sqs
    join HighFrequentTweetStream#window.time(15 min) as hfts
    on sqs.symbol == hfts.company
insert into InterestingStockQuotesStream
    sqs.symbol as company,
    sqs.price as lastTreadedPrice,
    hfts.words as wordsTweeted
```

Listing 2.3: SiddhiQL for define a simple Join query.

Aggregate and Join queries are most expensive in terms of performance. This is due to the fact that in joining a window is locked during the matching process for each event from other stream. This locking is necessary to ensure accuracy and avoid race conditions in the matching process. With the high volumes of incoming events and with every event arrival in either stream triggers the joining process and effectively locking the time window of other stream, this can be huge overhead to event processing performance. Thing get worse when the time window size is getting long and there are

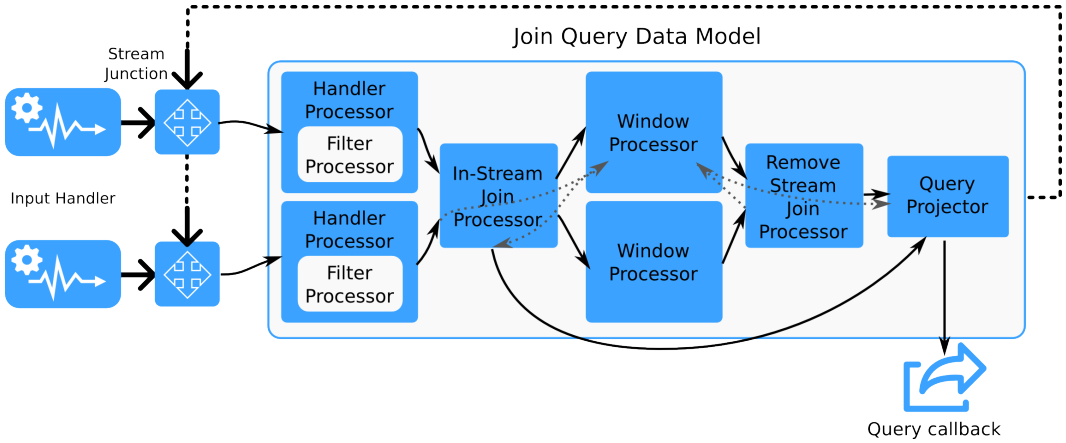


Figure 2.7: Siddhi Join query data model.
Data model of a join query with two data streams.

lots of events stored in time window, which is very common scenario. In this research we are more concerned about aggregate and join queries as we think there are rooms for further optimizations using parallel hardware.

2.2.3 Processor Architecture

Siddhi processing architecture follows producer-consumer design pattern. There are event generators and event consumers. Siddhi has *Processors* as the basic building block and chaining processors forms the processing architecture. Processors have an input event queue, an output event queue, an event generator and set of *Executors*. Executors are the internal processing element of processors (see Figure 2.8). Generating and chaining of executors are done by the Query parser using the query object model which was generated using user defined query in SiddhiQL.

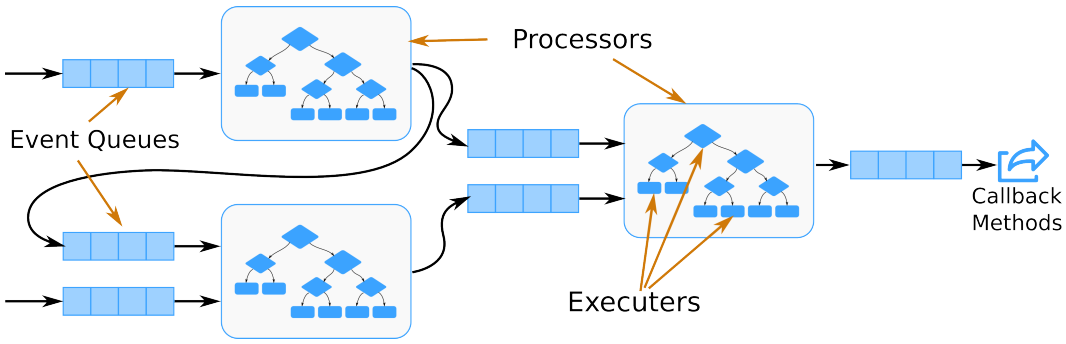


Figure 2.8: Siddhi Processor Architecture.
Siddhi processor architecture follows the producer-consumer design pattern.

In the perspective of single processor, input events are placed into the input event queue, where they are fetched one at a time by the processor and fed into internal executors. Executors process each event and return boolean result saying whether the processed

event has matched or not. If the event is matched, it is forward to next executor or event generator, otherwise event is discarded. This processing method increases the efficiency of processing and gradually decreases the number of events to process as the event flow through the processing engine, if queries are optimized to have the least success filters in the front.

Event generators generate output events based on the query definition and place them into output event queue. Each output events from the processors are put into output event queue as soon as they are available. Output event queue of one processor can be an input event queue of another processor. A processor can have more than one input event queues and more than one output event queues.

Executors inside the processor are arranged in tree like structure (see Figure 2.5) and there can be more than one tree of executors in a processor. But only one tree of executors get processed at a given time. As explained earlier, executor tree is processed starting from its root and follows the depth first search order. At the each executor the event is evaluated and return a boolean value. If the returned value is false, the further execution stopped and return false recursively and the tree returns false. This way the unnecessary processing can be avoided as soon as the one executor evaluated to false. We believe the executor processing model can be parallelize and make it more efficient using parallel hardware. There are several types of executors defined in Siddhi implementation.

- AndExecutor,
- OrExecutor
- NotExecutor
- ExpressionExecutor
- PatternExecutor
- FollowedByExecutor

2.3 Parallel Hardware Architectures

Since 2002, the performance scaling curve for single CPUs has slowed to a great degree, where by ruling out the Moore's law for uniprocessors. The Moore's law says computer power doubles every 18 months [37]. With the number of transistors increases in a single die, soon the hardware will reach the limits of silicon. Due to the power and thermal considerations, increasing transistor count in single chip no longer will offer performance improvements for single threaded applications. For this reasons research

community has look into other ways to achieve the performance through *parallelizing the processing* and use of *heterogeneous computing architectures*.

There exist several architectures to enable parallel processing of data, which are classified using Flynn's taxonomy [38]. The classification is based on the concurrency of instruction and data streams available in the architecture. An instruction stream is the set of instructions that makes up a process, and a data stream is the set of data to be processed.

1. **Single Instruction, Single Data stream (SISD):** a sequential computer which dose not exploit any parallelism, like uniprocessor system.
2. **Single Instruction, Multiple Data streams (SIMD):** a single instruction is broadcast to array of compute units and each unit execute the instruction on different data item. Vector/array processors (i.e. GPUs) and Broadband Engines are the most common examples of this category.
3. **Multiple Instruction, Single Data stream (MISD):** usually used in fault tolerant computing systems like Space Shuttle flight control systems.
4. **Multiple Instruction, Multiple Data streams (MIMD):** multiple processing units like computing cores process multiple data streams using multiple instruction streams.

Today's most common parallel hardware architectures, with the decreasing scale, are;

- **Grid computing systems** - a combination of computer resources from multiple administrative domains applied to a common task.
- **Massively Parallel Processor (MPP) systems** - also known as "Supercomputer architecture", where large number of processors (or separate computers) to perform a set of coordinated computations in parallel.
- **Cluster computing systems** - use of network of general-purpose computing nodes to perform a set of coordinated computations in parallel.
- **Symmetric Multiprocessing (SMP) systems** - multiple, identical CPUs/cores (in power of 2) is connected together to work as one unit.
- **Multi-core processor systems** - a single chip with numerous computing cores, also known as "Chip Multi-processors (CMP)".

Most of the above parallel hardware systems, such as SMP and cluster computing systems, fall under MIMD category in Flynn's taxonomy. So they are further categorized using the type of memory they use: shared memory and distributed memory. In shared memory type systems, each CPU that makes up the system is allowed access to the same memory space, where as in distributed memory type

systems, each CPU that makes up the system uses a unique memory space. MPP and cluster computing systems are the most common distributed memory type systems. SMP is a shared memory type system. In SMP, the multiple CPUs/cores have access to a single shared main memory, which makes parallelization of serial code relatively straightforward. The main programming method for parallelization on SMP architectures are using POSIX threads (pthreads) and OpenMP. The POSIX threads standard defines an application programming interface (API) for explicit creation, management and synchronization of multiple threads, whereas OpenMP mainly consists of a set of compiler directives (and a supporting API) that allows for implicit parallelization.

Efficient memory access is an important design consideration in multiprocessor systems with many cores where increasing the number of processors naturally increases the number of accesses to the memory. So maintaining an efficient cache coherency on a single-shared-bus becomes less practical when bandwidth between the processors and shared memory staring to become the bottleneck as the number of processors increases. Non-Uniform Memory Access (NUMA) architecture, another shared memory architecture, divides memory into multiple banks; each assigned to one processor. As the name suggest, physical distance between the processor and the memory changes the access speeds of memory. Processors have faster access to their local bank than remote banks attached to other processors.

Usually every hardware architecture requires modification to existing sequential algorithms, programming model and memory access pattern in order to gain maximum out of the processing power offered by the underlying hardware. Most often mapping sequential program into multiple threads will not gain much as expected. The programming model and style used by the each of these parallel hardware architectures provides a great significance as they provide the interface to utilize the processing power offered by the underlying hardware. So we will discuss the programming models along with each of these architectures.

Heterogeneous computing refers to the use of multiple processor architectures by a single application. By executing "compute kernels", whose computational characteristics fit the characteristics of different architectures than where it is hosted, applications can see large improvements in performance and energy efficiency. Most common heterogeneous platforms include multi-core CPUs, many-core GPUs, FPGAs, or other application-specific hardware platforms.

2.3.1 Multi-Core Processor Systems

As the frequency of uniprocessors is reaching its limits due to hardware manufacture limitations, the researchers and hardware vendors started producing commodity CPUs

that again reflect Moore's law-style scaling, which support multiple and concurrent processing with multi-core/multi-thread architectures. Today, dual-core, quad-core and 8-core CPUs are common in consumer PCs. And the scaling continued with new hardware designs; for example, there are already specialized 64-core processors and ×86-based many-core architectures that contain 64 discrete ×86 cores with vector extensions [39]. These CPUs implement several parallelization techniques to increase performances while giving the impression that they work sequentially: branch prediction, out-of-order execution, superscalar. All these techniques increase the complexity of the CPU, limiting the number of CPUs that can be included on a single chip.

2.3.2 Field-Programmable Gate Arrays

Field-Programmable Gate Array (FPGA) is a semiconductor device with programmable lookup-tables (LUTs) that are used to implement truth tables for logic circuits with a small number of inputs (on the order of 4 to 6 typically). FPGAs may also contain memory in the form of flip-flops and block RAMs (BRAMs), which are small memories (on the order of a few kilobits), that together provide a small storage capacity but a large bandwidth for circuits in the FPGA. Thousands of these building blocks are connected with a programmable interconnect to implement larger-scale circuits.

Because of the ability to customize the hardware through programming to the use case it is used, FPGAs are most used in special use cases where high performance is the major requirement. But the high cost of devices prohibit them from using in general purpose computing use cases.

2.3.3 Accelerator Co-processors

Creating parallel computer systems connecting generic CPUs is the common approach of achieving parallelism. Another approach is to use of special purpose hardware which are suitable for special purpose, as a co-processor. Often these co-processors are called *Accelerators*. GPUs and Cell Broadband Engines are most common of this type. A single core of a co-processor is optimized for a particular task, such as GPUs are optimized for graphics processing, hence is simple in design. So they do not take much area on the chip, where by it enables to put thousands of processing core in a single chip. These cores collectively offer raw processing power that exceeds the thousands of CPUs combined.

2.3.3.1 Cell Broadband Engine

Cell Broadband Engine [40], [41] contains a PowerPC Processor Element (PPE) which is suited for processes requiring frequent thread switching, and 8 Synergistic Processor Elements (SPE) which are cores optimized for floating point arithmetic. These 9 cores are connected using a high-speed bus called the Element Interconnect Bus (EIB), and placed on a single chip.

Using the conventional programming models in multi-core hardware architectures is challenging to effectively exploit the maximum performance of parallelism offered by those architectures. As a solution alternative programming models are developed, like Stream programming model [42], which allows programmers to write programs in sequential style and framework take care of automatic parallelization. Stream programs use *data-flow* programming style, which is a different programming style from traditional *Von Neumann programming model*. The performance benefits of stream programming comes through its characteristics of bulk loading of data into a "local memory", operating on the data in parallel, and bulk storing of the data back into memory. Stream programming model is the preferred programming model for Cell Broadband Engine.

2.3.3.2 Graphics Processing Units (GPUs)

One of the key problem in modern processor design is to overcome the slowness of memory. Even though there are off-chip, low latency and high speed random access memory, these memory technologies are not keeping up with the pace of progress made in the throughput of processors cores. Moreover, as the high volume data streams scenarios becoming common, this limitation between processor and memory has become a huge bottleneck. The time between the issuing of a memory request by a core and the subsequent response from off-chip memory can be very long, up to hundreds or even thousands of processor cycles, and the gap is widening. Increasing bandwidth between memory and the processing cores is a one possible solution, but this dose not reduce the latency of memory fetch. System designers have came up with several solutions to this problem, such as use of fast on-chip cache memory to avoid the unnecessary memory round trips, use of compile-time and run-time prediction and speculation to make sure that required data is already present on-chip when it is needed, and finally, reorder the instruction stream to lessen the impact of memory-related stalls. Implementing above solutions in processing cores to reduce memory latency make the processing cores complex and thus reduces the number of cores in a single die.

The use of Graphics Processing Unit (GPU) as a accelerating co-processor to multi-core CPUs have become arguably the most popular heterogeneous platform configuration

today. There are several reasons behind this, but the most influential reason being GPUs today provide some of the highest performance per dollar and the lowest power consumption per FLOPS of any computing platform. Even GPU initially designed as a specialized hardware platform for graphics and video processing, soon the research community identified its usefulness in other parallel processing problems. So the years of evolution converted the special purpose GPU architecture to a fully programmable general purpose hardware platform. The use of GPUs for general purpose computing (GPGPU) is increased with emerge of GPU programming environments like Nvidia CUDA [43]. Since then, GPUs are used in vast number of places ranging from home personal computers (PC), laptops, gaming consoles to high-end computing clusters, solving many parallel problems in image processing, computer vision, signal processing, linear algebra and graphics algorithms.

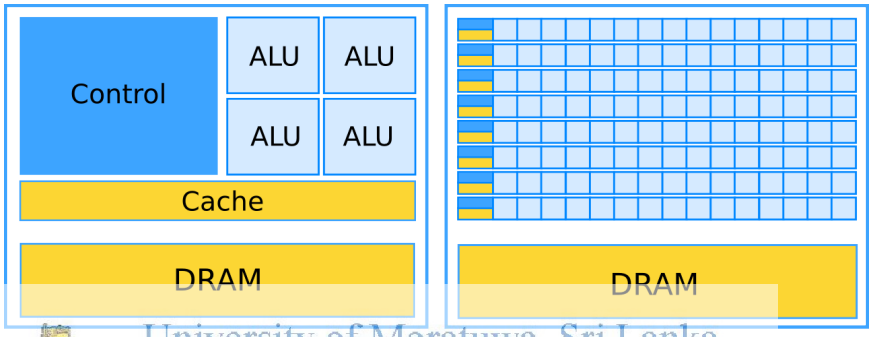


Figure 2.9: CPU and GPU Architecture.
CPU dedicates more transistors to a program control while more transistors are dedicated to the data processing in GPU.

GPU falls under SIMD category in Flynn's taxonomy, where they are applying uniform, moderately complex operations to large volumes of data parallely. This constitutes a special subset of parallel computation, often called data-parallel or "stream" computing [44]. The target workloads for a GPU are much less vulnerable to memory-related stalls as in CPUs. The main target of the design of GPUs is to apply similar operations to large amounts of data, the exact ordering of data is less important. Relaxing the restrictions like this has made the design of GPU processing cores simple and inexpensive, which allows to pack thousands of them in a single chip [24]. CPU dedicates more transistors to a program control than to a data processing. On the other hand, GPU is the opposite, where more transistors are dedicated to the data processing (see Figure 2.9). GPU overcome the memory slowness issues faced by CPUs by introducing much faster main and cache memory hierarchy, which we will explain in next section. But the concept of simple design that governs the architecture of GPUs made this memory capacity per processing core very small compared to what we currently have in CPU systems. But this limitation is alleviated by using

high throughput and massively parallel processing power offered by the GPUs with combining specially designed parallel algorithms.

GPUs first appeared in the 1980s as hardware specially designed to graphics processing tasks. The technology evolve over the next few decades improving its capabilities like floating point arithmetic operations, graphics-related operations and massively parallel operations in the same domain. Over the past few years, a growing community has identified that the GPUs can be successfully used in solving other problems in other domains where they exhibit similar characteristics as graphics processing [22]. General workload characteristics mapped to GPU computing are;

- **Computational requirements are large** - GPUs can deliver an enormous amount of compute performance to satisfy the demand of complex real-time applications.
- **Parallelism is substantial** - The graphics pipeline is well suited for parallelism. The fine-grained closely coupled programmable parallel compute units supports data-parallel problems.
- **Throughput is more important than latency** - GPUs give priority to the high throughput execution of parallel tasks rather giving priority to low-latency execution of single task.

GPUs was initially developed as a fixed-function special-purpose processor with efficient graphics processing as its main goal. The "Graphics Pipeline" is a series of processing steps that create a final picture from set of geometric primitive shapes in a 3-D world coordinate system. Each processing step is fixed in the sense of programability but had the ability to configure. Graphics pipeline has following steps [22].

- *Vertex Operations* - The input primitives are formed from individual vertices. Each vertex must be transformed into screen space and shaded, typically through computing their interaction with the lights in the scene. Because typical scenes have tens to hundreds of thousands of vertices, and each vertex can be computed independently, this stage is well suited for parallel hardware.
- *Primitive Assembly* - The vertices are assembled into triangles, the fundamental hardware-supported primitive in today's GPUs.
- *Rasterization* - Rasterization is the process of determining which screen-space pixel locations are covered by each triangle. Each triangle generates a primitive called a "fragment" at each screen-space pixel location that it covers. Because many triangles may overlap at any pixel location, each pixel's color value may be computed from several fragments.
- *Fragment Operations* - Using color information from the vertices and possibly fetching additional data from global memory in the form of textures (images that

are mapped onto surfaces), each fragment is shaded to determine its final color. Just as in the vertex stage, each fragment can be computed in parallel. This stage is typically the most computationally demanding stage in the graphics pipeline.

- *Composition* - Fragments are assembled into a final image with one color per pixel, usually by keeping the closest fragment to the camera for each pixel location.

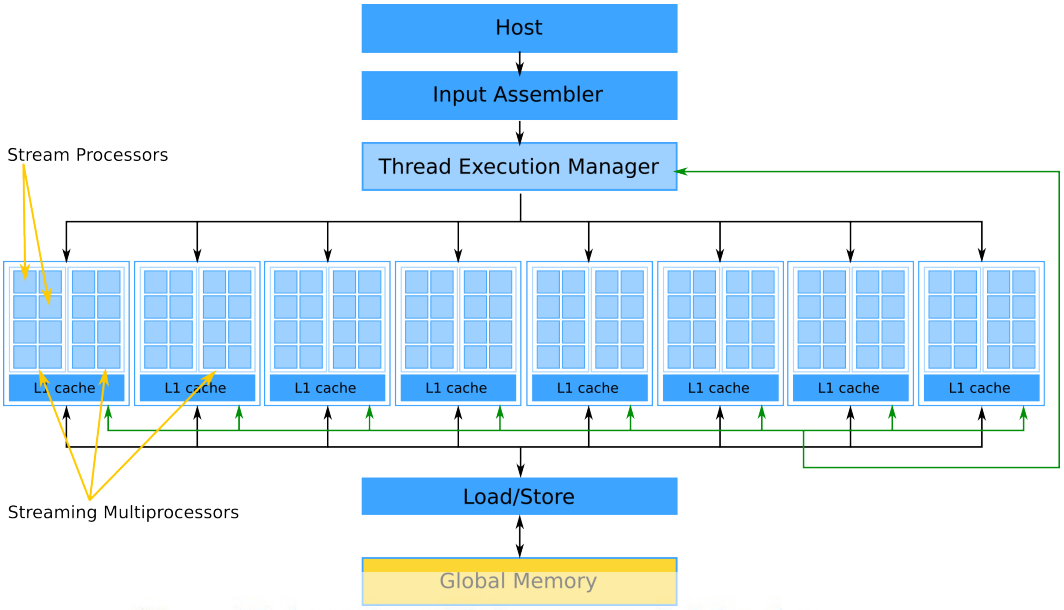


Figure 2.10: Basic modern GPU architecure.
A basic GPU architecture [22] with 16 streaming multiprocessors of 8 stream processors each. One streaming multiprocessors contains shared instruction and data caches, control logic, shared memory and eight stream processors.

The fixed-function processors lacked the customizability of efficiently express more complicated shading and lighting operations that are essential for complex effects. The solution was to introduce per-vertex and per-fragment operations with user-specified programs which are able to run on each vertex and fragment. These user-defined programability replaced the fixed-function processors in GPUs, which paved the way to modern GPU architecture. Earlier GPUs was optimized for graphics pipeline, where it can achieve task-level parallelism by executing different processing stages parallelly and data-level parallelism by executing many threads within each processing stage. When it comes to the modern GPUs (see Figure 2.10), the GPU architects have changed and simplified the design by providing a collection of single fully-programmable hardware unit known as *Stream Processor* and the programs load balance these programmable units in their application in order to achieve either task-level parallelism or data-level parallelism.

There are several GPU hardware vendors in the industry, but the market is most dominated by the GPU products from Nvidia and ATI (acquired by AMD).

2.3.4 GPU Programming Environments

GPU programming model falls under SIMD category. Each instruction is executed on different data items in parallely and usually independent data element cannot communicate with each other. Each processing element can read from shared memory location ("gather" operation) and write back to shared memory location ("scatter" operation). So the processing elements in GPU proceed in lockstep, running the exact same code. Programmers are also possible to write code that dose not follow this principle, where some processing elements follow different branch of the code. But this flexibility come with a price. In GPUs, processing elements are grouped together into blocks, and blocks are processed in parallel. If processing elements follow different control flow within a block, the hardware computes both sides of the branch for all elements in the block. The size of the block is known as the "branch granularity" and has been decreasing with recent GPU generations. So in order to get the maximum performance out of the GPUs, programs must structure to follow same control flow.

Mapping general-purpose computation into GPUs had to follow the same steps and procedures that normal graphics application had followed. The program had to structure in accordance with graphics pipeline even application had nothing to do with graphics processing. The invention of general purpose programming environments on GPUs alleviated these difficulties by providing natural, non-graphics interface to GPU hardware. To exploit this new general purpose hardware architecture, a new breed of application frameworks emerged wrapping the complexity of graphics processing constructs and provide developers with a simplified API. The most commonly used such programming models for GPUs are CUDA [43] and OpenCL [45]. These frameworks provide not only a set of APIs but also collection of tools and libraries to debug GPU applications and extend the capability of GPU processing environment. In this new non-graphics programming model, developers define the computation as a structured grid of parallel threads and the hardware executes the computation unit in load-balanced parallel threads in SIMD fashion. Additionally these programming models allow use of same memory buffer for both reading and writing, which was not allowed in Graphics APIs. Using the same buffer for reading and writing enables new in-place algorithms which has less memory footprint.

2.3.4.1 GPU Programming with CUDA

Compute Unified Device Architecture (CUDA) was first introduced by Nvidia in 2007 on its G80 GPU series. Although CUDA is a vendor specific technology which only supports Nvidia produced GPU hardware, over the use of several years made it the most popular programming model for GPUs with numerous extensions and libraries.

CUDA Programming Model

GPU architecture is built around a scalable array of multi-threaded Streaming Multiprocessors (SMs). Streaming Multiprocessors are used to execute multiple threads in Single-Instruction-Multiple-Data (SIMD) fashion. A parallel task construct in GPU programming model is termed as *kernel* and indicated by a flagged method in the program (with the `__global__` specifier in CUDA-C). In CUDA terminology CPU is termed as *Host* and the GPU is termed as *Device*. The kernel is first distributed to the available Streaming Multiprocessors according to the instructions given by the developer. Developer has the control over how many threads to be run in parallel. The Host and Device have separate memory spaces. So it is necessary to do memory allocation on the device and copy data to the allocated memory. Then the CPU invokes the kernel and once the task complete the result is copied back to CPU's memory space. So the programming in CUDA has following high level procedure.

1. Define compute "kernel"
2. Allocate necessary memory in GPU device.
3. Copy input data from CPU memory to GPU memory.
4. Load kernel code and execute it, caching data on chip for performance.
5. Copy results from GPU memory to CPU memory.

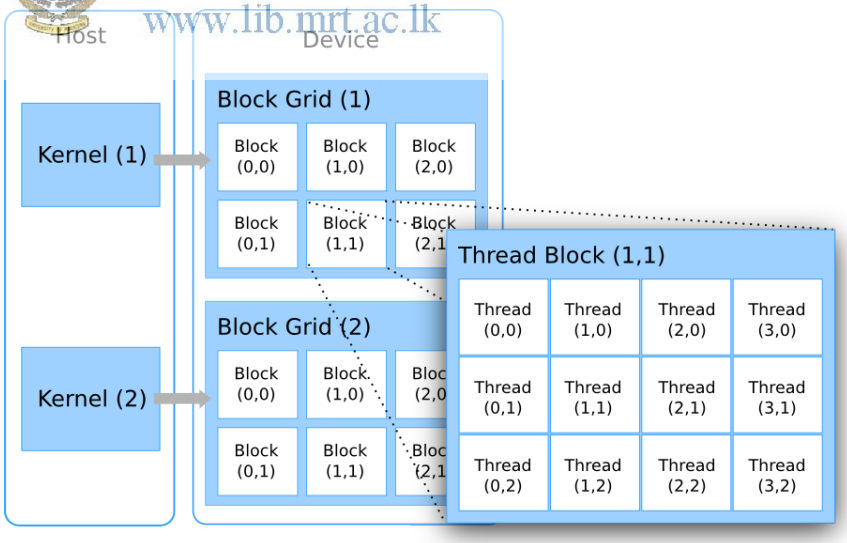


Figure 2.11: CUDA threading model.
CUDA kernel is executed by grid of thread blocks.

A parallel task is sub-divided into sub tasks where they can independently be solved and each sub tasks are assigned to block of threads. Within these block of threads, the

parallel threads are cooperatively solve the sub task. So there can be several thread blocks one for each sub task. Each thread has a unique id and coordinates inside its block. Similarly each block has a unique id and coordinates inside its grids (see Figure 2.11). Depend on the capability of GPU device there are maximum number of threads per single block.

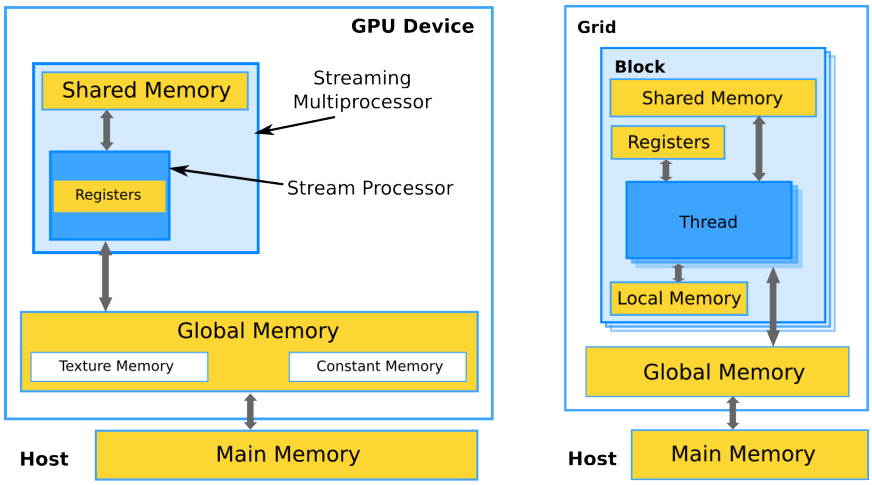
CUDA Thread Model

Usually only one kernel execute at a given time inside the GPU and there can be thousands of threads formed in blocks to run the kernel code. CUDA threads are extremely light-weight than CPU threads, so creating and switching CUDA threads incur only very low additional overhead. Thread within a same block can cooperating with each other when they need to coordinate their memory access through synchronization. CUDA programming models provides "memory barriers" for this purpose. Threads within different thread block cannot cooperate with each other. This characteristic leads programs to transparently scale to any number of processors, because GPU hardware is free to schedule thread blocks on any processor in any order at any time.

CUDA framework provides atomic functions for use in solving some synchronization problems. But these functions needs to be used carefully, because if atomic functions are used badly, they would also have an impact on poor performance.

CUDA Memory Model

CUDA has sophisticated memory hierarchy in order to deliver high performance memory access to parallel threads. Each thread have access to multiple level of memory spaces (see Figure 2.12). A thread have access to its private local memory, shared memory within the same thread block and finally all threads have access to a global memory within the Device. Shared memory is much faster than global memory and shared by threads in single thread block. The constant memory (constant memory cache) and the texture memory are part of global memory. Texture memory actually is a special hardware which is used to access the global memory. Fast memory caches are preferred with small amount of memory requirements when programming in CUDA. But it is important to understand these caches are not coherent. Summary of CUDA memory is shown in table 2.2.



(a) GPU Memory Model (b) Memory Access Model

Figure 2.12: CUDA memory model.
Simplified CUDA memory model. Different kinds of memories have different access models.

Table 2.2: Summary of CUDA memory hierarchy.

Memory	Location	Cached	Access	Scope
Local	Off-Chip	No	Read/Write	One thread
Shared	On-Chip	N/A	Read/Write	All threads in a block
Global	Off-Chip	No	Read/Write	All threads and Host
Constant	Off-Chip	Yes	Read	All threads and Host
Texture	Off-Chip	Yes	Read	All threads and Host

2.3.5 Java in GPU Programming

There is growing interest in using Java for High Performance Computing (HPC) applications, which is based on the its appealing characteristics: built-in multithreading support, object orientation, platform independence, portability, type-safety, security, wide community of developers, and finally, it is use of core training language for computer science students. Apart form other HPC approaches using Java, there are several ongoing research efforts to implement Java support for data parallelism using hardware accelerators and co-processors. In this research our goal is to improve the performance of a CEP engine, which developed using Java, by accelerating it through GPU architecture. So it is important to understand the state-of-the-art in using Java with GPGPUs or any other co-processors.

In this research, our target GPU programming environment is CUDA. So we will first

describe researches that more focused on Java with CUDA. There are other researches that used Java in GPU programming without using CUDA, which we will describe later in this section.

There are three methods to use CUDA enable GPU device with Java language (1) Use Java Native Interface (JNI) which is the most straightforward but also the most complicated way; (2) Use tools which are capable of generating a CUDA code directly from a Java byte-code; (3) Use a Java library which wraps the CUDA Driver API–the lower level approach to use CUDA environment in other language than C/C++.

JNI technology allows calling any native function or library within Java program. It supports implementation of any Java class method in the C/C++ programming languages as well. JNI can be used to invoke CUDA kernels from Java application, but doing so is very hard because of inherent complexity of JNI technology. Even simple operation like copying value of class variable has to be done in about three steps. Further more, using memory pointers is hard with JNI, because Java dose not support direct memory access through pointers. Although it is hard, there is an advantage using JNI for CUDA programming because it access lowest level of CUDA environment bypassing *CUDA Runtime Environment*, JNI gives multi-platform high performance integration with CUDA.

Automatic CUDA code generation methods allow programmers to develop parallel programs for CUDA enabled devices without knowledge of CUDA programming. CUDA code generation is done either extending Java compiler to generate byte-code compatible to CUDA runtime or directly changing byte-code.

Rootbeer [46] is a higher-level Java API for writing parallel programs that can run on GPUs. When developing applications that use Rootbeer API, a programmer creates sections of Java code, which is much like CUDA kernel, that can run on a GPU. Programmer needs not to worry about the serialization and deserialization of data, memory transfers, or kernel creation or kernel launch that are required for a CUDA program. Rootbeer API mimics the Java Runnable API, where programmers have to implement the task of an application which run on a separate thread. Rootbeer generates CUDA code dynamically at the compile time which do the exact same thing written in Java code. Apart from that all the locally defined and used data types including user defined data types are serialized and copied to CUDA device by the Rootbeer at runtime. To improve the performance of serialization and deserialization, Rootbeer generates necessary bytecode for data serialization/deserialization at compile time. Other than dynamic method invocation, Rootbeer supports all the major Java features.

JCudaMP [47] is a research effort to implement OpenMP for Java language. JCudaMP supports both CPUs and GPUs as the back-end for processing. Although it only provides very basic set of features for CUDA back-end, the programing is relatively easy than JNI. CUDA advanced features like shared-memory is not available through JCudaMP, so performance is very poor relative to CUDA-C.

In *java-gpu* [48], they introduce a new annotation called `@Parallel`, where the loops can be annotated as parallel. So these methods annotated as parallel can be offloaded to Nvidia CUDA compatible graphics cards, without explicitly writing any CUDA C codes. The PTX code is generated automatically for annotated method and data transfers are handled automatically.

In [49], authors present a programmer-friendly API for accessing CUDA devices from Java. The research is mainly for developers who may be familiar with Java and CUDA but not with JNI. In order to use the work they have published, a developer needs to write CUDA code and Java code himself with using provided API and the glue-code will be generated using JNI for Java to CUDA communication. This research is not practically usable since there are no any publically available code or library for this work.

CUDA Driver API wrappers do not provide low level access to CUDA as in JNI nor it provide high level API as in JCudaMP. These wrappers provide middle ground of low level access and higher level API. Some implementation uses both CUDA Driver API and part of CUDA Runtime API.

The most widely adopted CUDA Driver API wrapper for GPGPUs is *JCuda*, which is available as free and open-source library [50]. JCuda, first developed in 2009 as a simple java wrapper around CUDA SDK, to help Java developers to easily program parallel applications for GPGPUs. After several years of progress, JCuda now translate CUDA methods into Java methods and introduces Java objects that are CUDA specific. For example JCuda has introduced device Pointer object, `cudaMalloc` method for memory allocation or `cudaFree` for memory freeing. Call to such Java method invokes corresponding CUDA method through the wrapper. Basically programming with JCuda is similar to programming with CUDA Driver API. JCuda is not only a API wrapper, but it also contains bindings for CUDA libraries like CUBLAS or CUSPARSE as well. Although JCuda is not 100% feature complete as CUDA, it resembles almost all required core CUDA constructs and method calls. The project contains proper documentation through its official site [50].

jCUDA [51] (Java for CUDA) offers CUDA bindings for the Java language with double precision operations and object oriented programming for CUDA. This project has similar goals as JCuda, but its object oriented design enables programmers to write

clean codes with exception handling. This is available for free for both commercial and academic purposes but the project is not maintained now.

There are other research projects like *JaCuda* [52] and *jacuzzi* [53] which have the similar goals, but not currently maintained.

The lack of publically available performance metrics for CUDA implementations on Java language makes it harder to developers to use CUDA with Java in production grade applications. The available performance metrics [49], [54] are few years old and do not cover newer features introduced to CUDA enabled GPUs.

Open Computing Language (OpenCL) [45] was developed by Khronos Group consortium (Intel, AMD, NVIDIA, and ARM) and released in 2009. It aims at supporting more hardware and to provide a standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. The table 2.3 summarizes comparison of basic features between CUDA and OpenCL [55].

Table 2.3: Comparison between CUDA and OpenCL.

Trade-offs	**CUDA**	**OpenCL**
Kernel code	Simple	Simple
Kernel setup	Simple	More complicated
Portability	Low	High
Library availability	High	Low

There are few research approaches [56], [57] who have used CUDA in Java based projects to gain high performance through parallelism. Most of them being recently published research shows that there a growing interest in research community to accelerate object oriented languages using GPUs.

In *OpenJDK Project Sumatra* [58], they try to improve the performance of Java applications by taking advantage of GPUs and Accelerated Processing Units (APUs). Sumatra allow Java Virtual Machine's JIT compiler to generate GPU ISA code directly. This would allow the JVM to target code which seems suited to GPU offload. The initial research focus is to improve the performance of JVM itself by enabling code generation, runtime support and garbage collection on GPUs. Using GPUs for Java applications will be done once JVM has the capability to use GPUs. Moreover there is ongoing research announced by IBM [59] which will enable existing IBM Java runtimes for server-based GPU accelerators and explore acceleration in ordinary workloads under existing APIs. If implemented in other popular Java runtimes like Oracle JDK for commodity GPU products, this approach will allow millions of Java developers to

accelerate a broad range of applications using GPU accelerators, and achieve speedups that will dramatically improve the capabilities of the Java applications.

In [60], authors describe a Java Byte-code Execution Environment called *JaBEE*, which supports common object-oriented constructs such as dynamic dispatch, encapsulation and object creation directly on GPUs. JCuda, JavaCL and OpenCL do not allow object oriented processing on device side, which is a necessary requirement if object oriented languages are used with GPUs. The main goal of that research is to evaluate the challenges, limitations and opportunities associated with running byte-code languages directly on the GPGPUs.

There are other several implementations of Java bindings for OpenCL like *JoCL* [61] and *JavaCL* [62]. On the other hand, the open-source project *Aparapi* [63], developed at AMD, facilitates execution of Java programs on OpenCL devices. Aparapi is not a binding for OpenCL, instead it provides JIT compilation of Java byte-code to OpenCL kernels at runtime.

2.4 Related Work

As we reiterated throughout the literature review, combining CEP and parallel hardware is a challenging task. Successful use of parallel hardware with right design and algorithms will increase the performance of CEP implementations. There are a few researchers who have tried this approach in the past. In this section we describe the research work of those related researches.

There are three approaches researchers have tried to improve the performance of CEP implementations.

1. Distributed processing with CEP agent network
2. Parallelizing processing construct to multiple threads on multi-core CPUs
3. Parallelizing data parallel operations in many-core systems (GPUs)

Although our primary interest in this research is to explore the possibility of using many-core parallel hardware architectures in improving CEP performance, we will also list related researches done in other two approaches. Because concepts in those researches can be borrowed to our research.

Related research in using parallel hardware technologies to improve the CEP processing performance can be categorize into two broad sections; parallel hardware used in complex event processing and general event processing implementations and parallel hardware used in pub-sub systems. Apart from that we will also refer to other parallelization and optimization techniques used in CEP to improve performance.

2.4.1 Event Processing on Parallel Hardware

Research work by Cugola and Margara [26] is the first publicly available work done in exploring the use of parallel hardware in event processing domain. Although some commercial CEP vendors [15] claim they have implemented some part of their CEP engines on top of CUDA environment, but there are no any published work on these implementations.

Cugola and Margara has investigated on use of parallel hardware, specially GPUs, to improve the performance of CEP implementations. They have proposed and developed few parallel algorithms which speedup event processing in their CEP implementation called T-REX [17]. They have developed algorithms for both the multi-core CPUs using OpenMP and GPGPUs using CUDA. The main objective of their research was to increase the performance of common CEP operators, like filter operator, sequence operator, and aggregate operator, that are supported by most of the Query Languages. First, they have defined the meaning of these operators unambiguously using a query language call "TESLA [30]" which they have developed as part of their T-REX CEP engine. Since most of current CEP implementations follow similar meaning to common operators as defined by TESLA, they believe their research would benefit by other CEP implementations too. Since SiddhiQL and TESLA have similar meaning in their common CEP operators, we believe we also can adapt some of their research work in our design.

Most CEP implementations use *automata* for model sequencing rules. Even Siddhi has adopted this approach. In automata model, intermediate results are stored until final composite event pattern is detected. But in paralel hardware like GPUs, it is hard to implement automata efficiently, because each automata is different from one another and require different processing.This characteristic poses a great overhead to CPU-GPU communication and quite inefficient to implement in GPUs. So authors have developed a novel algorithm called *Column-based Delayed Processing (CDP)*, where events are just stored in a column based data structure with in the CPU memory until the *terminating event*' for the processing query is received. By the term terminating event they meant the possible last event of a sequence pattern to finalize the detection. The processing of events started only after a terminating event is received.

When processing according to CDP algorithm, there are separate column data structures for each event type with specific condition and timing constrains, and incoming events are processed to check for condition attributes and stored in relevant column. Once a terminating event is received (event C in this example), the CPU starts processing and copy each columns one by one to GPU for processing.

For example see the following query for sequence for three primitive events; `A -> B -> C`.

```
define ComplexEvent()
from C(p=$x) and each B(p=$x and v>10) within 8 min. from C and
last A(p=$x) within 3 min. from B
```

Listing 2.4: TESLA languange query for sequence pattern.

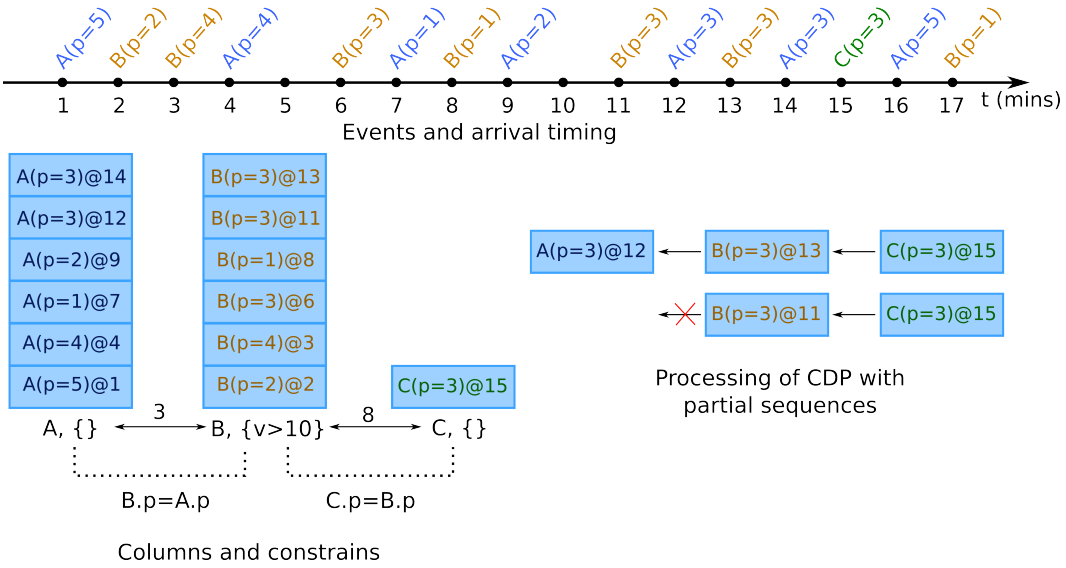


Figure 2.13: CDP algorithm data structures and processing for listing 2.4.

CDP algorithm has separate columns for each primitive events with parameter constrains. Events are stored in columns and processing is delayed until terminating event is arrived.

CDP creates three columns for storing events in three primitive event types (see Figure 2.13). One column for primitive event `A` with no parameter constrain, one column for `B` with parameter constrain `v>10` and last column for `C` without any parameter constrain. All three event types have relationship constrain where parameter `p` of every event should have same value in the sequence pattern. Moreover, the timing constrains enforces event type `B` should happen within 8 minutes of event type `C` has occurred and event type `A` should happen within 3 minutes of event type `B` has occurred. The last column has space for only one record, as it represents the terminating event and the processing starts on the arrival of first terminating event.

Processing is done column by column starting from last to first (`C to B to A`). At each column, first they are deleting events which do not adhere to timing constrains, there by decreasing number of events to process. Then they process each remaining events to get valid sequences who match parameter constrains. Each matching event is used to create partial sequences, which is a list of matched events so far. This partially matched sequences are used to do the matching process of next columns. This process continues until the first columns or there are no matching events in next column. Finally, the set of partial sequences are used to create new resulting composite events.

Implementing CDP algorithm for GPUs is done using CUDA. Authors have divide the computation into two parts where operations that do not gain much performance form GPUs are done directly on CPUs, like deleting expired events in each columns. Those operation executed on CPUs also need to follow strict sequencing of operations, hence doing them in GPUs may require synchronization between threads, which is quite inefficient. Operations which is possible to do in parallel, like matching individual events in each column with current partial sequences, are done in GPUs.

In their approach, authors have selected several queries with different complexities and developed two algorithms for each selected queries, one for multi-core CPUs (Automata-based Incremental Processing (AIP) algorithm) and other one for both multi-core CPUs and GPUs (CDP algorithm). Then they have analyzed the performance of three implementations under different workloads to compare each other. Their result shows that under various workloads, GPUs have performed well than other two implementations, when there are complex rules associated. The CDP algorithm implementation on GPU has shown a speedup of 25 compared to CPD implementation on multi-core CPUs. As they have mentioned in the paper the GPU implementation leads to an average speedup of 40 with their hardware configuration. On the other hand, multi-core CPUs have performed well in simple rule scenarios. Authors have concluded that GPUs should use in cases where complex queries are associated with high volumes of data and multi-core CPUs should use when queries are simple and data load is moderate.

Streaming aggregation is one of the performance-critical core operation in distributed stream computing domain. It is also largely related to event processing where stream aggregation dose the summarization of incoming data in a stream, for example calculation of averages in high frequency trading applications. Complex event processing engines have a separate event aggregation phase and in our research aggregation is one of the aspect we are trying to improve the performance. Streaming aggregation is a data parallel use case and largely affected by the data transfer overhead than the actual computation, hence improving the performance (both data transfer and computation) of this has a positive effect on overall performance of the CEP engine.

In [27], they research on how the streaming aggregation can be effectively implemented on different parallel hardware architectures. In their research, they investigate how this data parallel task is behaving in three different parallel hardware architectures; multi-core CPUs, GPUs and Cell Broadband Engine architecture. They have implemented three algorithms in three architectures for a case study involving streaming aggregation and compared the performance metrics. The three parallel architectures that used in this research covers the two extreme ends and the middle ground of multi/many core architectures, where Intel Core 2 Quad system which its in one end of the spectrum and Nvidia GeForce GTX 285 GPU in the other end of the spectrum and Cell Broadband

Engine which covers the middle ground of above two ends.

The case study they used is a live stock market data analysis system to discover "bargain" purchases where the current asking price for a stock is less than the volume-weighted average price. The simulated data feed contains a set of stock trades and quotes for over 2000 symbols from about a months data feed. The aggregation occur when calculating the volume-weighted average prices for each symbols in the simulated feed. The interesting fact about the results of this research is what they have discovered in their GPU based implementation. They have implemented three variations of same algorithm using CUDA for GPUs. Both algorithms transfer data to GPU memory before starting GPU computations and only the result data is transferred back to main memory. The first variation of CUDA algorithm uses synchronous bulk communication between the host and the GPU memory. The second algorithm uses asynchronous bulk communication while the third algorithm dose many small transfers of data to GPU memory. The first two synchronous and asynchronous algorithms transfer the whole data set in one copy to the GPU, where the asynchronous algorithm uses GPU hardware provided independent memory controller. The independent memory controller supports copying of data without intervention of computation hardware in GPU, so the data transfer time can be overlapped with the actual computation time by sending data set for future calculations while GPU performs calculation for current data set. In last algorithm, they have copied the data set in small batches with maximum transfer count limitation. Overall performance comparison shows the Cell Broadband Engine architecture out performs the other implementations even though GPU has more computation capability than Cell architecture. This is because Cell has direct access to main memory like multi-core CPUs have, so no need of data transfer as in GPUs. Within the three implementations of GPU algorithms, the synchronous and asynchronous algorithms performed equally when the size of the data transfer is small and has increased the overall aggregation time when the data set size increasing. The fine grained memory transfer algorithm has more overall aggregation time than other two algorithms when data set size is small but it out performs the other two algorithms when data set size increases.

In [64], authors have discussed possible use of GPGPU technology for parallel event processing with CUDA or OpenCL. They have mentioned that it is possible to outsource highly computation intensive tasks like pattern matching in string data to libraries which use GPUs for processing [65].

2.4.2 Pub-Sub Middleware on Parallel Hardware

As we have explained in Section 2.1.2, complex event processing has its roots in publish/subscribe systems, where pub-sub systems do relatively simple content based

matching of incoming data. Hence any improvement done to increase the performance of pub-sub systems are very important in this research because there is a possibility these improvements can be applied to the CEP systems.

In [28], they are trying to improve the performance of content based matching in event dispatching component of pub-sub network. In pub-sub style middleware systems, there are are set of event brokers inter connected with each other and each event broker subscribe to upstream event brokers for receive events with interested event attribute patterns (see Figure 2.14). These interested event attribute patterns are based on the subscriptions it receives from its downstream clients. In a practical scenario the number of subscriptions received by an event broker can be as high as over thousand subscriptions. So this event matching over incoming high speed event stream can easily become the bottleneck of the whole pub-sub system. They have proposed a CUDA based parallel algorithm for GPUs to increase the throughput of event matching over set of subscribed filters. The proposed algorithm, called "CUDA Content-based Matcher (CCM)", is then evaluated against the currently existing most efficient solution for content based matching called "SFF [66]".

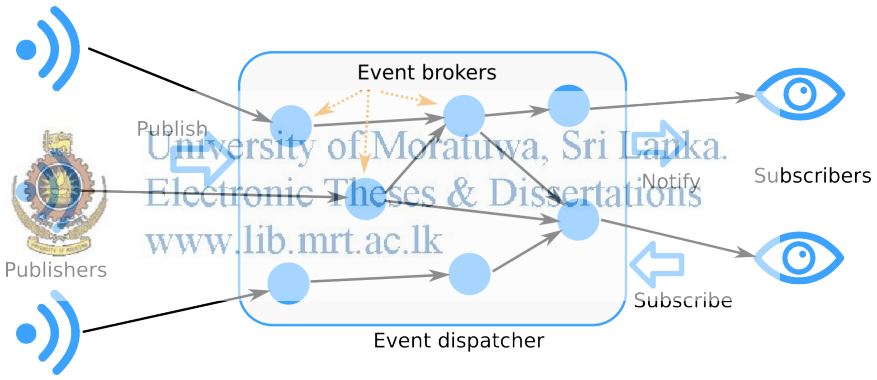


Figure 2.14: Publish-Subscribe network.
Publish-subscribe network consist of event brokers who do the event routing based on the registered subscriptions.

In the event matching component of the event broker, whenever an event is entered to the component, usually it is matched against the available filters sequentially. In this research the algorithm they have purposed do the matching of single event over all available filters in single pass parallely using larger number of threads in GPUs. The CCM algorithm consist of two phases: a constraint selection phase and a constraint evaluation and counting phase. The available subscriptions are relatively static over the period of time, so in their algorithm they have encoded the available subscriptions in special data structures and stored permanently in GPU memory (see Figure 2.15). Each incoming event is encoded as set of attributes and the filters are encoded as conjunction of attribute constrains. For example an event with weather sensor data is encoded as `e = [ area, ‘‘area1’’ , temp, 25 , wind, 15 ]`, while

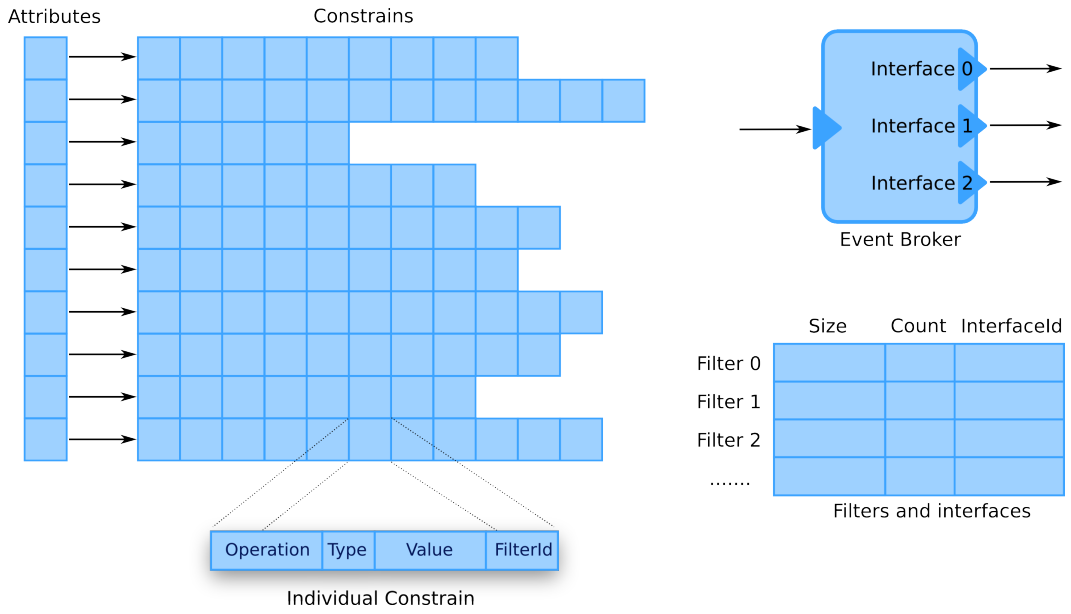


Figure 2.15: CUDA Content-based Matcher (CCM) algorithm data structures. CCM algorithm uses data structures in GPU memory to encode the available filters and their individual attribute constrains.

a filter for weather data is encoded as `f = (area = ''area1'' AND temp > 30)`. When two filters are joined with disjunction (`p = [(area = ''area1'' AND temp > 30) OR (area = ''area2'' AND temp < 60)]`) they are called *predicates* and treated as two filters where at least one filter should be matched to satisfy the predicate. Each attribute constrain in the filters are grouped by the attribute id and stored in constrain table. Each cell in constrain table consist of the operator, value and the filter id of the particular constrain. The operator can be any of `==, >, <, <=, >=, !=`. The filter table consist of the size of total constrains that should be matched in order to satisfy the filter, the number of constrains matched in current event processing cycle and the interface id which the matching event should be forwarded.

When an event come to the broker, it first encode its attributes as a set of key value pairs in CPU level for ease of processing. Then this event representation is transferred to GPU memory. In GPU, before start processing an event, the count values in filter table is reset to zero. There are as much as thread invoked in GPU to process all cells in constrain table at once. Each thread reads the relevant attribute from the event, if it is there, and check if that attribute value match the constrain represent by that thread. If so increment the count value of filter pointed by the filter id in the constrain. This increment is done in synchronous manner. Once all the threads are done, the filter table is processed by pre-assigned set of threads to check if the count value is equal to size value, which means the filter has matched. The matched filter ids are sent to CPU, where the relevant event is forward through respective interfaces.

The implementation of algorithm has done several optimizations in CUDA to reduce the memory transfers between CPU and GPU. The performance results from the research shown that CCM algorithm can gain speedup from 7x to 13x when there are over one million constrains. Further, results shows the speedup increases when the number of attribute in the events increases and number of constrain per filter increases.

The first research on CMP to improve the Event processing performance [67]

2.4.3 CEP on Other Hardware Architectures

Apart from using the commodity parallel hardware architectures for improve the performance of CEP implementations, there are several researches [68]–[71] which have used re-configurable hardware technologies like FPGAs. The characteristics of FPGA hardware enables implementation of CEP engines to specially optimized for the use cases they are used in. So in FPGA-based solutions its more about choosing the trade-off between the degree of parallelism versus the desired application requirement. The drawback here is that although FPGA hardware offer very fast processing power, they are expensive and not widely used in the industry, while the GPU based technologies are cheap and more accessible for programmers.

Chapter 3

Siddhi GPU Query Processors

Siddhi implementation has a high-level processing entity called "QueryRuntime" which encloses all the processing constructs of particular event query. There are separate QueryRuntime instances for each event query defined in Siddhi CEP instance. Our proposed approach implements a new type of QueryRuntime called "GpuQueryRuntime" which follows the same programming API as the QueryRuntime, but internally it off-load event processing to configured GPU devices. The high level architecture of the proposed solution is shown in Figure 3.1. Users can easily choose which QueryRuntime to use just by changing event query definition.

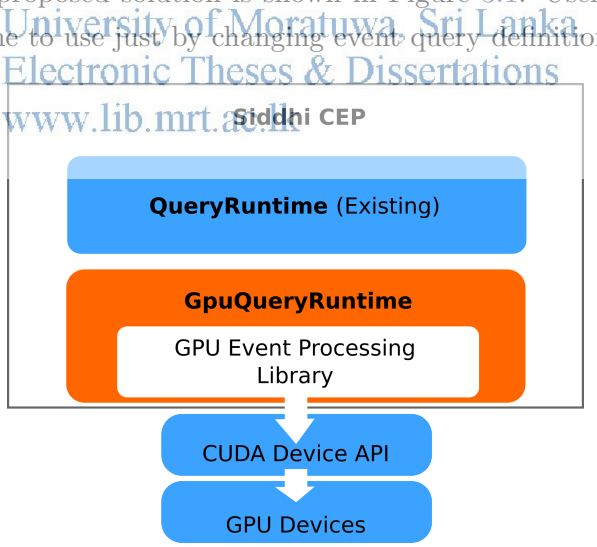


Figure 3.1: Higher Level Architecture of Proposed Solution.
Proposed solution consist of a new QueryRuntime for GPU event processing and general purpose event processing library.

To decouple low level implementation of GPU communication and kernel handling from Siddhi code changes, we have developed a GPU event processing library that manages all the low level GPU tasks. This library is accessed through a well-defined, simple programming API.

This chapter provides in depth details of new Siddhi GpuQueryRuntime and how it interact with the event processing library. Next chapter will detail on the implementation of the event processing library. In Section 3.1, we present overview of the current architecture of Siddhi CEP and we describe in-depth the internal implementation details of complex event processing operators that we are trying to improve in this research. Section 3.2 describes our proposed changes to Siddhi architecture to support GPU event processing while Section 3.3 explains on implementation of proposed GPU query processing runtime.

3.1 Siddhi CEP Architecture

Before describing the details of the changes we proposed to existing Siddhi architecture, it is important to understand the internal details of current architecture. As we have described in Section 2.2, Siddhi has four main components: i) input adapters, ii) Siddhi-Core, iii) output adapters, and iv) query compiler. We will detail on Siddhi-Core and its internal implementation details in this section.

The latest release of Siddhi CEP engine (Version 3.0.0) have introduced some significant changes to the Siddhi architecture that largely improve the overall performance of the engine. Use of Disruptor Queue [72] as the main event data queuing mechanism is one of these changes which contributed to increased parallelism and performance in new version of Siddhi. Earlier version of Siddhi CEP (Version 2.0.0) used readily available Java `java.util.concurrent.BlockingQueue` to decouple data adapters and event processors. When the incoming event load is very high, the performance of BlockingQueue degrades drastically.

Our work is based on Siddhi 3.0.0 version. Architectural changes introduced in Siddhi 3.0.0 make it easy to implement alternative query processing runtime as we have done in our research.

3.1.1 Siddhi Architectural Components

As explained in the literature review chapter, Siddhi supports both single-threaded and multi-threaded event processing. Which threading mode should be used in a particular query can be configured at query definition level. It is very important to understand how Siddhi CEP engine creates, allocates and utilizes Java threads in each of its processing constructs, and how inter-thread communication happens within Siddhi CEP engine. Because our suggestions to improve query processing performance in Siddhi is based on the opportunities and limitations of this existing threading model.

When describing Siddhi query processing elements we use symbolic representation for each processing elements as shown in Figure 3.2.

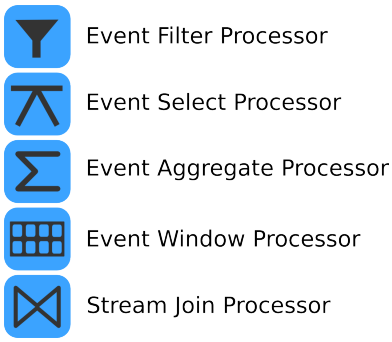


Figure 3.2: Siddhi Query Processors.
Symbolic representation of Siddhi Query Processors we used in other figures.

Whenever a user wants to use Siddhi for an event processing task she has to define a "Execution Plan" for that particular task. An execution plan contains a set of stream definitions, a set of stream partition definition and a set of event queries. Inside Siddhi CEP execution plan is represented by a runtime Object called `ExecutionPlanRuntime` which wraps all the `StreamDefinitions`, `QueryRuntimes`, `PartitionRuntimes` and `StreamJunctions` for that particular execution plan. `StreamDefinitions` contains all the meta information about each an every input event streams, like their attributes and their types. `QueryRuntimes` contain all the query processing runtime constructs created by the `SiddhiCompiler` according to user defined event query. `StreamJunctions` contain all the input and gateway points of each and every streams created for that particular ExecutionPlan.

For example, consider the SiddhiQL query defined in Listing 3.1. It has a stream definition for "StockStream" event stream and event query for filter out some events matching to given criteria. The meta information of the execution plan and the query are given using "Query Annotations". Query annotations are a new feature introduced in latest Siddhi version and has been a very useful feature for our proposed query runtime implementation. More details on how we used query annotations will follow in later sections. Here in this execution plan we have used Query annotation for define execution plan name (`@plan:name('FilterQueryPlan')`), query name (`@info(name = 'query1')`) and to inform ExecutionPlanRuntime to use muti-thread mode to process this query (`@plan:parallel`).

```
1 @plan:name('FilterQueryPlan') @plan:parallel
2   define stream StockStream(symbol string, price float, volume
      long);
3
4 @info(name = 'query1')
5   from StockStream[symbol == "GOOG" AND price > 100.5]#window.
      time(5 min)
6   select price, volume, count(price) stockCount
7   insert into GoogleTopStockStream;
```

Listing 3.1: Siddhi ExecutionPlan.

Each event query defined in the execution plan will create a processing construct called `QueryRuntime`. QueryRuntime contains all the runtime construct that needs to process incoming events from the defined input event streams. The most notable runtime construct in the QueryRuntime is called `StreamRuntime` which is an abstract implementation for different types of StreamRuntimes like `SingleStreamRuntime`, `JoinStreamRuntime` and `PatternStreamRuntime`. StreamRuntime has a `StreamReceiver` and a set of Event Processors organized as a chain according to their processing order.

Each of the streams defined in the execution plan has an `InputHandler` and a `StreamJunction` (see Figure 2.3). StreamJunction act as a hub for events going out and coming into stream. It employees publisher-subscriber mechanism using Disruptor queue to receive and send events to a particular stream. All the events coming from outside world feed into CEP engine via InputHandler and publish into Disrupter queue. `StreamReceivers` subscribe for events in Disruptor queue. Once an event or a set of events received from Disruptor queue, StreamReceiver do necessary conversion and forwards it to first event processor in event processor chain. As shown in Figure 3.3, once an event or a set of events received to the first of event processor chain, they flow through every event processor in the chain one by one until the end of the chain. Some events may discarded by event processors like Filter Event Processor and some events get enhanced by adding new attributes by event processors like Event Aggregate Operator. Once the event reached the last event processor called `QuerySelector`, it will convert according to the output event stream definition and published to output event queue. Output event queue is also a StreamJunction with a disruptor event queue. All the subscribers for output event stream should subscribe for this StreamJunction.

Depend on the threading model used (i.e., single-threaded or multi-threaded) and the number of distinct input event streams, QueryRuntime organization can be differ. As shown in Figure 3.4-(a), single event query configured to use single input event stream is the simplest form of QueryRuntime organization.

If there are more than one query defined in a ExecutionPlan and each of these queries

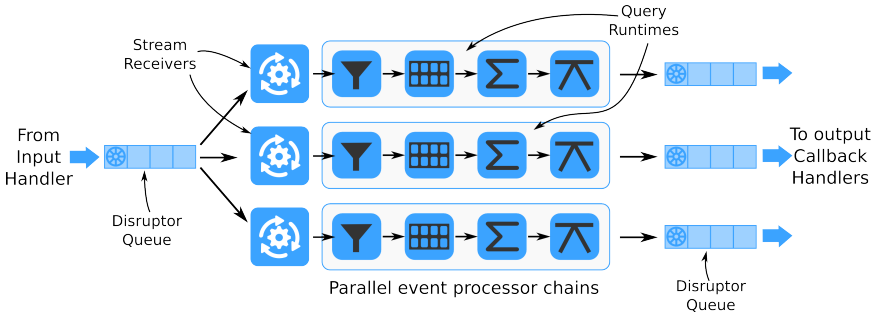


Figure 3.3: Siddhi Event Processing Constructs.
Siddhi event processing constructs and event flow through the system.

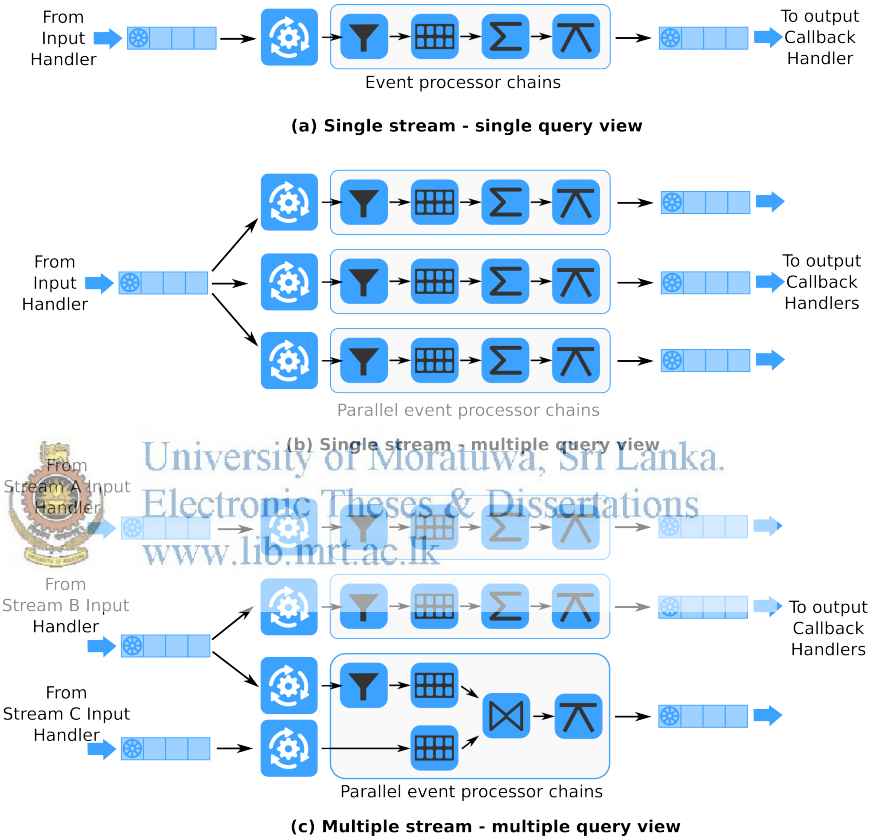


Figure 3.4: Siddhi QueryRuntime organization based on threading model and input stream count.
Depend on the threading model (single-threaded or multi-threaded) used and the number of distinct input event streams, QueryRuntime organization differs.

uses the same input event stream, then the QueryRuntime organization is as shown in Figure 3.4-(b). If the threading model used is single-threaded processing, then when an event received from the disruptor queue, it will be processed sequentially by each event processor chain in same thread. If the threading model is multi-threaded event processing, then the event is processed by each event processor chain in different threads.

If the ExecutionPlan defines multiple distinct input event streams and multiple event queries for these input streams (see Figure 3.4-(c)), then in the single-threaded processing, each event received form input stream is processed sequentially by each event processor chain in same thread. If the threading model is multi-threaded event processing, then the event is processed by each event processor chain in different threads.

3.2 Siddhi GPU Query Runtime

Our proposed approach for improving Siddhi query processing performance is to implement a new Siddhi query runtime that utilizes both the CPU and GPU devices for query processing. We call this new query runtime `GpuStreamRuntime` and it follows the same interface as the other implemented query runtimes like `SingleStreamRuntime` and `JoinStreamRuntime`. In fact there are two implementation of `GpuStreamRuntime` for single-streams and event-join-streams. More details on this follows.

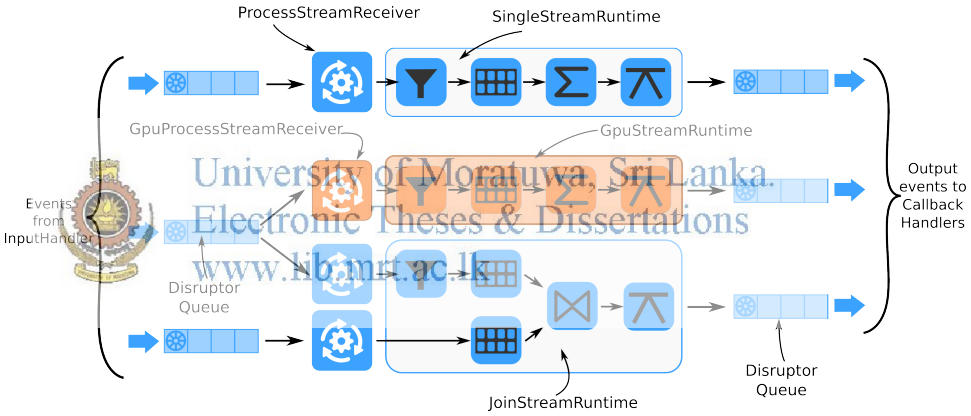


Figure 3.5: Proposed solution for GPU event processing.
GpuStreamRuntime implements the same interface as other query runtimes but internally uses both CPU and GPU devices for query processing. Existing query runtimes can coexist with GpuStreamRuntime and they can interact with each other within a ExecutionPlan.

As shown in the Figure 3.5, `GpuStreamRuntime` is only processing events for one particular query. There can be as many number of `GpuStreamRuntime`s in a particular execution plan and this number is highly depends on the available GPU device count and their capabilities. User can configure to use `GpuStreamRuntime`s and other query runtimes for different queries in same execution plan. These query runtimes can coexist with each other and also they can interconnect to process output event stream of each other (see Figure 3.6. So it is possible to use existing query runtimes which entirely utilizes CPU for query processing and `GpuStreamRuntime`s which utilizes both CPUs and GPUs for query processing, in one execution plan that has very complex

set of event processing queries. User can configure queries that need high performance processing but less memory requirement for event storage to run on GPUs and rest of the queries run on CPUs. With our proposed solution Siddhi will create necessary runtime constructs to process events in harmony with CPU and GPU devices.

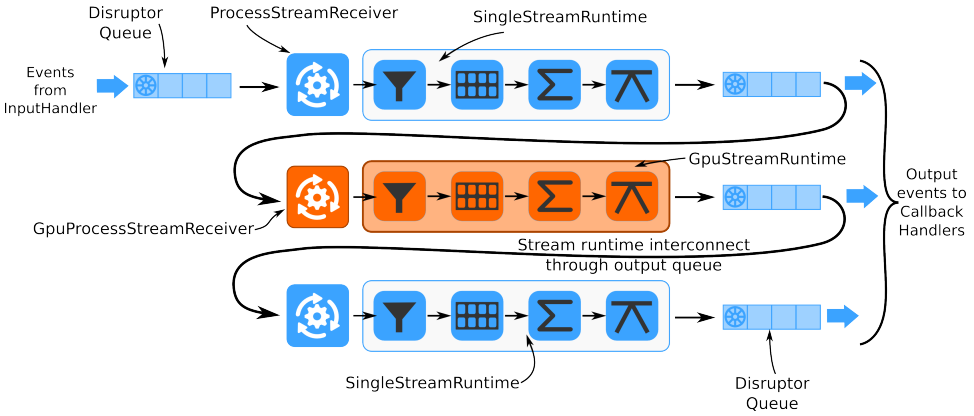


Figure 3.6: Interconnecting GpuStreamRuntime and SingleStreamRuntime with output event queue.
GpuStreamRuntime can subscribe to output events from SingleStreamRuntime and SingleStreamRuntime can subscribe to output events from GpuStreamRuntime.

```
1  @plan:name('GoogleQueryPlan') @plan:parallel
2  define stream StockStream(symbol string, price float, volume
       long);
3
4  @info(name = 'query1')
5  from StockStream[symbol == "GOOG"]#window.time(5 min)
6  select price, volume, count(price) stockCount
7  insert into GoogleStockStream;
8
9  @info(name = 'query2') @gpu(cuda.device='0', batch.max.size='
       2048', batch.min.size='1024', block.size='128')
10 from GoogleStockStream[stockCount > 100 AND volume > 10000]
11 select price, volume
12 insert into GoogleTopStockStream;
13
14 @info(name = 'query3')
15 from GoogleTopStockStream#window(5 min)
16 select "GOOG" as symbol, price, volume
17 insert into GoogleFiveMinuteTopStockStream;
```


Listing 3.2: Using Query Annotatinos to configure GpuStreamRuntime.

We use Siddhi Query Annotation feature to configure `GpuStreamRuntime` at the query definition time. Example SiddhiQL query definition with query annotation to use `GpuStreamRuntime` is shown in Listing 3.2. If a query is tagged with @*gpu* annotation, Siddhi runtime will create `GpuStreamRuntime` instead of other query runtime. @*gpu*

Table 3.1: Siddhi Query Annotations for GpuStreamRuntime.

Annotation	Description
cuda.device	Which CUDA device should be used by the `GpuStreamRuntime` to process events for the assigned stream. In a environment where multiple GPU devices are attached to the server, by using this annotation users can select appropriate CUDA device. And this can also be used to manually load balance queries which use `GpuStreamRuntime`.
block.size	Number of GPU threads allocated for one thread block (thread block dimension). This annotation should be set depend on the CUDA device and its capabilities. More details on how this value is used will be described in Chapter 4. If not set this will be defaults to 128 threads.
batch.min.size and *batch.max.size*	The input events for `GpuStreamRuntime` are grouped to form a batch of events. Because it is very efficient to process batch of events in `GpuStreamRuntime` than processing individual incoming events. `GpuStreamRuntime` has to transfer all these events into GPU device memory and if we do so for each and every incoming events by calling CUDA memory copy API calls (`cudaMemCpy`), there will be lot of CUDA API calls and overhead for API calls will be significant. To reduce this we batch incoming events and copy them once to the GPU memory. In GPU these event batch is processed in parallel at once and copy back resulting events to host memory. The minimum and maximum number of events for a batch is configured via these two configurations.
string.sizes	When copying events to GPU we should know size of each event beforehand to pre-allocate required memory in GPU device memory space. If attributes of an event have primitive data types like integers and floats we can easily calculate size of an event. But if there are text attributes with varying lengths (like email body text or stock symbol name) we have to define maximum possible size for that particular attribute. This is done via this configuration. Users can define maximum size for each attribute like $string.sizes =' symbol = 32, emailBody = 100'$.
work.size	When we process events in GPU we create optimum number of threads suitable for the task. And we can configure how many events should be processed by each created GPU thread to increase parallelism. This is done via this configuration. If for example user define `work.size='100'`, then each GPU thread will process 100 events from its assigned event batch. More information about this will be explained in Chapter 4.

annotation can be more elaborated using annotation configurations.There are currently six annotation configurations are defined for `GpuStreamRuntime` and what each of these annotations mean is detailed in Table 3.1.

3.3 Internals of GpuStreamRuntime

The `GpuStreamRuntime` has six major functionalities (see Figure 3.7).

1. **Create and configure GPU event processing constructs**
   User defined SiddhiQL queries will be compiled and fed into Siddhi processing engine by Siddhi Query Compiler. These compiled queries have all the necessary meta information about each event streams, each defined partitions and each defined event queries. These meta information can be programmaticaly accessed using Siddhi Query API. Once `GpuStreamRuntime` received this compiled query information, it will call necessary GPU event processing library API functions to define and create processing constructs in GPU memory. If for some reason `GpuStreamRuntime` could not initialize properly, then we will fall back to other existing query runtimes.

2. **Input event batching**
   As defined in the GPU query annotations, we use input event batching to reduce CUDA API call overhead and improve event processing throughput. Batching many small transfers to GPU memory into one larger transfer performs much better because it eliminates most of the per-transfer overhead. Events received through Disruptor queue are in the form of batch of events. But the size of these event batches can vary drastically and some times it may contain very low number of input events. So we have implemented minimum and maximum limits for the number of input events in a batch (`batch.min.size` and `batch.max.size`). Inside `GpuStreamReceiver`, a specialized implementation of `StreamReceiver` for GPU, we buffer input events until we receive minimum number of input events or maximum event receive timeout is reached, whichever happens first. Usually when there are high frequent input events, minimum number of input events is reached before maximum event receive timeout is reached. Once the required number of events are received, they are pass to GPU Event Processing Library to further processing. If we receive number of events that exceed the maximum number of events, they are partitioned to match the maximum number of events and fed into GPU Event processing library separately.

3. **Serializing input events**
   The input events received via InputHandler are Java Objects that represent event master data and event attributes. Our GPU Event processing library cannot

process events in Java Object format, because it is implemented in C++ and CUDA C/C++ languages. So the input events needs to be converted into a format that is both understandable by the GPU Event processing library and should be able to easily fed into GPU Event processing library and transfer to GPU device memory. We have come up with a event data serialization mechanism which can achieve both above requirements and it is described in Section 3.3.1.

4. **Invoke GPU processing using GPU Event processing library**
   Once all the input events are fed into GPU Event processing library, we invoke GPU event processing by calling relevant GPU Event processing library API function.

5. **De-serialization of output events**
   Once GPU event processing is done, GPU Event processing library will copy back resulting events into host memory and return control to `GpuStreamReceiver`. Received events are then forward to the last event processing construct called `GpuQuerySelector`. `GpuQuerySelector` has the same functionality as `QuerySelector`. `GpuQuerySelector` select only required attributes from the processed input events and generate output events. In `GpuQuerySelector` we have to de-serialize events received from GPU processing stage before they are further processed. Finally, de-serialized and attribute selected events that are in Java Object format are queued to output event queue.

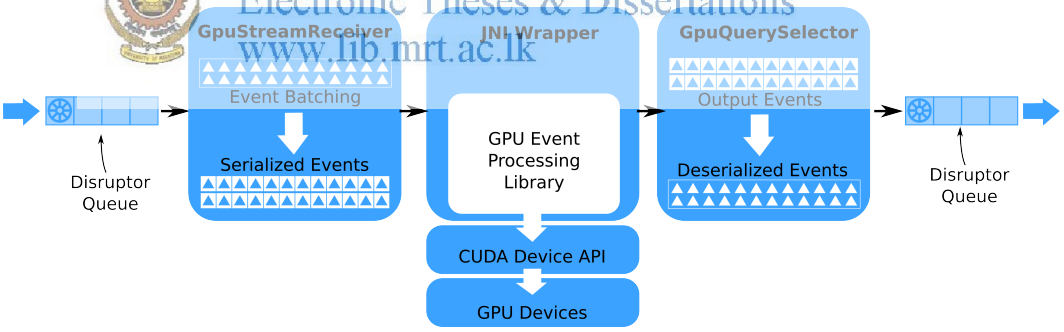


Figure 3.7: Internals of GpuStreamRuntime.
`GpuStreamRuntime` internally do input event batching and serialization, invoke GPU event processors and finally de-serialization of resulting events.

3.3.1 Event Serialization

When designing Event data communication library, the biggest challenge was to choose a communication medium between Siddhi runtime and CUDA runtime. Since there will be very high frequent data transfers happen back and forth between these two runtimes the communication medium should support this high frequent data transfers without showing significant performance overhead.

The main requirements of event serialization was to:

- Convert Java representation of event data into a format that is understood by Java, C++ and CUDA-C.
- Conversion should not add significant overhead to event processing at host side (CPU).
- Serialized data should be easily processed using CUDA C/C++ in GPU runtimes.
- Serialized data should be easily copied to GPU device memory with or without minor modifications.
- Same mechanism can be able to use for de-serialization phase.

Some of these requirements contradict with each other, so we have to manage the trade-offs while finding a solution. Our first approach was to create and allocate memory for all the GPU related data structures inside the library and call a JNI wrapped methods from Java side to fill those data structures with received event data. This method has significant performance overhead when there are lot for input events in an event batch. For example if an event has 5 attributes, for each event we need to execute 5 JNI method calls to fill those 5 event attributes. If there are 2000 events in an event batched we need to execute 10000 JNI method calls. In our experiments, JNI method call took average 25 nanoseconds to execute. Our performance results for this mechanism showed that more than 90% of the GPU query processing time inside the `GpuStreamRuntime` was spent on event serialization phase. That time measurement was even without the event de-serialization phase implemented. This approach was not a viable option so other alternatives were explored.

3.3.1.1 Java NIO ByteBuffer

We found the best way to serialize and transfer event data form Java side to C++/CUDA-C is to convert them into their binary representation and store them in a pre-allocated contiguous memory buffer. This way, even if there is a little bit of overhead in Java side to convert data into binary form, it will be very easy to interpret these data in C++ and CUDA-C side. Because C++ and CUDA-C supports direct memory casting to data types with zero overhead. Even we can use our custom data structures in this way to represent events as sequence of bytes.

The optimal data structure to represent contiguous, pre-allocated memory in Java side is Java New I/O (NIO) `ByteBuffer`. This was part of Java NIO API which was developed to allow Java to interact with features that are I/O intensive in J2SE 1.4. NIO `ByteBuffer` and NIO API provides access to low-level memory buffers that are also visible to non-Java code. Moreover, it is also possible to access low-level memory

that was allocated using non-Java code through NIO `ByteBuffer`. NIO API handles all Byte Ordering issues and provide convenient API to Java code to write and read arbitrary bytes to arbitrary locations in the memory buffer.

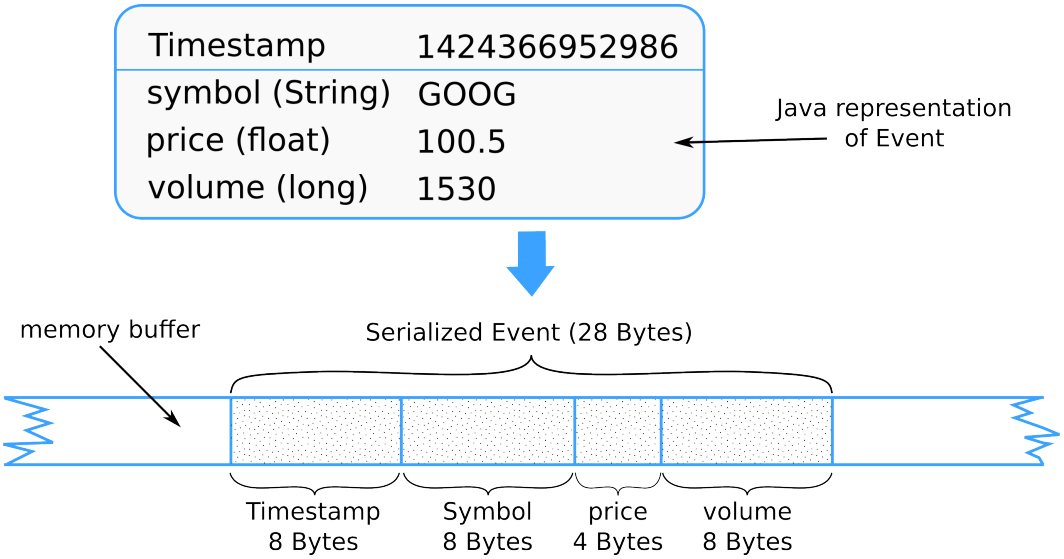


Figure 3.8: Event serialization in to memory buffer.
In event serialization, each attribute of Java event representation is written in to memory buffer in their binary data format.

In our proposed solution we have allocated necessary memory buffers inside the GPU event processing library and accessed these memory buffers through NIO `ByteBuffer` from `GpuStreamReceiver` where the actual serialization part is implemented. When serializing an event, first, metadata of the event is written into memory buffer followed by its attributes in defined order (See Figure 3.8). All the events will take a fixed size on the memory buffer and this size is pre-known.

3.3.1.2 JNI Wrapper for Event Processing Library

GPU event processing library is implemented using C++ and CUDA-C languages. So in order to access API functions implemented in this library, Java code needs to use Java Native Interface (JNI). We have created a JNI wrapper for our library API functions that enables us to easily integrate Siddhi query runtime with this library.

Since the GPU event processing library can be enhanced by implementing new event processing constructs, it will be necessary to update and maintain JNI wrapper with every changes to the library API. Maintaining JNI classes is a tedious and time consuming task. To alleviate this burden we have used a Java library [73] that helps to automatically generate JNI wrapper classes for given C++ API. This way we could easily integrate our GPU library into Siddhi code base. Siddhi development use Maven as a build tool. Both our query runtime implementation and the GPU library are

also integrated to the maven build system. So it is easy to any user to get our implementation and use it in their projects. Currently it needs CUDA runtime SDK to be installed in order to compile the GPU library. But if there is no CUDA SDK found, the build will continue as normal without building GPU specific parts.

3.3.2 Increasing Performance in GpuStreamRuntime

While implementing these features, we have done performance test for each of the implemented parts. Our initial performance tests showed that there is a significant overhead in the serialization and de-serialization code when we are serializing/de-serializing over 2000 events as a batch. In the serialization, first all the events are accumulated in a buffer and once the batch limit was hit, serialization of the accumulated events in the buffer started. For example if there are 4096 events in the batch and each event has 10 attributes including metadata, then there will be 40960 iterations and same amount of NIO API calls to write all attributes into memory buffer. This has created a sudden burst of CPU usage and has severely impact the performance of the other parallel queries.

To address this issue, first, we have changed the design to serialize events as and when they are arrived. Even this approach has the same number of iterations as the previous method, it does not create a sudden burst of CPU usage. Next we have created a thread pool to make serialization parallel. Since we know the size of an event beforehand and it is possible to write event data to any position in the memory buffer and multiple threads can access same underlying memory buffer wrapped through different ByteBuffers, we can distribute the task of serialization into multiple threads. Each thread is assigned part of the events in event batch and given a starting position in the memory buffer to write their assigned events. While this approach had improved the serialization by roughly a factor of number of threads, it has negatively effected the performance when there are several parallel streams. Because each parallel stream creates their own thread pool and having lot of running threads in the JVM has reduced the expected performance gain. So we can only use this approach when there are enough CPU resources in the system or few parallel streams in Siddhi runtime.

Particular problem with the serialization and de-serialization of Siddhi events is that the schema of the event is not directly represented in the Java Event representation. Siddhi uses a class with array of Java Objects to represent an event. The Object array can have any value. So at the serialization time we have to get the actual type information of each event attribute from either Event metadata definition or by using Java reflection. Bytecode level serialization are the fastest serialization methods, but cannot be used in this problem because of the above reason.

Chapter 4

GPU Event Processing Library

The major part of this research work was spend on designing and implementing event processing constructs for GPU environment. Implementing low-level GPU device handling inside Siddhi makes it hard to debug because of Java to CUDA C/C++ communication. Our approach is to decouple all the low-level GPU device handling from the Siddhi and implement them in a separate library. We have implemented a general purpose GPU event processing library to the very purpose. This chapter describes the design and implementation details of this library and how it is optimized to yield high performance event processing.

In Section 4.1, we describe existing researches on accessing CUDA enabled GPU devices with in Java language and our experience on integrating those research approaches with our solution. Section 4.2 elaborates on the implementation of our general purpose GPU event processing library while Sections 4.3, 4.4 and 4.5 explain on the GPU event processing algorithms for Filter Processor, Window Processor and Stream Join Processor, respectively.

4.1 CUDA Access for Java Code

The main goal of this research is to find the possibility of using GPU hardware devices provided computing power to increase the event processing performance in Siddhi CEP engine. With the implementation of new Siddhi query runtime, which is described in Chapter 3, baseline framework for Siddhi to communicate with GPU devices is established. But within Siddhi it is hard to directly access GPU devices. As we have explained in the literature review chapter, there are several other research work done in the past to address this very problem. The two most notable research approaches are evaluated that enable direct access to GPU devices within Java VM; namely JCuda [50] and Rootbeer [46].

JCuda is the most widely used Java Library to interact with the CUDA runtime and driver APIs. A sample application is implemented for event processing using JCuda which transfer synthetically generated events from Java to GPU. There are major limitations with the implementation of JCuda that prohibits its use in our research work. JCuda has two methods of interacting with CUDA runtime; JCuda Runtime API and JCuda Driver API, which use *CUDA Runtime API* and *CUDA Driver API* respectively. Using JCuda Runtime API is easy, since it is not required to write any CUDA code, JCuda library auto generates CUDA code according to application requirement. But JCuda Runtime API lacking some basic low-level control over CUDA device that our approach require. On the other hand, JCuda Driver API enables user to write custom CUDA kernel code and call them inside Java using JCuda library calls. But user has to manually do all the low-level memory management of GPU memory space inside Java code. This is not straight forward as it is with pure CUDA code. JCuda could not provide the necessary control over our data structures in CUDA side and the debugging of CUDA kernels with Java code was not straightforward as it was with pure CUDA C/C++. Because of these reasons JCuda was not used in our solution and hence performance measurements were not done with JCuda.

Rootbeer [46] provides novel approach to writing CUDA applications in Java. It provides automatic CUDA code generation, data serialization/de-serialization and automatic kernel launch. The most notable contribution by the Rootbeer is its serialization mechanism. Rootbeer is able to serialize all the necessary data for the kernel execution and transfer them to GPU memory very efficiently. It dose this by static analysis of user written Java Kernel Code and generating bytecode for serialization of required data. Because of the bytecode level serialization mechanism, Rootbeer can achieve impressive performance.

Rootbeer was used to CUDA code generation for a sample application that does event data serialization of Siddhi events. The performance results did not show any improvements of event serialization for CUDA code generated by Rootbeer. This is because for the particular use-case Rootbeer could not generate optimized CUDA code. Rootbeer generates bytecode for serialization if it can find data type at the compile time. But in our case, Siddhi dose not represent event attribute types in event representation at compile time. It merely use a class with Java Object array to represent event attributes. Object can have value of any type, and the actual type information is depend on the event schema defined by the user at runtime. So CUDA data serialization code generation provided by Rootbeer is not usable for Siddhi event serialization.

4.2 Event Processing Library for GPU

Since none of the previous researches could provide necessary features and control over GPU devices, a custom event processing library called *libGpuEventRuntime* was developed for GPU devices that support CUDA Runtime. The main functionality of this library is to event processing on GPU rather than general purpose Java to GPU communication. So we could limit the scope of interaction points and improve the performance for event processing specific cases.

When designing the event processing library, it was considered the extensibility and compatibility with other types of GPU computing runtimes like OpenCL. So the communication with the library from Java code is done via a simple API, wrapped through JNI, and Java ByteBuffer wrapped memory buffers was used to transfer event data between Java code and the library.

Using a separate custom developed library and writing custom CUDA kernels provides better control over the low-level hardware and convenience of optimizing for the task at hand. As we explained in Chapter 3, the JNI wrapper is auto generated by using a special library, so the overhead of coding for the library is very minimum for a user of the library. Basic event processing API is stable and will not change, while the API for specific event processing constructs like Filter Processor and Event Window Processor can be changed. This is because users are able to modify, enhance and add new event processing construct to the library, if they are familiar with CUDA. Even it is possible to enhance the performance of existing event processing constructs in the library without changing the API. So users of the library can gain increased performance with zero changes in Siddhi code.

Segregating lower level implementation of GPU event processing in to a separate library and accessing it through a unified API gave us the ability to test and debug Siddhi and event processing library separately. The library can be tested and debug separately using its API. The testing can be done using both Java and non-Java code.

The three main tasks done by the event processing library include:

1. Allocating memory buffers for event data serialization.
2. Create and organize event processing constructs.
3. Call event processing functions of Event processors, which in turn invoke CUDA kernels.

Form Siddhi code the library API is accessed by the `GpuStreamRuntime`, where it calls necessary functions to initialize the library and create GPU event processors. `GpuStreamRuntime` initialize the library to create a `GpuQueryRuntime` (see Figure 4.1). There should be one `GpuQueryRuntime` for each `GpuStreamRuntimes`. Which means

there are separate instance of library runtime for each event queries defined by the user. `GpuQueryRuntime` provides necessary runtime environment for creating and maintaining GPU processors. It holds an array of `GpuStreamProcessors` which handles each stream related to the particular query. Most of the time there is one instance of `GpuStreamProcessors` in the array, but for Join streams, Sequence and Pattern streams there can be more than one `GpuStreamProcessors`. As shown in Figure 3.3, in Siddhi, each stream has chain of event processors to process events in defined order. Following the same design we have created a chain of `GpuEventProcessor` chain in each `GpuStreamProcessors` that process events. For each event processor type in Siddhi there is a matching `GpuStreamProcessors` implementation in the library. Currently we have implemented three `GpuStreamProcessors` for following operators.

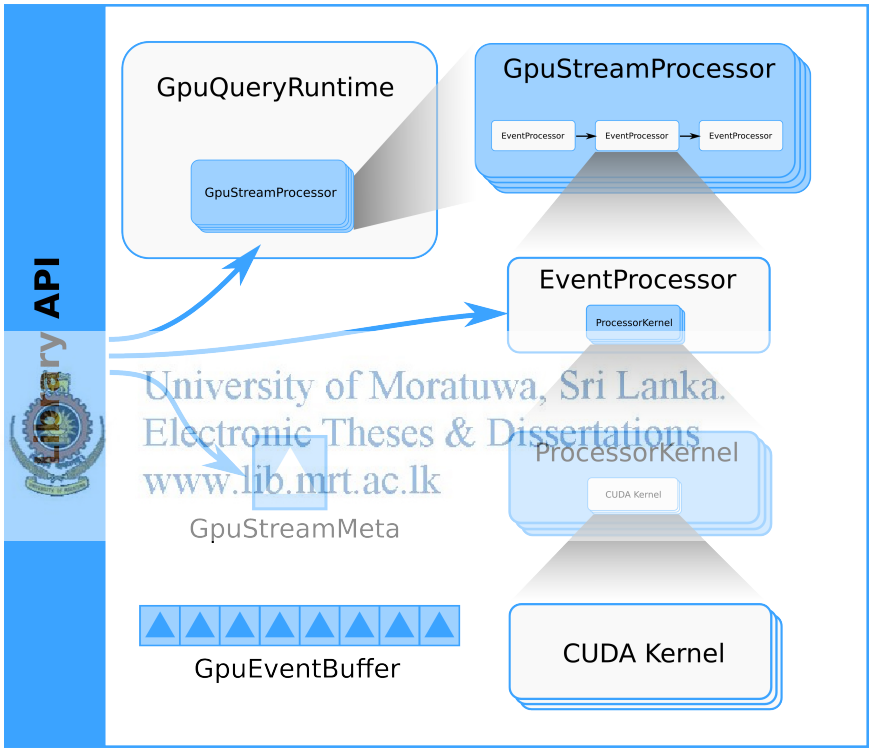


Figure 4.1: High-level design of GPU Event Processing Library.
GPU Event Processing Library provides access to GPU Event processing constructs and memory buffers through well-defined API.

- `GpuFilterProcessor` - Provides event filter processing.
- `GpuLengthSlidingWindowProcessor` - Provides sliding window operator with fixed length event window.
- `GpuJoinProcessor` - Provides stream join operator for two event streams. Currently supports only event window with fixed length.

Each event processor implementation can have one or more `GpuProcessorKernels`. `GpuProcessorKernel` wraps the actual CUDA kernels implemented for this GPU event

processor. Usually there can be more than one `GpuProcessorKernel`s per GPU event processor and more than one CUDA kernel per `GpuProcessorKernel`. The library API only provides access to `GpuQueryRuntime` class, `GpuEventProcessor` implementations and `GpuStreamMeta` class. All the other classes and data structures are hidden to library users. Following API functions are provided by the library.

- ```
  void GpuQueryRuntime::Initialize(
            string queryName,
            int cudaDeviceId,
            int inputEventBufferSize)
  ```
  Initialize new GpuQueryRuntime environment by providing query name, GPU device ID and input buffer size in bytes. GPU device ID is the numerical index of attached GPU devices. Input buffer size is used for allocating host side and GPU device side memory for input event memory buffer.

- ```
  void GpuQueryRuntime::AddStream(
            string streamId,
            GpuStreamMeta * streamMetaInfo)
  ```
  Create new stream representation in GPU side using the given meta info. `GpuStreamMeta` contains the definition for each attribute of the events that can arrive through this stream.


- ```
  void GpuQueryRuntime::AddProcessor(
  string streamId,
  GpuProcessor * processor)
  ```
  Add new GPU event processor instance in to the stream identified by `streamMetaInfo`. `processor` can be any specific implementation of the `GpuProcessor`

- `char * GpuQueryRuntime::GetInputEventBuffer(string streamId)`
  `int GpuQueryRuntime::GetInputEventBufferSizeInBytes(string streamId)`
  Get memory buffer for input events and its size in bytes. This memory buffer is accessed by `GpuStreamRuntime` from Siddhi side using a NIO `ByteBuffer`.

- `int GpuQueryRuntime::Process(int numberOfEvents)`
  This function is called when the `GpuStreamRuntime` is done serializing the input events and it need to start processing that events. Input parameter set how many events in the buffer should be processed and the output parameter returns the number of resulting events.

API of each event processors will be discussed in their relevant sections.
  ```

4.2.1 GPU Event Processors

Each implementation of GPU event processors should implement interface functions defined by the `GpuProcessor`. Following interface functions are defined in `GpuProcessor`.

- ```
  void GpuProcessor::Configure(
              int streamIndex,
              GpuProcessor * prevProcessor,
              GpuProcessorContext * context)
  ```

- ```
  void GpuProcessor::Initialize(
              int streamIndex,
              GpuStreamMeta * streamMetaInfo,
              int inputEventBufferSize)
  ```

- ```
  int GpuProcessor::Process(
              int streamIndex,
              int numEvents)
  ```


The `Process()` method is the entry point to the GPU processor. It gets called by either `GpuQueryRuntime::Process()` method or by the `Process()` method of previous GPU Processor in processor chain. And the GPU processor should also implement copying of event data back to host memory if it is the last processor of the processor chain. All the memory buffers are wrapped by a data structure called `GpuEventBuffer` which provides convenient event data wrapper to memory buffers. All the memory copying of host-to-device and device-to-host are implemented in the `GpuEventBuffer`.

4.2.2 Event Processor Basic Concepts

Before designing each GPU event processor we have analyzed existing event processors in siddhi, how events are flown through these processors and how they are maintaining the state of the processor. The following facts were observed in general for each event processor.

- **Each event processor have a state**
  By state what meant here is the values in the memory of each event processor instance. For example, event window processor has an array of events stored in the memory which called "Event Window". For a given time there can be some set of events in this event array. For a given time this is called the state of that
  ```

window event processor for that time instance. Values stored in memory for that particular event processor for a particular time defines its state for that time instance.

- **Each event processor gets one or more events as input**
- **After processing input events there can be zero, one or more output events**
  Most of the event processors output more than one events, but there are event processors like event filter processor which can filter out all incoming events. In some event processors there can be more output events than the number of input events (for example stream join processor).
- **After processing a batch of input events, the state of the processor can be changed**
  The stored data inside the event processor instance can have new set of values after processing all events. For example, the window event processor can have new set of events in its window buffer after it processed batch of events.

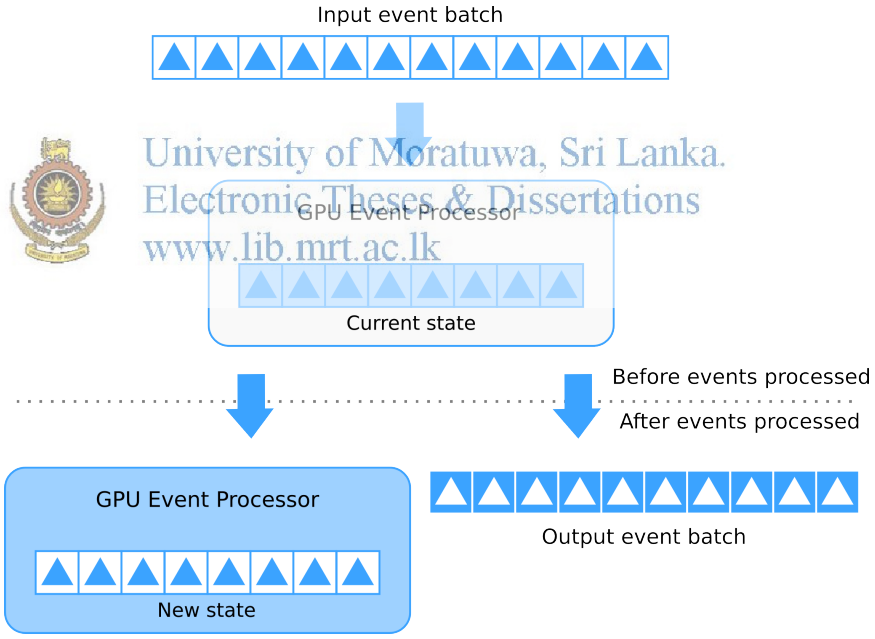


Figure 4.2: Basic concept of GPU event processors.
When designing GPU event processor algorithms, the guiding principle was to maintain same output events, pre- and post- states of event processor as the sequential algorithm.

These factors are important because when designing parallel event processing algorithm for GPUs primary objective was maintaining same pre and post state in event processor and generate same output events for particular batch of incoming events (see Figure 4.2). We considered above observations as a guideline for implementing GPU processors. They helped us to to design and implement novel algorithms for

parallel event processing in GPUs without just translating sequential algorithms to their parallel version.

4.3 GPU Filter Event Processor

Filter processor provides selective output of events based on the user defined selection criteria. Events that matched to the given criteria output to the output event buffer and forward to the next event processor. Other events are discarded. See sample filter query defined in Listing 4.1.

```
1 from StockStream[price > 100.5 AND symbol == "GOOG" OR symbol
    == "FB"]
2 select symbol, price, volume
3 insert into TopStockStream;
```

Listing 4.1: Filter query defined in SiddhiQL.

Siddhi implements Filter processor using object oriented expression tree evaluation. The selection criteria is a collection of *Expressions* that ultimately evaluate to a boolean value. Expressions can be in many types.

- Condition expressions - AND, OR, NOT, COMPARE, PREFIX, SUFFIX, CONTAINS, etc.
- Mathematical expression - ADD, SUB, MULT, DIV, MOD, etc.
- Variable expression - Value of event attribute symbol
- Constant value expression - 100.5, “GOOG”, etc.

Siddhi defines separate Java classes, called Executors, for each of these expressions. In fact, there are separate classes for each expression for each type combination like `AndExpressionExecutorFloat`, `MultiplyExpressionExecutorInt`, etc. When a user defined filter query is compiled and comes to Siddhi runtime, it creates an expression tree of these Executor class instances as shown in Figure 2.5. When an event needs to be evaluated using this expression, the root Executors of the expression tree will be called with event as a parameter. The expression tree is evaluated in in-order tree traversal and returns a boolean value indicating the matching or none matching of the particular event. The advantage of using an expression tree is it stops evaluating the given event at first instance it finds the whole tree will evaluate to true or false, without evaluating the whole tree.

In our approach to implement Filter operator in GPUs, we first tried implementing the CCM algorithm propose by Cugola et. al. [28] for their content based event dispatching

mechanism. This solution could provide very high throughput of event processing (over 8 million events per second) for basic operators like AND, EQUAL, GRATER_THAN, etc. But the solution could not evaluate most of other expressions supported by Siddhi. For example, it could not process expressions that are connected with OR operator. This approach was not continued.

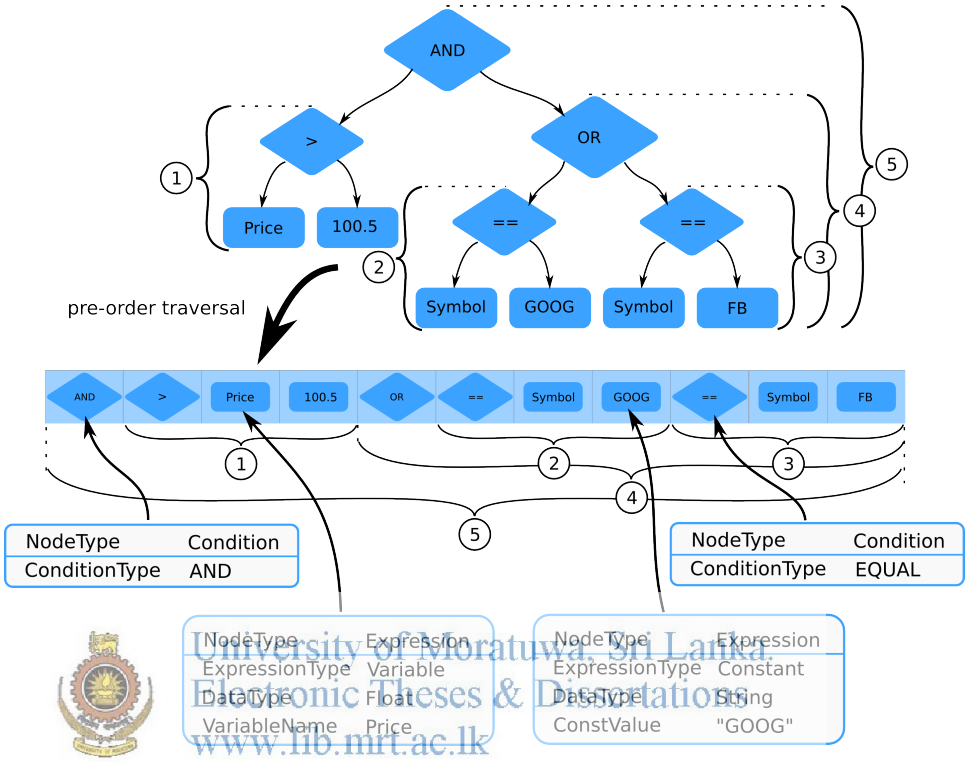


Figure 4.3: Filter operator expression tree conversion to executor array for query defined in Listing 4.1.
Expression tree is converted to an array using post-order traversal and constructing executor node to represent each node in the tree.

The second approach was to implementing the same expression tree as a tree data structures in GPU memory and evaluate it using a given event. This way, instead of evaluating events sequentially, we could evaluate every event in the batch in parallel at once. But the initial performance results showed otherwise. Evaluating a tree data structure is not optimized in GPU environment. Hence, the expression tree was converted to an array based data structure following pre-order tree traversal. As shown in Figure 4.3, each element in the array is of ExpressionNode or ConditionNode type and contains the expression/condition type, data type of the evaluated result, constant value if the node is a Constant value expression and event attribute name if the node is a Variable expression.

```
__device__ float AddExpressionFloat(FilterEvalParameters & _rParameters)
{
        return (ExecuteLeftFloatExpression(_rParameters) +
                    ExecuteRightFloatExpression(_rParameters));
}

__device__ bool GreaterThanCompareIntInt(FilterEvalParameters & _rParameters)
{
        return (ExecuteLeftIntExpression(_rParameters) >
                    ExecuteRightIntExpression(_rParameters));
}

__device__ bool AndCondition(FilterEvalParameters & _rParameters)
{
        return (EvaluateLeft(_rParameters) & EvaluateRight(_rParameters));
}
```

Listing 4.2: CUDA functions for filter expression evaluation.

A set of CUDA device functions was implemented to represent each expression/condition types and each of these functions evaluates its child nodes calling relevant expression functions. Few sample CUDA expression evaluate functions are shown in Listing 4.2. Pointers to all these functions are stored in a function pointer array indexed by the type of the function.

In the CUDA kernel we followed the algorithms, shown in Algorithm 4.1 and Algorithm 4.2, to evaluate a given event with the expression array. GPU filter processor will invoke numbers of GPU threads equal to the number of input events in the batch. So each GPU thread is processing one input event form the batch. Each GPU thread executes the `FilterEntryFunction` shown as Algorithm 4.1. A GPU thread gets its assigned input event from the `inputEventBuffer` by calculating the offset of the input event using `sizeOfEvent` and `threadIdx`. This is shown in line 2. `expressionNodeArray` is a globally accessible data structure which contains the expression nodes as discussed in Figure 4.3. In line 4, this expression node array is accessed to get the first expression evaluation function pointer. The first expression evaluation function pointer is always a Condition Expression. Expression evaluation function pointer is recursively executed to get the overall evaluation result. If overall evaluation result is `True`, then the input event is copied to the output event buffer (see line 7). If the overall result is `False`, the input event is discarded.

In each of the filter evaluation expression functions, first, the `expressionNodeArrayIndex` is incremented to access the next node in `expressionNodeArray`. Then the left and right operands of the current expression, which represented as nodes in the array, is executed recursively. If the results of the operands are boolean values, they are evaluated according to the evaluator function and the resulting boolean value is returned. Otherwise the variable value or the constant value is returned. This is shown in Algorithm 4.2.

Algorithm 4.1 FilterEvaluation - Entry Function.

```
1: function FilterEntryFunction
Input: inputEventBuffer
Input: outputEventBuffer
2:     inputEvent ← inputEventBuffer[sizeOfEvent * threadIdx]
3:     expressionNodeArrayIndex ← 0
4:     EvalFunction ← expressionNodeArray[expressionNodeArrayIndex].Func
5:     eventMatched ← EvalFunction(inputEvent, expressionNodeArrayIndex)
6:     if eventMatched == True then
7:         outputEventBuffer[sizeOfEvent * threadIdx] ← inputEvent
8:     end if
9: end function
```

There are two versions of GPU Filter evaluation function. If the Filter event processor is the only event processor defined in the query, then we use one version of the function. In this case we optimized the output events by only output index of the events in the input event batch that matched the given criteria. This reduced the number of operations in the GPU kernel and also reduced size of the output buffer. For all the other cases, the other version of the filter function is used, where it output full event data to the output event buffer.

In the second version of the GPU filter function, where it output full event data to the output event buffer of matched events, to identify the location to write output event by each GPU thread, a thread synchronization mechanism was needed. Since synchronizing GPU threads in multiple thread block may have negative impact on performance, we have used a synchronization-less algorithm. There are three phases of this algorithm. An integer array is created with the size of number of GPU threads. In first phase each thread process its assigned input event and check if it matches the given filter criteria. If it matches, integer array position matches to its thread index is updated with the value one. If assigned event did not matched the filter criteria, the array is updated with value zero. In the second phase of the algorithm, we performed parallel prefix-sum on this integer array using a GPU prefix-sum algorithm. Prefix-sum provides the number threads that has output data to be written to output event buffer. By using the integer value in the index of each GPU threads' thread index, it can get the output event buffer writing index. The resulting integer array is used by the third phase of the algorithm to update output event buffer without overwriting other threads event output data.

The most notable advantage of this approach over the other previous approaches is this can support almost all filter criteria evaluation functions currently supported by Siddhi. The only filter criteria evaluation function that cannot be supported by this approach is user defined functions. But if the user is familiar with CUDA, she can write a custom CUDA function which does the same task as the user defined function.

Algorithm 4.2 FilterEvaluation - Expression Functions.

1: **function** BINARY CONDITION FUNCTION
Input: *inputEvent*
Input: *expressionNodeArrayIndex*
2: $expressionNodeArrayIndex++$
3: $EvalFunction \leftarrow expressionNodeArray[expressionNodeArrayIndex].Func$
4: $leftResult \leftarrow EvalFunction(inputEvent, expressionNodeArrayIndex)$
5: $EvalFunction \leftarrow expressionNodeArray[expressionNodeArrayIndex].Func$
6: $rightResult \leftarrow EvalFunction(inputEvent, expressionNodeArrayIndex)$
7: **return** $ApplyBinaryOp(leftResult, rightResult)$
8: **end function**

9: **function** UNARY CONDITION FUNCTION
Input: *inputEvent*
Input: *expressionNodeArrayIndex*
10: $expressionNodeArrayIndex++$
11: $EvalFunction \leftarrow expressionNodeArray[expressionNodeArrayIndex].Func$
12: $result \leftarrow EvalFunction(inputEvent, expressionNodeArrayIndex)$
13: **return** $ApplyUnaryOp(result)$
14: **end function**

15: **function** BINARY EXPRESSION FUNCTION
Input: *inputEvent*
Input: *expressionNodeArrayIndex*
16: $expressionNodeArrayIndex++$
17: $EvalFunction \leftarrow expressionNodeArray[expressionNodeArrayIndex].Func$
18: $leftResult \leftarrow EvalFunction(inputEvent, expressionNodeArrayIndex)$
19: $EvalFunction \leftarrow expressionNodeArray[expressionNodeArrayIndex].Func$
20: $rightResult \leftarrow EvalFunction(inputEvent, expressionNodeArrayIndex)$
21: **return** $ApplyBinaryExpressionOp(leftResult, rightResult)$
22: **end function**

23: **function** VARIABLE EXPRESSION FUNCTION
Input: *inputEvent*
Input: *expressionNodeArrayIndex*
24: $expressionNodeArrayIndex++$
25: $attrIdx \leftarrow expressionNodeArray[expressionNodeArrayIndex].AttrIdx$
26: $attributeValue \leftarrow inputEvent.Attributes[attrIdx]$
27: **return** $attributeValue$
28: **end function**

29: **function** CONSTANT VALUE EXPRESSION FUNCTION
Input: *inputEvent*
Input: *expressionNodeArrayIndex*
30: $expressionNodeArrayIndex++$
31: $constVal \leftarrow expressionNodeArray[expressionNodeArrayIndex].Const$
32: **return** $constVal$
33: **end function**

This user written custom CUDA functions can be linked and called within GPU filter processor.

4.4 GPU Sliding Window Event Processor

Window event processor is used to store input events in an event buffer for temporal processing. There are several types of Window event processors implemented in Siddhi, such as Length Window, Time Window, Length Batch Window, Time Batch Window and Unique Window. In this research we have implemented only Length Sliding Window Processor. Length Sliding Window Processor store input events in a fixed length event buffer and when a new event comes to the Window processor, if the buffer is full, least recent event in the buffer is removed and output to the output event buffer as an *Expired Event*. This way it maintain a fixed length sliding window of last received events. The incoming event is always output to the output event buffer (see Figure 4.4). Sample length sliding window query is shown in Listing 4.3. Window event processor is mainly used for event aggregation and stream joining.

```
1 from StockStream#window.length(100000)
2 select symbol, avg(price) as avgPrice, sum(volume) as
     totalVolume
3 insert into WindowStockStream;
```

Listing 4.3: Length sliding window query defined in SiddhiQL.

The reason for implementing only length sliding window processor is that it has deterministic memory usage in GPU memory. GPU memory needs to be pre-allocated in order to achieve full performance, because memory allocation in GPU is an expensive operation. Length sliding window has fixed size memory requirement whereas time sliding window processor has non-deterministic memory requirement. If the input event rate is so high, it is hard to define a size for GPU memory buffer beforehand. So as a proof of concept, only the length sliding window processor is currently implemented. Implementation of length batch window processor would be the same as length sliding window processor.

As stated in the Section 4.2.2, every event processor has state before processing set of input events and this state is changed once the processing is complete. This behavior is obvious in the window processor and guided us to design our GPU event processing algorithm. There are two phases in the algorithm; (i) generate output events, (ii) set post-process window state from input events. These two steps are handled by two CUDA kernels and explained in following sections.

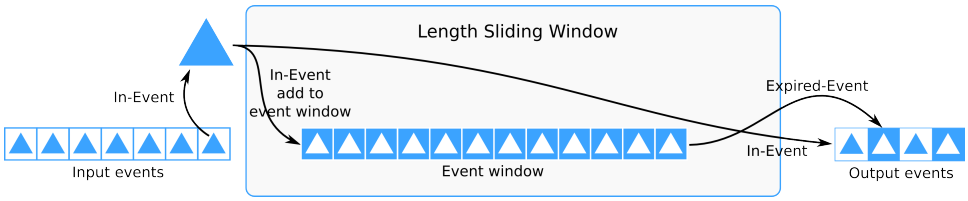


Figure 4.4: Length Sliding Window Processor.
Length sliding window maintain a fixed length of previous input events in a event buffer. If the buffer is full, least recent event is removed to make space for new events.

4.4.1 Generating Window Output Events

In the first phase of the algorithm, output events for a set of input events are calculated. In sequential algorithm this is done by processing one input event at a time and checking it with the current state of the even window. This sequential method needs update of event window with each input event in order to produce correct output events. In a parallel algorithm for calculating output events, updating event window and using updated event window from other thread is not practical as this approach requires tight synchronization between threads. In GPU threads, synchronization between multiple threads can be expensive, if they are in two different thread blocks.

In the sequential algorithm, it is observed that there is a connection between output event for each input events and the final output events for the same input event batch. As shown in Figure 4.5, the output events set is a sequenced collection of {*expired,input*} event pairs calculated for each input events in input order. It is also observed that {*expired,input*} event pair can be calculated in parallel for each input event without depend on other thread's values. The only required data for the parallel calculation is initial state of the event window and the batch of input events.

A parallel algorithm were developed, as shown in Algorithm 4.3, to calculate {*expired,input*} event pairs for each input event and update the output event buffer using calculated event pair. GPU event window processor invokes a number of GPU threads that match number of input events in the incoming event batch. Each GPU thread is assigned an event in the input event batch and it calculates the {*expired,input*} event pair for its assigned input event.

At the initialization, GPU event window processor allocates GPU memory for event window and output events buffer. Output events buffer has the twice the size of input event buffer, since there can be two output events ({*expired,input*} event pair) for each input event, as shown in line 1 and 2 of Algorithm 4.3. Each GPU thread is allocated two event positions in the output buffer to write its output events. When writing events to the output event buffer, each GPU thread uses its thread index to access its allocated buffer position in output buffer. Always the first position is used to write the expired

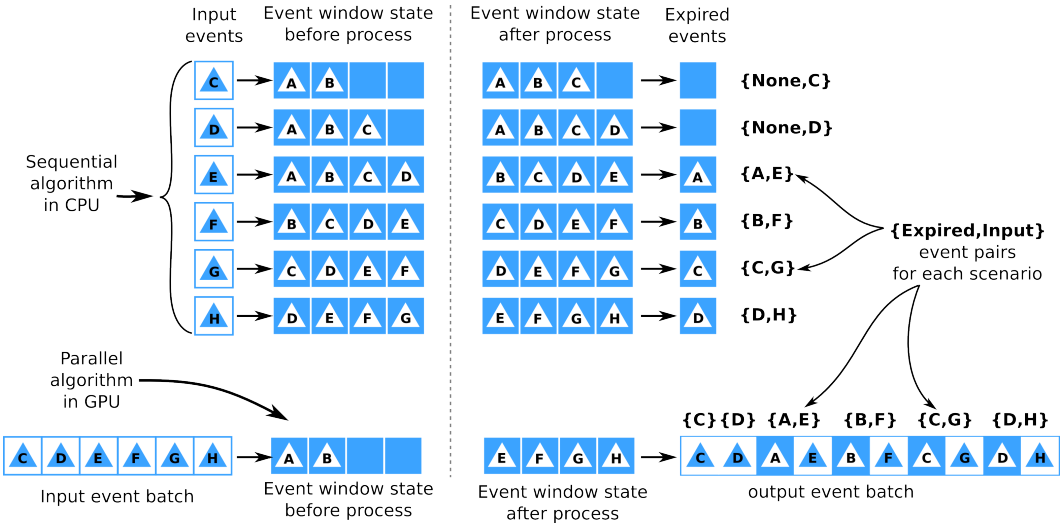


Figure 4.5: Length Sliding Window Processor - output event calculation.
In sequential algorithm output event for each input event is calculated one event at a time. In parallel algorithm output events for a input event batch calculated each event in parallel using expired/input event pairs.

event and the second position is used to write the input event. In case if there is no any expired event for a particular input event, then the `Null` value is written to expired event position. This way the synchronization between GPU threads is removed.

To calculate the expired event for a particular input event, first, the remaining event window buffer size is checked. If there are remaining buffer positions in window buffer, input event is written to that position and no expired event is generated (see line 3 of Algorithm 4.3). If event window size is larger than current thread's thread index, then expired event is calculated from event window buffer (see from line 4 to line 11). If event window size is less or equal to current thread's thread index, then expired event is calculated from input event buffer (see from line 13 to line 15).

4.4.2 Set Post-process Window State

Once the output event calculation phase is completed by the GPU processors, event window update processor is scheduled to update the state of the event window using input events. When this process completes the execution, event window should have same state as the sequential version of event window processor.

We invoke number of GUP threads that match to number of input events in the batch. Each GPU thread is responsible for one input event and the Algorithm 4.4 is used in each GPU thread to update event window using its assigned input event.

Inputs to this algorithm are input event buffer, event window buffer, input event count, window length, window remaining count and size of an event in bytes. The window

Algorithm 4.3 EventWindowOutputEventsCalculation

Input: $inputEventBuffer$
Input: $eventWindowBuffer$
Input: $outputEventBuffer$
Input: $inputEventCount$
Input: $windowLength$
Input: $windowRemainCount$
Input: $sizeOfEvent$

1: $inputEvent \leftarrow inputEventBuffer[sizeOfEvent * threadIdx]$
2: $outputEventPair \leftarrow outputEventBuffer[sizeOfEvent * threadIdx * 2]$

3: **if** $threadIdx \geq windowRemainCount$ **then**
4: **if** $threadIdx < windowLength$ **then**
5: $bufOffset \leftarrow sizeOfEvent * (threadIdx - windowRemainCount)$
6: $expiredEvent \leftarrow eventWindowBuffer[bufOffset]$
7: **if** $expiredEvent \neq Null$ **then**
8: $outputEventPair[0] \leftarrow expiredEvent$
9: **else**
10: $outputEventPair[0] = Null$
11: **end if**
12: **else**
13: $bufOffset \leftarrow sizeOfEvent * (threadIdx - windowLength)$
14: $expiredEvent \leftarrow inputEventBuffer[bufOffset]$
15: $outputEventPair[0] \leftarrow expiredEvent$
16: **end if**
17: **else**
18: $outputEventPair[0] = Null$
19: **end if**

20: $outputEventPair[1] \leftarrow inputEvent$

Algorithm 4.4 EventWindowStateSet

Input: $inputEventBuffer$
Input: $eventWindowBuffer$
Input: $inputEventCount$
Input: $windowLength$
Input: $windowRemainCount$
Input: $sizeOfEvent$

```
1:  inputEvent ← inputEventBuffer[sizeOfEvent * threadIdx]
2:
3:  if windowLength > inputEventCount then
4:      shiftDistance ← windowLength − inputEventCount

5:      if inputEventCount > windowRemainCount then
6:          exitEventCount ← inputEventCount − windowRemainCount
7:          windowWritePosition ← threadIdx + shiftDistance

8:          EventShift(eventWindowBuffer,
9:                                  windowWritePosition,
10:                                 exitEventCount,
11:                                 windowRemainCount)

12:         bufOffset ← sizeOfEvent * windowWritePosition
13:         windowEvent ← eventWindowBuffer[bufOffset]
14:         windowEvent ← inputEvent
15:     else
16:         shiftDistance ← windowLength − windowRemainCount
17:         windowWritePosition ← threadIdx + shiftDistance
18:         bufOffset ← sizeOfEvent * windowWritePosition
19:         windowEvent ← eventWindowBuffer[bufOffset]
20:         windowEvent ← inputEvent
21:     end if

22: else
23:     shiftDistance ← inputEventCount − windowLength

24:     if threadIdx ≥ shiftDistance then
25:         windowWritePosition ← threadIdx − shiftDistance
26:         bufOffset ← sizeOfEvent * windowWritePosition
27:         windowEvent ← eventWindowBuffer[bufOffset]
28:         windowEvent ← inputEvent
29:     end if

30: end if
```

Algorithm 4.5 EventShift

Input: $eventWindowBuffer$
Input: $windowWritePosition$
Input: $exitEventCount$
Input: $windowRemainCount$

```
endPosition ← windowWritePosition
previousToEnd ← endPosition
while endPosition ≥ 0 do
    if eventexistineventwindowatendPosition then
        previousToEnd ← endPosition
        endPosition ← endPosition − exitEventCount
    else
        break
    end if
end while
if endPosition < 0 then
    endPosition ← previousToEnd
end if

while endPosition < windowWritePosition do
    eventOffset ← endPosition + exitEventCount
    fromEvent ← eventWindowBuffer[eventOffset]
    eventWindowBuffer[endPosition] ← fromEvent
    endPosition ← endPosition + exitEventCount
end while
```

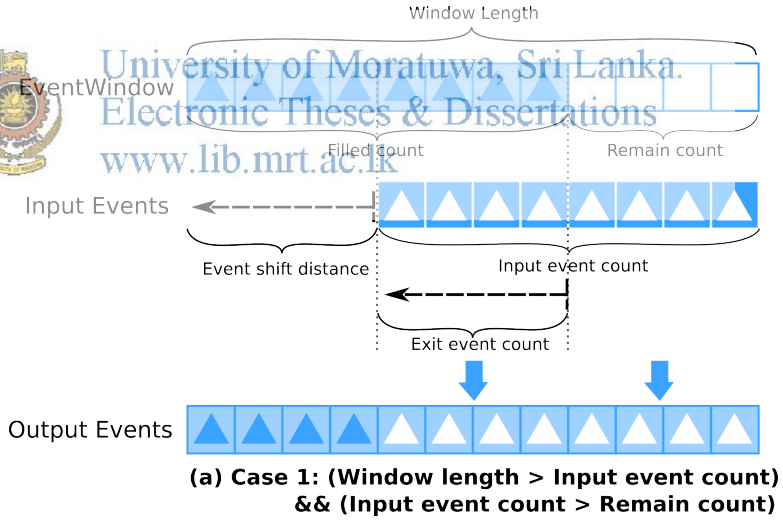
remaining count means the number of free slots in an event window that still to be filled with events. Apart from above inputs each thread gets its thread specific indexes (threadIdx and blockIdx) as inputs.

Using these input values each thread performs three main steps in this algorithm.

1. **Find assigned input event**
   Each GPU thread finds its assigned input event from the input event batch using thread specific indexes. Since we invoke GPU kernel with threads matching to number of input events in batch, each input event is assigned to one GPU thread. This task is explained in line 1 in Algorithm 4.4.

2. **Find writing position in event window**
   Finding event window position can be explained in three scenarios.

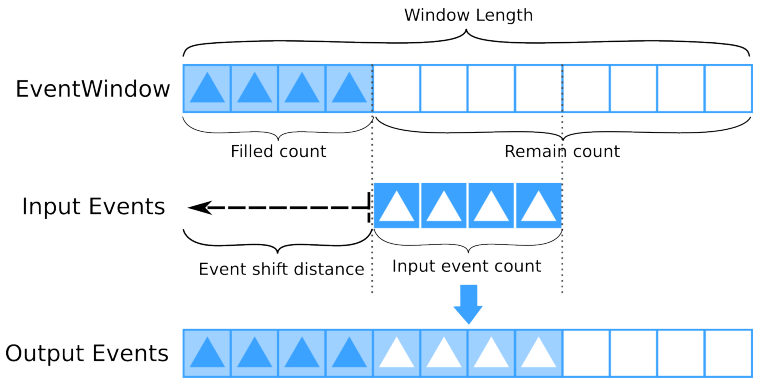
   - **Event window length is larger than input event count and window remaining count is less than input event count**
     As shown in the Figure 4.6, in this scenario all the input events are written to event window and the existing events in event window are shift forward to make space for input events. This is shown in Algorithm 4.4 from line 5 to line 15.



(a) Case 1: (Window length > Input event count) && (Input event count > Remain count)

Figure 4.6: Event window state set algorithm: scenario 1.
Event window length is larger than input event count and window remaining count is less than input event count.

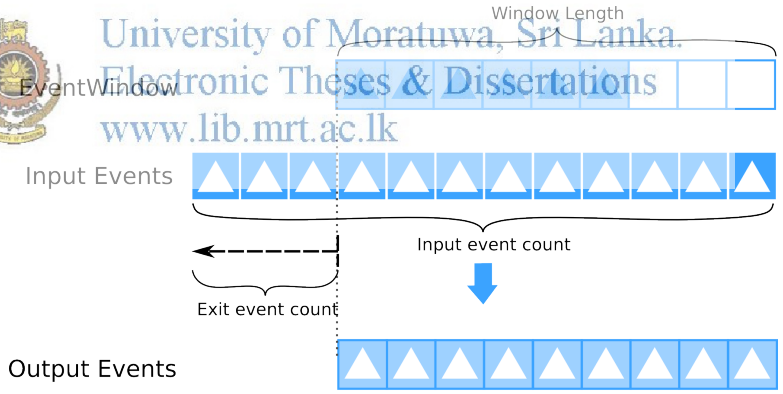
   - **Event window length is larger than input event count and window remaining count is larger or equal than input event count**
     As shown in the Figure 4.7, in this scenario all the input events are written into event window as previous scenario. Since there are enough space left in event window for new incoming events, no events will be removed from event window. This is shown in Algorithm 4.4 from line 15 to line 21.



(b) Case 2: (Window length > Input event count) && (Remain count > Input event count)

Figure 4.7: Event window state set algorithm: scenario 2.
Event window length is larger than input event count and window remaining count is larger or equal than input event count.

- **Event window length is less or equal to input event count**
  As shown in the Figure 4.8, in this scenario, not all input events written into event window. Number of events that matched to event window length from the back of the input event batch is written into event window buffer and the rest is discarded. This is shown in Algorithm 4.4 from line 22 to line 30.



(c) Case 3: Window length <= Input event count

Figure 4.8: Event window state set algorithm: scenario 3.
Event window length is less or equal to input event count.

There may be threads that do not write its assigned event to event window depend on input parameters.

3. **Writing input event data to window position**
   Once a thread has input event position and window writing position and if there is no existing event in the window at the writing position, it copies input event data into that position. We have used direct memory copy for copying event data to event window.

If there is an existing event in the event window in the writing position, then we use a separate algorithm to shift existing events to make space for input events (see Figure 4.9 and Algorithm 4.5). Event shifting is associated with an shifting distance which was calculated along with step two. Starting from event window writing position, we first check if there is an existing event in that position, if so it is moved shift distance number of event positions backward. If there is also an existing event in this new position, we move that event shift distance further back. This logic apply recursively until we reache the begin of event window.

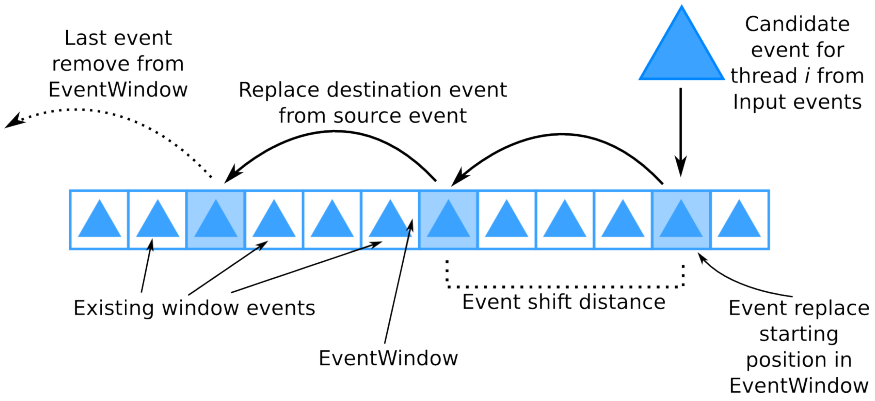


Figure 4.9: Event replace mechanism of EventWindow state update GPU algorithm. Each GPU thread replaces particular Window position with its assign event. If there is already an event, shift it backward by Shift Distance. Last event removed from Window.

Initial version of this algorithm used recursive function to replace events in Window with assigned event, recursively traverse to the last position in event window and swapping events on the way-back. Although it is easy to develop recursive functions for this particular task, performance measurements showed that recursive functions are expensive in GPU environment. This is particularly observable when event window size is very larger than input event batch size.

4.5 GPU Event Stream Join Processor

Event stream join processor is used to join two event streams based on the attribute values of their individual events. Each streams should associate an event window in order to join the two streams. The join condition is evaluated for each input events with the events in other streams event window and all the matched events are added to a new event stream. Sample stream join query is shown in Listing 4.4. In the sample query, StockQuotesStream, which alias to *sqs*, is joined with HighFrequentTweetStream, which alias to *hfts*, using the join condition `sqs.symbol == hfts.company`. When an input event from stream StockQuotesStream comes to the join processor, all the

events in HighFrequentTweetStream event window which having company attribute value equal to symbol attribute value of this event are selected and written into InterestingStockQuotesStream along with this event. In Siddhi, the output event from join processor is called *StateEvent* and it wraps both input event and the matched event from window. The selection from event attributes form both events in StateEvent happens at the Selector processor.

```
1 from StockQuotesStream#window.length(100) as sqs
2     join HighFrequentTweetStream#window.length(1000) as hfts
3     on sqs.symbol == hfts.company
4 select sqs.symbol as company,
5        sqs.price as lastTradedPrice,
6        hfts.words as wordsTweeted
7 insert into InterestingStockQuotesStream;
```

Listing 4.4: SiddhiQL definition of a stream Join.

There are two scenarios that joining of the events happens in the join processor, (i) when a new event comes to the join processor from either of input streams, (ii) when an event in either of event windows leaving the join processor when it is expired. This behavior can be controlled by the user at the definition of the join query. It is possible to restrict from which events the join operation is triggered, either from input events or from expired events and it is also possible to control events from which event stream can trigger the joining process. Moreover, user can define a time restriction on the matched events saying the two matched events should be with in a given time difference. Event flow of stream join processor is shown in Figure 4.10 for (a) join with input events and (b) join with expired events.

In single-threaded mode, Siddhi process events from both input streams sequentially at the exact order that they are arrived to the system. In multi-threaded mode, two input streams should be processed in parallel and the join stream should synchronize the access to the join stream processor by two threads. Synchronization between two threads when high frequent input stream processing is so expensive, hence multi-threaded mode is currently not implemented in Siddhi for stream join processors.

Our approach of implementing parallel event processing GPU algorithm for stream join is based on the same concept we followed in event window processor; output event calculation and event window update. We process two event streams in parallel in two separate `GpuStreamRuntimes`. These runtimes have two `GpuStreamReceivers` both subscribed to their respective input stream's `StreamJunction`. It is also possible that these two `GpuStreamReceivers` subscribe to the same `StreamJunction` if they are joining the events from the same event stream. As shown in Figure 4.11, both `GpuStreamReceivers` wrapped in a new query runtime called `GpuJoinStreamRuntime`.

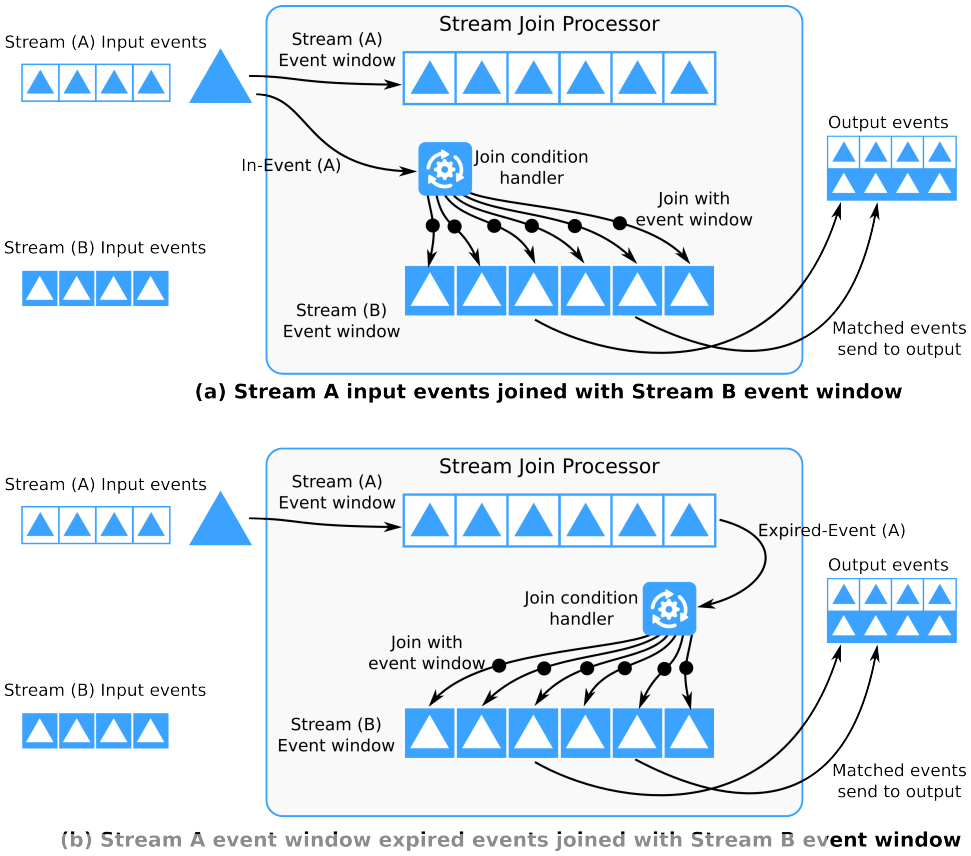


Figure 4.10: Event stream join processor joins two event streams based on a join condition. Joining of events happens at two scenarios; (i) new event comes to the join processor, (ii) an existing window event leaving the join processor when it is expired. At both scenarios the particular event is joined with events in other stream's event window.

Inside `GpuStreamReceivers`, using GPU event processing library individual GPU event processing constructs are created and added to each runtimes. Both `GpuStreamReceivers` using the same GPU stream join processor instance so they both have access to other streams' event window. Since timestamps of each input event can get the same timestamp value if the input events arrive at high input rate, when entering to `GpuJoinStreamRuntime`, each input event from both event streams get a 64bit sequence number that is unique inside that `GpuJoinStreamRuntime`. This sequence number is later used in CUDA kernels to identify causal ordering of events in both input streams.

GPU join processor contains event windows for both input streams and a join condition handler. There are several possible configurations in the GPU join processor by changing join triggering parameter to any of:

- Trigger joining only from events coming from left event stream
- Trigger joining only from events coming from right event stream

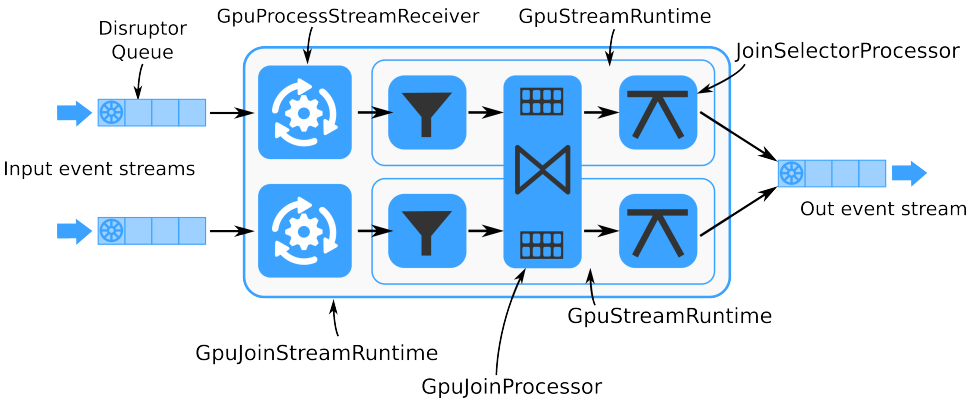


Figure 4.11: Siddhi GPU event processor model for stream join processor.
There are two separate `GpuStreamRuntime`s for each input streams and they are both wrapped in a `GpuJoinStreamRuntime`. Both `GpuStreamRuntime`s configured to use same join stream processor inside the GPU event processing library. There is a custom selector processor inside each `GpuStreamRuntime`to de-serialize and build output events from output event memory buffer received from GPU event processing library.

- Trigger joining from events coming from both event streams

and by changing event source parameter to any of;

- Join only with input events
- Join only with expiring events
- Join with both input and expiring events

For each of these configurations, there is a separate CUDA function implemented to generate output events using input events and event window. Depend on the configuration used, total GPU memory allocation can be differ. GPU join processor allocates GPU memory for both event windows and output event buffer. Each event stream will output input event batch size into number of events in the other event streams window number of events. This count is doubled if join is triggered for both input events and output events. Since each event stream do the joining in parallel they output their joined events into two separate output event buffers. These two event buffers are then processed in parallel by two special event selector processors called `GpuJoinSelectorProcessor`. `GpuJoinSelectorProcessor` de-serialize output event buffer and construct Siddhi representation of output events. All the event aggregation happens inside this selector processor. Finally, all output events are published to stream junction of respective output stream.

As mentioned earlier, there are two phases of GPU stream join processor; output event calculation and event window state update. In the first phase, each stream will launch number of GPU threads matched to number of input event count and each GPU thread is assigned an input event in input event buffer. In the earlier version

of this algorithm each GPU thread joined its assigned input event with all the events in other stream's event window sequentially. If there are lot of events in the event window, this sequential processing consumes lot of time. The algorithm is changed so each thread will process part of the event window, called "work size", by joining it with assigned input event. With the latest version of the algorithm `(input event count * work size)` number of GPU threads invoked and `(other stream event window size / work size)` number of GPU threads, called "work group", assigned one input event form the input event batch (see Figure 4.12). Each GPU thread in work group executes Algorithm 4.6.

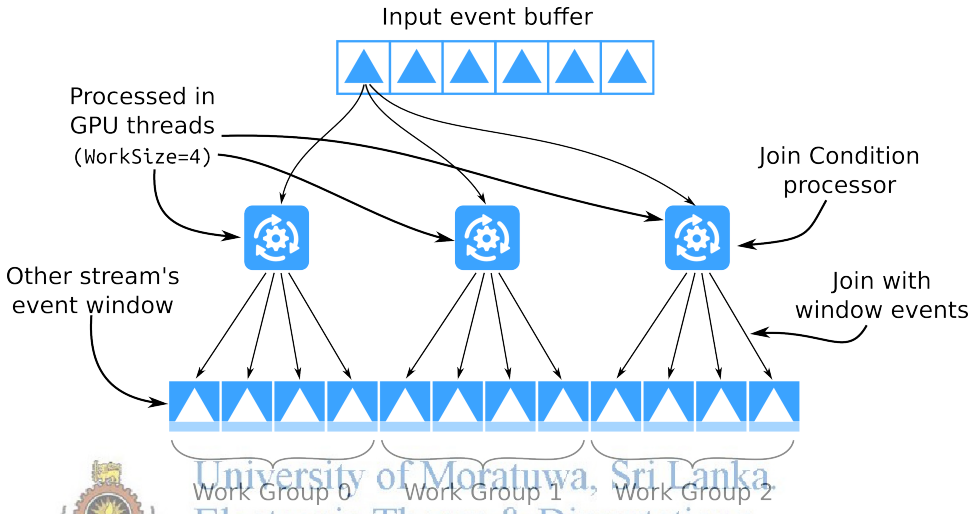


Figure 4.12: GPU thread allocation for join output event calculation phase. Number of input events into work size number of GPU threads created and work group size GPU threads assigned one input event from input event batch. Each GPU thread join its assigned input event with work size number of events in other stream's event window.

The second phase of join processor is same as the second phase of event window processor. So they both uses the same algorithm to update post-process state of their event window.

4.5.1 Concurrent Event Window Access

Concurrent access to event window buffer of other stream should be synchronized as they are accessed by two threads. It was observed that, if we consider one event window, even if both threads are reading event data from the event window, only one thread is actually updating the event window at window state update phase. The GpuStreamRuntime that handles particular stream is responsible for updating its event window. It is possible to use a locking mechanism to access both event windows, but that would yield significant performance overhead. So a lock-free algorithm was used to access event windows.

Algorithm 4.6 StreamJoinOutputEventsCalculation

Input: $inputEventBuffer$
Input: $inputEventCount$
Input: $sizeOfInputEvent$
Input: $eventWindowBuffer$
Input: $windowLength$
Input: $windowRemainCount$
Input: $otherWindowBuffer$
Input: $otherWindowLength$
Input: $otherWindowRemainCount$
Input: $otherStreamSizeOfEvent$
Input: $joinCondition$
Input: $withInTime$
Input: $outputEventBuffer$
Input: $sizeOfOutputEvent$
Input: $workSize$
Input: $workGroupSize$

1: $workerCount \leftarrow otherWindowLength/workSize$
2: $windowStartEventIndex \leftarrow (threadIdx\%workerCount) * workSize$
3: $inputEvent \leftarrow inputEventBuffer[sizeOfEvent * threadIdx/workerCount]$
4: $outputEventSegment \leftarrow outputEventBuffer[sizeOfOutputEvent * threadIdx * workSize]$

5: $otherWindowFillCount \leftarrow otherWindowLength - otherWindowRemainCount$
6: **if** $windowStartEventIndex < otherWindowFillCount$ **then**
7: $workStart \leftarrow windowStartEventIndex + workSize$
8: $minValue \leftarrow min(workStart, otherWindowFillCount)$
9: $windowEndEventIndex \leftarrow minValue$
10: $matchedEventCount \leftarrow 0$
11: $index \leftarrow windowStartEventIndex$
12: **for** $index < windowEndEventIndex$ **do**
13: $windowOffset \leftarrow otherStreamSizeOfEvent * index$
14: $otherWindowEvent \leftarrow otherWindowBuffer[windowOffset]$
15: $segmentOffset \leftarrow sizeOfOutputEvent * matchedEventCount$
16: $outputEvent \leftarrow outputEventSegment[segmentOffset]$

17: $timeDiff \leftarrow (inputEvent.timestamp - otherWindowEvent.timestamp)$
18: **if** $inputEvent.sequence > otherWindowEvent.sequence \& timeDiff \leq withInTime$ **then**
19: $isMatched \leftarrow JoinConditionEvaluate(inputEvent, otherWindowEvent)$
20: **if** $isMatched == True$ **then**
21: *Copy attributes of inputEvent and otherWindowEvent to outputEvent*
22: $matchedEventCount \leftarrow matchedEventCount + 1$
23: **end if**
24: **end if**
25: **end for**
26: **end if**

As shown in Figure 4.13, Double Buffering mechanism for event window is used to provide concurrent access to event window data. For each event window, there are two identical sized event buffers allocated in GPU memory. Pointers to these buffers are stored in an array of size two and these buffers are accessed using two integer buffer indexes which called read-only buffer index and read-write buffer index. When it needs to access read-only buffer, the read-only buffer index is used as the offset of buffer pointer array and get correct event buffer index. Same mechanism is used to access read-write event buffer.

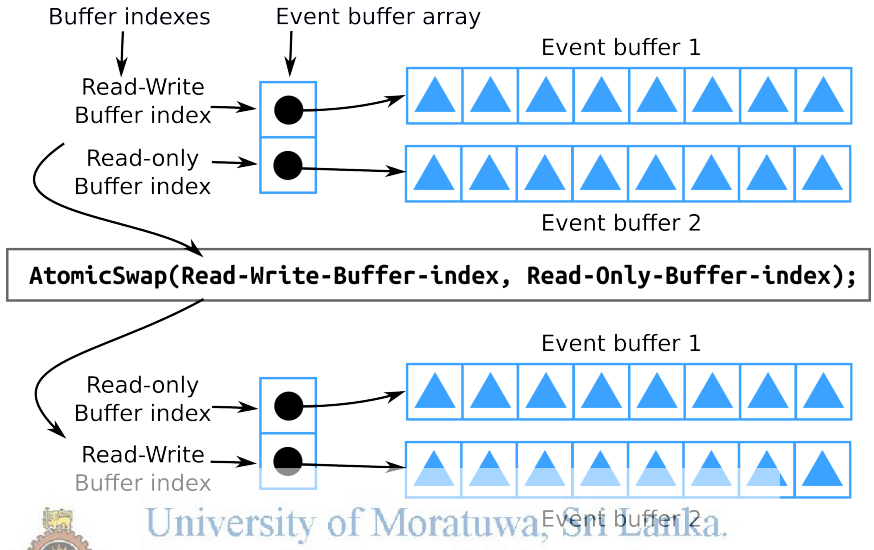


Figure 4.13: Event window double buffering.

To manage concurrent access to event window from both event streams of join processor, Double Buffering is used. One event buffer is in read-only mode while other buffer in read-write mode. Read-write buffer is updated and index to two buffers atomically swapped to reflect changes to read-only buffer.

A CUDA kernel always use read-write event buffer when accessing its own event window and use read-only event buffer when accessing other stream's event window. At the event window state updating CUDA function, the read-write buffer is updated to reflect the latest changes and once update complete the two buffer indexes are atomically swapped in host side. So now read-write buffer index will give previous version of the event buffer while read-only buffer index will give latest version of event buffer. After swapping indexes, the GpuStreamRuntime sync two buffers copying read-only buffer to read-write buffer using CUDA device-to-device memory copy mechanism. This way thread-safe concurrent access to two event windows are assured while avoiding any performance overhead. The only concern about this mechanism is increased memory requirement for event windows. Even with the twice memory requirement, performance gain we achieved is significant.

4.6 Improving Performance in Event Processing Library

Apart from the performance improvements done in the Siddhi side `GpuStreamRuntime`, there are several other mechanisms we used to improve query processing throughput in GPU event processing library.

- **Use of pinned CPU memory**
  For all the memory allocations for event buffers in CPU memory, we have used non-pageable (pinned) memory. So this avoids memory paging overhead for these frequently accessed memory regions and also enable us to use overlapped GPU operations.

- **Overlap operations in CPU and GPU**
  CUDA runtime API provides both blocking and asynchronous methods for same tasks like memory copying and kernel invocations. In the main control flow of the GPU processors, asynchronous CUDA runtime functions were used to overlap GPU operations with CPU. For example, there are scenarios which we need to copy two event buffers into GPU and invoke two CUDA kernels. In such situations, first memory copying to GPU device memory is done asynchronously and then invoke the two CUDA kernels. So copying of one event buffer happens while one CUDA kernel is running at GPU.

- **Coalesces memory accesses**
  All the event buffers are aligned to 8 or 16 byte boundaries, so it coalesces the memory accesses of the threads within the warp.

- **Multiple GPU device utilization**
  If there are more than one GPU devices attached to the server, users are able to utilize them just by specifying the suitable GPU device ID for each query. If device id is not specified, best GPU device in terms of GFLOPS is used. For example, if there are two GPU devices attached to the server and one of them has more memory and higher compute capability than the other, users are able to specify that GPU device for queries that need more memory or higher event processing throughput, while other queries can use the other device.

- **Reducing divergent execution**
  Divergent execution is when two GPU threads within the same thread block need to fetch different instructions at a particular time. For example, if there is an if statement in a CUDA kernel whose condition is dependent on the thread index, one thread will execute the body of the if and another will execute the else part. When this happens it is called the execution has diverged. GPU executed diverged path sequentially, so if there is lot of work done in diverged code block, the parallelism is reduced. Special care has taken in order to minimize

diverging code blocks. But it is inevitable to remove some of the diverging paths. For example, when filtering events based on their event attribute values, two consecutive input events can have totally different set of attribute values. So it is possible the two GPU threads that process these events with in a same warp and become diverge.

Chapter 5

Evaluation

To demonstrate the effectiveness of our work, we have performed an extensive experimental evaluation of our implemented GPU event processors using wide verity of event queries. Our main goal of this research is to find the possibility of using GPU hardware devices provided computing power to increase the event processing performance in Siddhi CEP engine. There are two measurements of event processing performance; (i) the throughput of event processing and (ii) the latency of event processing. In this research, we choose to increase the event processing performance of Siddhi by increasing the throughput of event processing.

Our evaluation had several goals. First, we wanted to compare our work with the existing sequential and parallel event processing algorithms implemented in Siddhi. Second, we wanted to understand and analyze our design and implementation choices that has impact on the performance. Finally, since the event processing performance is largely influenced by the number and the complexity of deployed queries and the type and load of the workload, we wanted to explore the parameter space to identify in which cases it is more profitable to use particular algorithm or hardware architecture.

This chapter presents the approach we took to evaluate our proposed solution for GPU event processing. Evaluating query processing performance of a CEP engine is strongly influenced by the input workload. A publicly available data set has been used as the workload of the experiments. Section 5.1 describes this workload. In Section 5.2, we explain how we carried out the experiment and the configurations of out experiment setup. Section 5.3 presents the evaluation results for the Event Filter Operator and analysis of the results while Section 5.4 presents the evaluation results and analysis for Stream Join Operator. We have also carried out an experiment to evaluate query processing performance when there are mix of event queries in the system, which we present the analysis in Section 5.5.

5.1 Analysis of the Workload

With the increased usage of event-based systems in many application domains ranging from real time monitoring systems in production, logistics and networking to complex event processing in finance and security, there is a appealing requirement for a standard benchmarking platform for event-based systems. In complex event processing domain alone, there are more than twenty CEP products, developed by both industry and the academic community, currently available in the market and there are no any standards and widely accepted way to compare these CEP engines. Usually vendors use their own benchmarking to demonstrate capabilities of their products. This problem is well recognized within the event processing community and there is on going process within "Event Processing Technical Society (EPTS)" to standards for Event Processing Systems [74].

ACM International Conference on Distributed Event-Based Systems (DEBS)* is a well-recognized conference and workshop on event-based computing which inaugurated on 2007. The conference provides forum for academia and industry to exchange and publish ideas. Starting on 2011 DEBS conference host an event called "Grand Challenge" in order to provide a common ground and evaluation criteria for a competition aimed at both research and industrial event-based systems. In Grand Challenge, they provide a real-world problem to be solved using event-based systems. A real-world dataset is also provided as the workload to evaluate solutions from participants. Since participants from both industry and academia submit their solutions and evaluate using a same data set, DEBS Grand Challenge competition data set can be used as a standard data set for evaluating and comparing event-based systems.

Our evaluation is based on the data set provided for 2013 Grand Challenge competition†. The challenge for 2013 was to demonstrate the applicability of event-based systems to provide real-time complex analytics over high velocity sensor data along the example of analyzing a soccer game. The provided real-world data set was recorded from a number of wireless sensors embedded in the shoes and a ball used during a soccer match. These sensor data spans the whole duration of the game and the maximum data rate reaches roughly 15000 sensor events per second.

Event schema of each sensor data is as follows;

`sid, ts, x, y, z, |v|, |a|, vx, vy, vz, ax, ay, az`

- `sid` - sensor id which produced the position event
- `ts` - timestamp in picoseconds

*http://debs.org/
†http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails

- `x, y, z` - position of the sensor in mm (the origin is the middle of a full size football field)
- `|v|` - velocity in μm/s
- `vx, vy, vz` - direction by a vector with size of 10,000
- `ax, ay, az` - absolute acceleration and its constituents in three dimensions in m/s^2

There are 49,576,080 events in the data set that range 71 minutes and 4 seconds. So in average there are 11,626 events per second. If an event processing system needs to provide real-time analytics processing these data set, it has to have event processing throughput of at least 11,626 events per second. In our evaluation we used this calculations as the baseline for deciding the event processing performance of our proposed solution.

Similar to the lack of standard workload for event processing systems, there is lack of standard measurement for the complexity of the event processing rules/queries. Having a standard measurement for the complexity for event processing queries helps to compare event processing performance of two queries. For evaluating this work, we have developed SiddhiQL queries for some of the event queries defined in DEBS 2013 challenge. Since our work does not implement all the event processing operators to run on GPUs, we are unable to get the performance measurements for complete individual queries which completely running on GPUs. Instead we have developed partial event queries to evaluate individual event processors and compared their performance by running them in GPUs and CPUs.

5.2 Experiment Setup and Methodology

Usually in Siddhi deployments, each use case is wrapped in their own execution plan. For an example, in DEBS 2013 challenge, there are several queries like running analysis of players, ball possession, shots on goal, etc., on which participants have to provide data as an output streams. If used Siddhi CEP as the event processing system, each of these queries may need several SiddhiQL queries to process input events and produce output events. So all the SiddhiQL queries that work on same higher-level query is wrapped in a execution plan. Depend on the resource usage, there can be several execution plans in one Siddhi instance.

Siddhi allocates resources per execution plan. Each execution plan has its own thread pools, executor service pools, input stream queues and output stream queues. Except for heap memory allocated for the JVM instance that Siddhi is running, execution plans do not share anything among them.

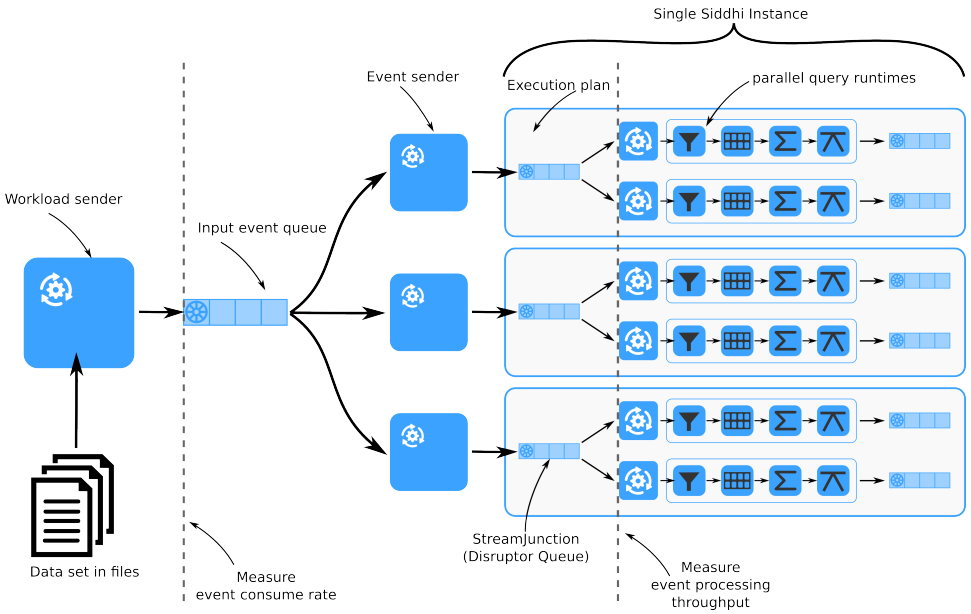


Figure 5.1: Architecture of the experiment setup.
In the experiment, each use case is configured in their own execution plan and there are multiple instance of same execution plan running in same Siddhi instance.

In our experiments, we configured each use case in one execution plan and there are multiple instance of same execution plan running in same Siddhi instance as shown in the Figure 5.1. Intension of running multiple parallel use cases was to measure how our algorithms scale with multiple high complex queries.

To simulate a real-world scenario, in this experiment setup, we have fed the workload into a queue where the application reads and put into Siddhi's stream junction. Then we measured time it takes to process all the input events put into this queue. Using this time measurement we can calculate event consume rate of the each configuration. Using the single-threaded mode as the base, we calculated how much speedup we gain for CPU multi-threaded mode, GPU single device mode and GPU multi device mode. The main event queue is a Java ConcurrentBlockingList. So if the event processing cannot keep up with event produce rate, there will be queue build up. If there is a queue build up, there will be increased latency when enqueue an event into the queue. We have measured and plotted the average event enqueue latency for each configuration.

We have also measured average time it takes to process an event batch which received to *StreamReceiver*. Batch size divided by this measurement gives the actual event processing throughput of Siddhi. In GPU event processors, time it takes to process an event batch includes individual time measurements of;

- time to serialize events in the batch,
- time to transfer serialized events to GPU memory,

- actual event processing time in GPU,
- time to copy result events back into CPU memory,
- and time to de-serialize and create event objects and send to output stream junction.

We have individually measured average time of each of these phases to identify what phase contribute lot to total processing time and identified possible places to improve. Individual time measurements are plotted against each configurations.

To avoid any bias in server load, we have repeated all the tests presented in this chapter five times and used the average of the measured values and the 95% confidence interval was always below 1% of the averages of measured values. There was no any other extra services running on the test server other than default OS services.

All the tests we explain in this section were executed on a 64bit Intel Core i7 950 CPU, with 8 cores running at 3.07GHz base frequency, and 7GB of DDR3 memory. To compile and run Siddhi, we used 64-Bit Oracle Java compiler and runtime version 1.6.0_26. CUDA SDK Version 5.5.0 for 64-Bit Linux was used to compile and run CUDA kernels and GPU event processing library and internally it uses gcc version 4.6.3. GPU device attached to the testing server was Nvidia GeForce GTX 480 with 480 CUDA cores and 1.5GB DDR3/GDDR5 RAM. Two GPU devices of Nvidia GeForce GTX 480 were used to do multi-device performance testing. To avoid the impact of Java garbage collection as much as possible we have allocate 6GB of heap memory in each of these tests and almost all the tests have uses less than the allocated amount of memory.

5.3 Filter Query Performance

```
1 define stream sensorStream ( sid string, ts long,
2         x int, y int, z int,
3         v double, a double, vx int, vy int, vz int,
4         ax int, ay int, az int, tsr long, tsms long );
```

Listing 5.1: Sensor stream definition.

For all the perfromance tests presented in this chapter, we have used Siddhi stream definition for sensor data stream as defined in Listing 5.1. This is the only input event stream for our test setup. When serialized using our event serialization technique, described in Section 3.3.1, each input event takes 104 bytes. So each memory allocation for event buffers are multiple of 104 bytes.

Table 5.1: Filter query test setup parameters.

Number of input events	49,576,080
Input event time span	71 minutes and 4 seconds
Average input event rate	11,626 events per second
Disruptor buffer size	8192
GPU thread block size	128 threads
GPU process event batch sizes	2048 events
Concurrent query count	2 to 50

All the test setup parameters for filter query evaluation are summarized in Table 5.1. In the early stage of the research we have evaluated the effect of GPU thread block size by running the same experiment with different GPU thread block sizes. It was found that thread block size 128 to 256 gives the best event processing throughput for most of the cases (see Figure 5.2). We have used thread block size of 128 for all of the tests we have presented here. Using the 128 threads per block than using 256, gives better chance of running multiple parallel queries in same GPU device.

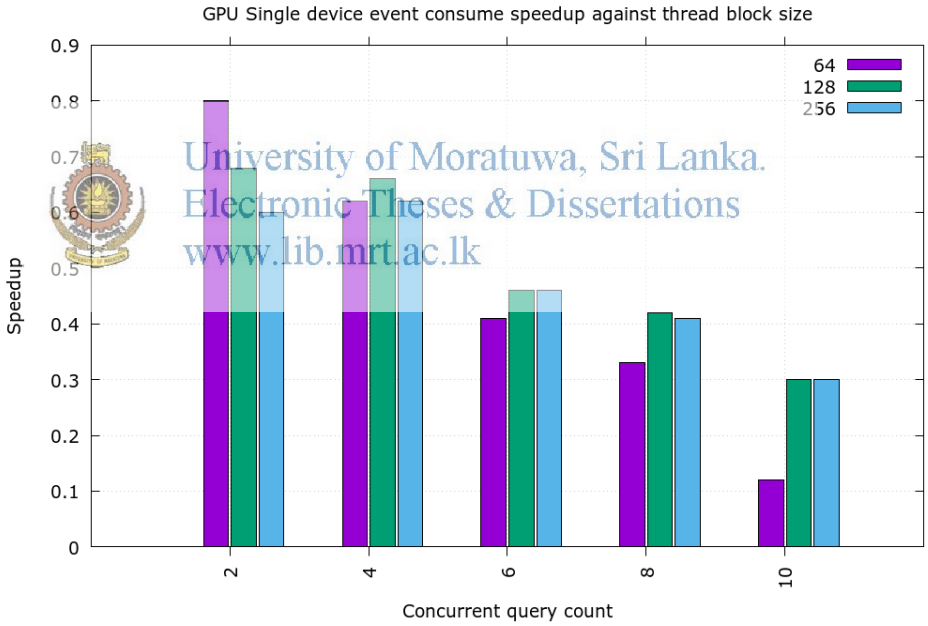


Figure 5.2: Filter query event consume rate speedup for different GPU thread block sizes.

```
from sensorStream[sid == '4' or sid == '8'
        or sid == '10' or sid == '12']
select sid, ts, x, y
insert into ballStream;

from sensorStream[sid != '4' and sid != '8'
        and sid != '10' and sid != '12'
        and sid != '105' and sid != '106']
select sid, ts, x, y
insert into playersStream;
```

Listing 5.2: SiddhiQL event filter queries used for evaluation.

For evaluating GPU filter processor, SiddhiQL filter query defined in Listing 5.2 is used. In practical usages filter queries are rarely used individually, but used in conjunction with other event processor operators. But intention of evaluating GPU filter processor is to compare performance gain or loss which can achieve by using GPU hardware. So evaluating individual filter query make sense in this scenario. Moreover, some of the GPU parallel algorithms developed for filter processor are used in other event processors like event stream join processor. So by individually evaluating filter query, we can isolate these algorithms' effect on the performance of event processing.

5.3.1 Event Consume Speedup Analysis

The Figure 5.3 analyzes the speedup of input event consume rate of event filter processor relative to single-threaded mode for different concurrent query counts. As stated in the previous sections, input events are queued to a common queue by the workload sender, where these events are then de-queued and fed into different parallel execution plans. If events are de-queued from the common queue as soon as they are enqueued means, system can process events in a higher rate than the event produce rate. If this scenario happens, it is called a "Speedup" of event processing. Event consume rate is measured at the workload sender by measuring size or emptiness of the common event queue. Total time it takes to process all enqueued input events, which also means time it takes to empty the common input event queue, is measured and along with the number of total input events the input event consuming throughput is calculated. If this value is greater than average event input rate, then there is a speedup of event consuming. We can also safely say there is a speedup of event processing if there is a speedup of event consuming.

Note that the concurrent query count is not incrementing uniformly. In order to do more test iterations we have skipped some of the configurations in between.

Siddhi event filter processor in single-threaded mode outperform both multi-threaded and GPU event processing throughput when there are a few filter queries. This is

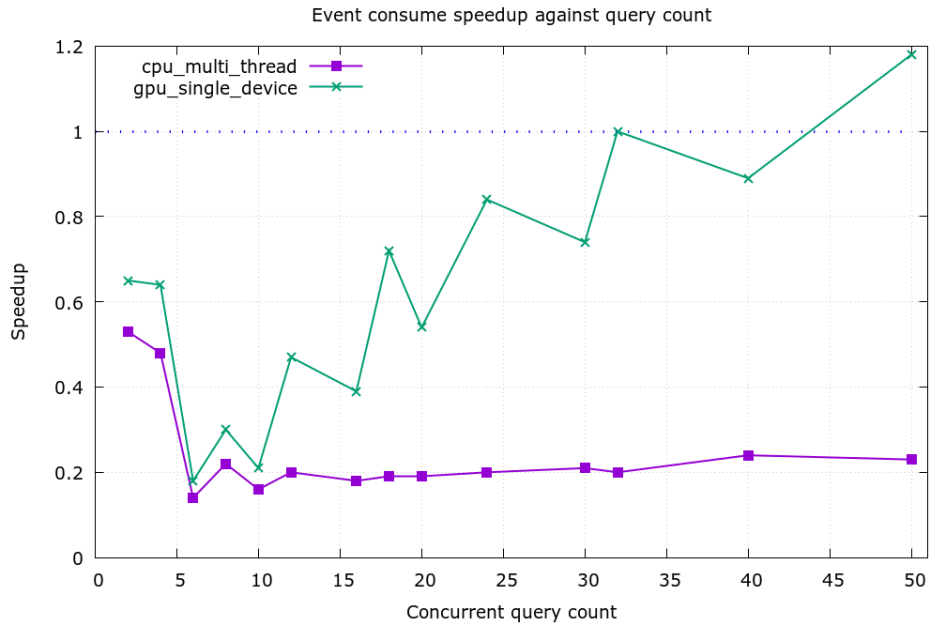


Figure 5.3: Filter query event consume rate speedup against concurrent query count (event batch size 2048 events).

because there are no any internal event queuing overhead in single-threaded mode as there is for multi-threaded mode. But once the number of concurrent queries increases, the effect of queuing overhead is overcome by the effect of parallel processing.

GPU filter processing is always has higher speedup than the multi-threaded mode. When the number of concurrent queries increases, CPU multi-threaded mode does not significantly increase its event processing speedup. But our GPU based filter processing continues to increase its event processing speedup with the increase of number of concurrent queries. So, in order to gain increased event processing performance for filter processor, the CEP engine should be loaded with lot of concurrent queries.

We have done performance test to evaluate the effect of event batch size by running the filter query in GPU with varying batch sizes. The results are shown in Figure 5.4. As per the performance results, there is no significant improvement in event consume speedup if we increase the size of event batch. So event batch size of 2048 is used for all the tests for filter processor.

5.3.2 GPU Processing Time Analysis

The lower event consume speedup and event consuming throughput of GPU filter processor is due to several reasons. First, the event serialization and de-serialization times consume more time from total GPU event processing time. This is clearly visible in Figure 5.5. And with the increase of number of parallel event queries, the average

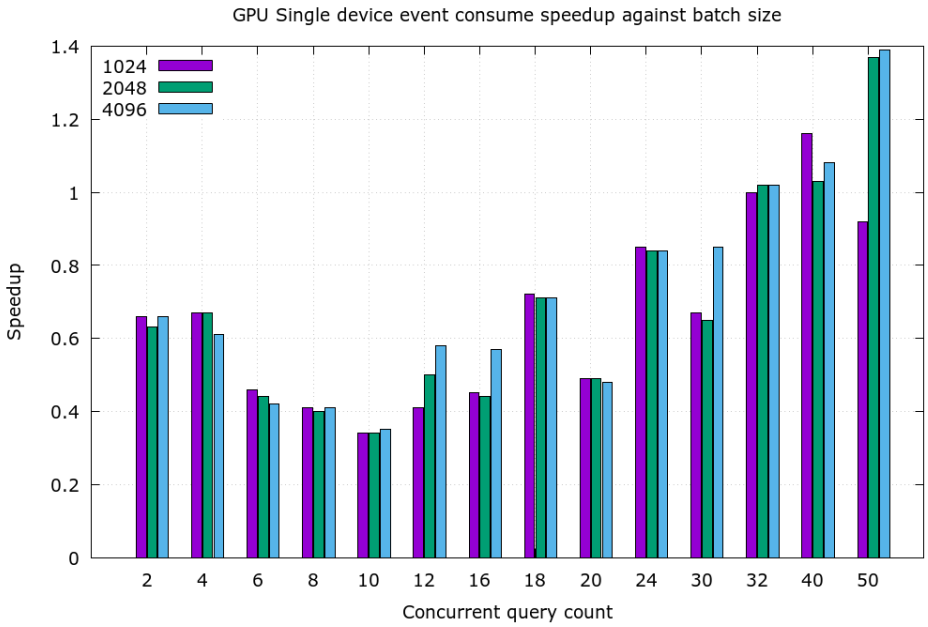


Figure 5.4: Filter query event consume rate speedup for different event batch sizes.

serialization and deserialization time increases. This behavior is due to the fact that event serialization and de-serialization happens in CPU and when number of queries increase, the GPU runtime has to compete with other parallel threads to do its tasks.

For serialization and deserialization, we have used dynamically generated Java bytecode to optimize serialization and de-serialization logic. This Java bytecode was generated at runtime using the event schema details for the particular event stream. So we have hard-coded like serialization logic in our application, that is optimized for the particular schema of the event stream. Even with this optimizations, these two phases consume more percentage of total GPU processing time. Figure 5.6 shows the percentage time spent in serialization, actual GPU event processing and de-serialization phases. The actual GPU processing time consumes about 10% of total processing time irrespective of number of queries. This is because, in GPU, even if there are more queries to process the task is done in parallel and GPU can accommodate more parallel tasks than CPU does.

Inside the actual GPU processing time, most of the time consumed by event data transfer from CPU memory to GPU memory. About two third for the total actual GPU processing time was taken by event data transfer to CPU memory to GPU memory. This is shown in Table 5.2. Actual Filter kernel time accounts for only about 30% of whole time and it is 32.542 microseconds on average. GPU device to CPU memory copy time is reduced because of the optimization we have done for filter result events. In individual filter queries we are not copying the actual output events from GPU to CPU, instead we copy only an array of integers representing the matched event index

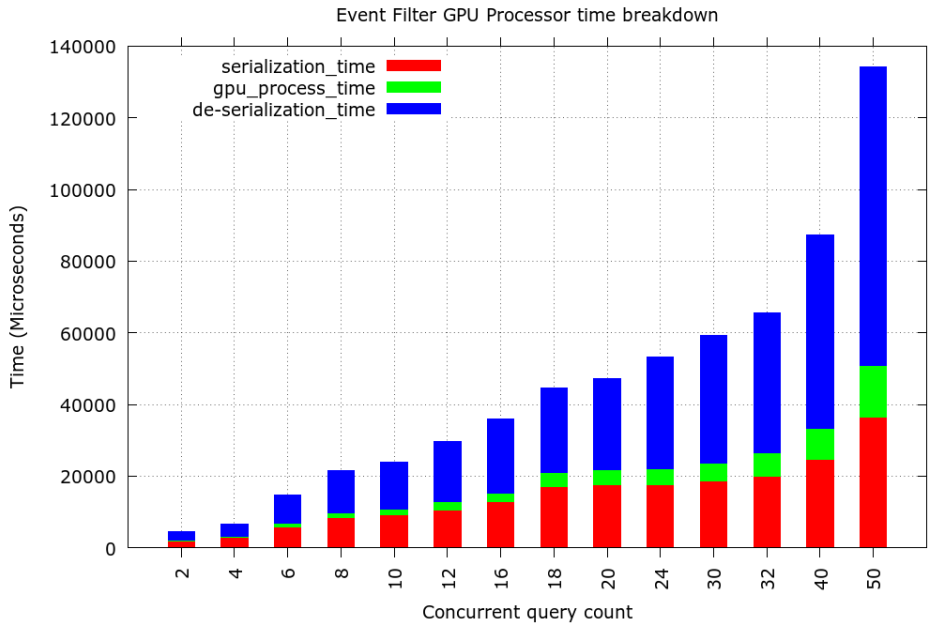


Figure 5.5: GPU Filter processor: GPU processing time breakdown (event batch size 2048 events).

in the input event buffer. Using this array of indexes we can create actual Siddhi events at the de-serialization phase.

Table 5.2: Filter query GPU processing time breakdown

GPU Kernel	Time(%)	Average Time
[CUDA memcpy HtoD]	66.73	72.938us
ProcessEventsFilterKernel(KernelParameters*, int)	29.75	32.542us
[CUDA memcpy DtoH]	3.52	3.8490us

The final reason for low event processing performance in GPU filter processor is due to CUDA kernel path divergence. CUDA profiler shows that there is 10% branch divergence in CUDA filter kernel. Filter kernel has control flow branches that depend on attributes of individual input events and in the real-world workload consecutive events with same event attributes are rare. We have re-factored the kernel code to avoid and resolve diverging paths up in the CPU side, but it is hard to eliminate all the path divergence.

While there is no performance gain by using GPU processing, this results show us where not to use GPU processing and when should we start using GPU processing.

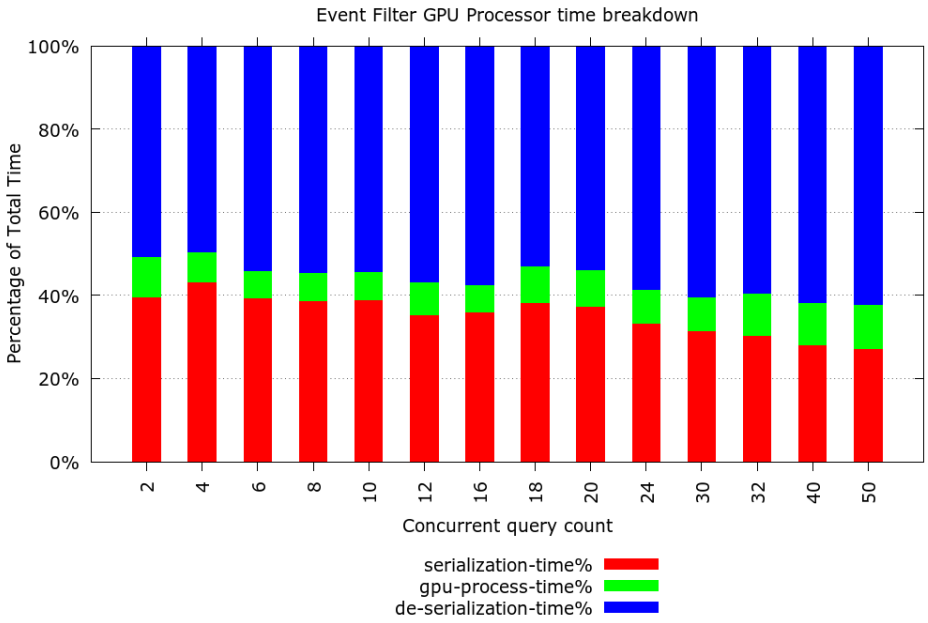


Figure 5.6: GPU Filter processor: GPU processing time as a percentage of total processing time (event batch size 2048 events).

5.4 Join Query Performance

Stream Join operator is inherently a complex operator than the previous event operator that we have evaluated. This is because stream join operator involves more than one event streams and should process events from both these input event streams simultaneously. Unlike the previous event operator, stream join operator has obvious data parallel use case where each input event needs to be process against all of the events in a event window. If event window size is larger, this cross joining process takes higher processing time when processed sequentially. Our implementation tries to improve the performance of processing join queries by utilizing parallel hardware and parallel algorithms.

Table 5.3: Join query test setup parameters.

Number of input events	49,576,080
Input event time span	71 minutes and 4 seconds
Average input event rate	11626 events per second
Disruptor buffer size	8192
GPU thread block size	128 threads
GPU process event batch sizes	2048 events
GPU Selector work size	100 events
GPU Selector worker count	8
Concurrent query count	1, 2, 3, 4, 5, 6, 7, 8, 9, 10

The performance test carried out to evaluate the GPU stream join processor has the same test setup as the previous tests and uses the stream join query defined in Listing 5.3. All the test setup parameters are summarized in Table 5.3. GPU thread block size was fixed at 128 thread and the event batch size was 2048 events for all the test iterations.

As stated in earlier test analysis, varying batch sizes do not have much effect on the event processing throughput once they exceed 2048 events. So we are not presenting the measurements of each batch size, instead performance measurements are plotted only for batch sizes of 2048 events. Increasing the input event batch size also means higher memory allocation in GPU for input event buffers. Because there should be memory buffers pre-allocated for each input event batch. Having larger input event batch size means, there is less room for more parallel queries in GPU device.

For stream join operator, current Siddhi implementation dose not provide a parallel algorithm. Hence, we are comparing event processing performance only for single-threaded mode and GPU processing modes.

GPU algorithms developed for event window operator are not evaluated individually since event window operator is not used in event queries individually. Instead, they are used conjunction with other event operators like stream join operator. So in this experiment the GPU algorithms for event window operator are also evaluated along with the GPU stream join operator.

Parallel processing event stream join operator in multi-threaded mode does not produce the same joined output events as the sequential event processing. This is because of the random nature of event batching in multi-threaded mode. But the output events does not violate the causal ordering of their input events. This nature is inherent in multi-threaded processing in both the CPUs and GPUs.

```
1  from sensorStream[sid == '4' or sid == '8'
2          or sid == '10' or sid == '12']#window.length(200) as a
3  join sensorStream[sid != '4' and sid != '8'
4          and sid != '10' and sid != '12'
5          and sid != '105' and sid != '106']#window.length(200)
               as b
6          on a.x == b.x and a.y == b.y
7                  and a.ts > b.ts
8                  and (a.ts - b.ts < 1000000000)
9  select b.sid as psid, a.sid as bsid, b.ts as pts,
10         a.ts as bts, b.x as px, b.y as py, a.x as bx, a.y as by
11 insert into nearBallStream;
```

Listing 5.3: SiddhiQL Join query used for evaluation.

5.4.1 Event Consume Speedup Analysis

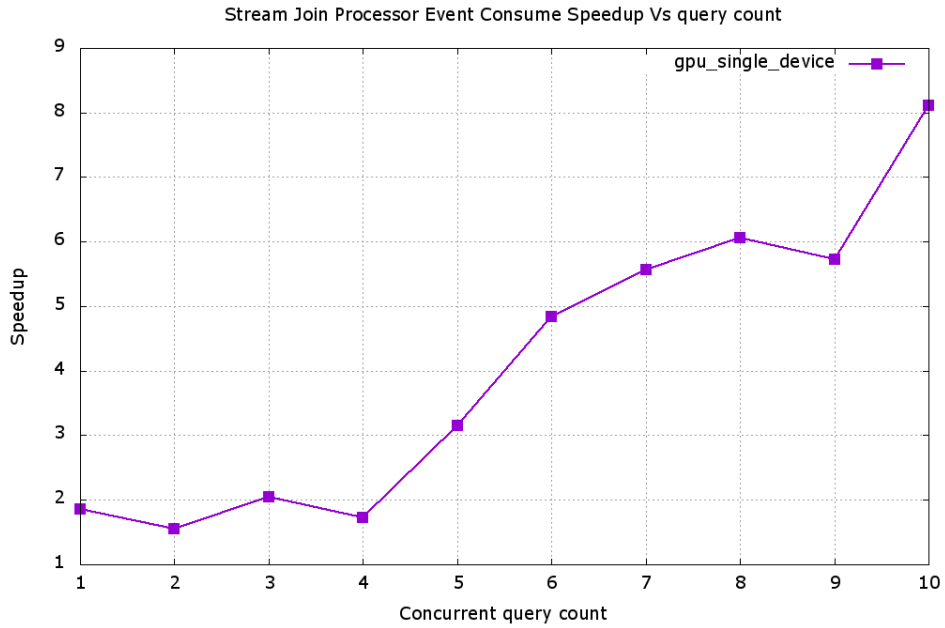


Figure 5.7: Join Query: Average event consume rate speedup against concurrent query count (event batch size 2048 events).

The Figure 5.7 analyzes the speedup of input event consume rate of join processor for different concurrent query counts. Input event consume rate denotes how quickly input events are consumed from the input event queue. As per the performance measurements, for a single join query, GPU join processor has achieved almost twice speedup than single-threaded mode processing when there are lower number of concurrent queries. With the increase of concurrent queries the speedup is also increasing.

In practical usage, once the input event consume rate speedup is at or below one, that means there should be queue build-up as the event processors cannot keep up with the event produce rate. Single thread CPU processing mode is not usable after four concurrent filter queries in this performance test. But the GPU join processor can maintain positive event consuming speedup until eight concurrent join queries.

To analyze how our algorithm scale with multiple GPU devices, we have done the same test using two identical GPU devices which we have used in earlier tests. The Figure 5.8 shows the average event consume rate speedup for CPUs, single GPU and two GPU devices. We have executed join queries in multiple of two and each GPU is assigned identical queries and there can be multiple queries assigned to a single GPU depend on number of concurrent queries. Almost all the time our algorithm has scaled twice with two GPU devices.

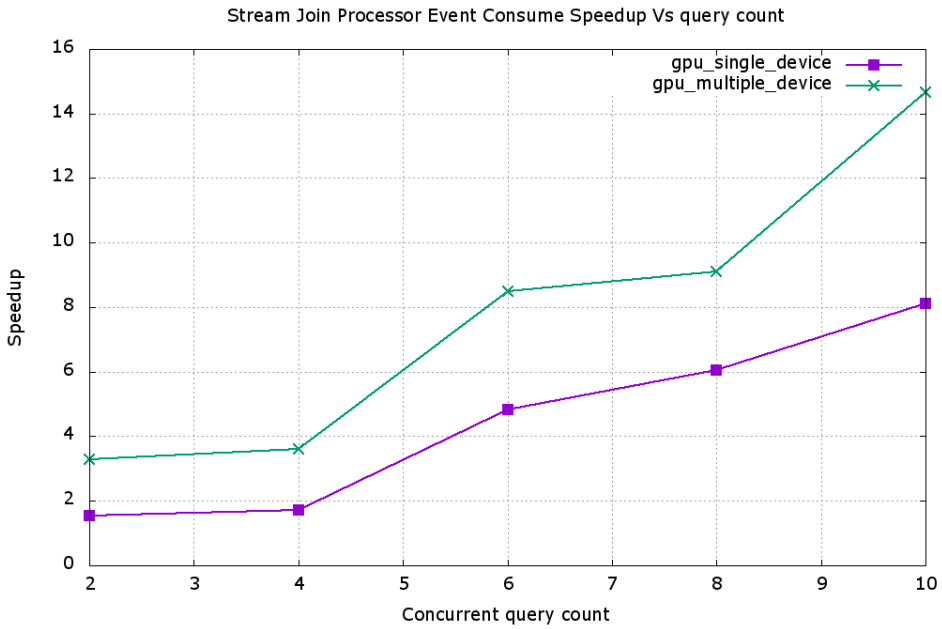


Figure 5.8: Join Query: Average event consume rate speedup for multiple GPU devices (event batch size 2048 events).

Once the input event consume rate is decreasing below 1, the input event queue is filled and the event producer has to wait until the event processors process some events and make space for new input events. This wait time for input event queue is measured and plotted against the number of concurrent query count in Figure 5.9. The queue publish latency graph is exactly matching with the event consuming speedup graph. In CPU single threaded processing mode, after query count four, the queue wait time is drastically increased because of input event consume rate is below one. For GPU event processors, the queue build-up is much less compared to CPU event processors. And the increase of queue build-up is also not rapidly increasing. This shows that GPU processors can maintain high input processing throughput even with multiple complex queries. The queue publish latency for two GPU deives scenario is always about half the latency of single device scenario.

Less queue build-up means less overhead on event producers. In a real-time high frequent event stream, if the input event queue is filled, either input events needs to be stored temporary until the event queue is get released or the input events should be discarded. Both options are not ideal when input events needs to be processed in-real-time and should not be discarded. GPU processing of events can cater for these kind of scenarios even with increased event processing latency.

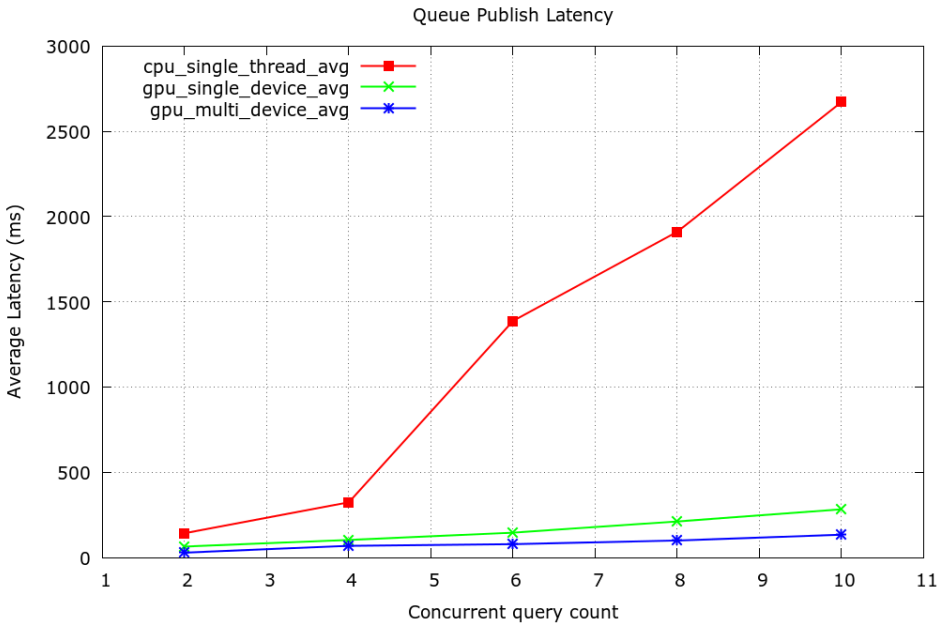


Figure 5.9: Join Query: Input event queue publish latency against concurrent query count (event batch size 2048 events).

5.4.2 GPU Processing Time Analysis

The rapid decrease of event consume speedup in GPU join processor is due to the effect of increase in GPU processing time for an individual join query. As shown in Figure 5.10, GPU processing time increases as the number of parallel query count increases. Profiling of CUDA kernels shows that most of the GPU processing time was consumed by the data transfer form GPU memory to CPU memory (Device-to-Host). Detailed breakdown of GPU processing time in GPU Join kernel is shown in Table 5.4. The table shows only percentage of time each kernel consume and the average time per kernel. The number of call to each kernel can be differ and usually the "CUDA memcpy DtoH" kernel has more calls than the others.

Almost 85% of the total GPU processing time of GPU Join kernel is consumed by Device-to-Host memory copy. Join kernel has a larger output event buffer than its input event buffer. This is because if all the input events matched with all the events in the event window, then there should be input event count times event window size output buffer. This is only for one stream of the join processor, for other stream also there is similar number of output events.

For example, for the join query defined for this test, if the input batch size is 4096 events, then there can be 1,638,400 total output events for this query. With the output event size of 44 bytes, the output event buffer should be 72,089,600 bytes (or 68.75 MBytes). CUDA profiler shows, to transfer this much of data from device to host at

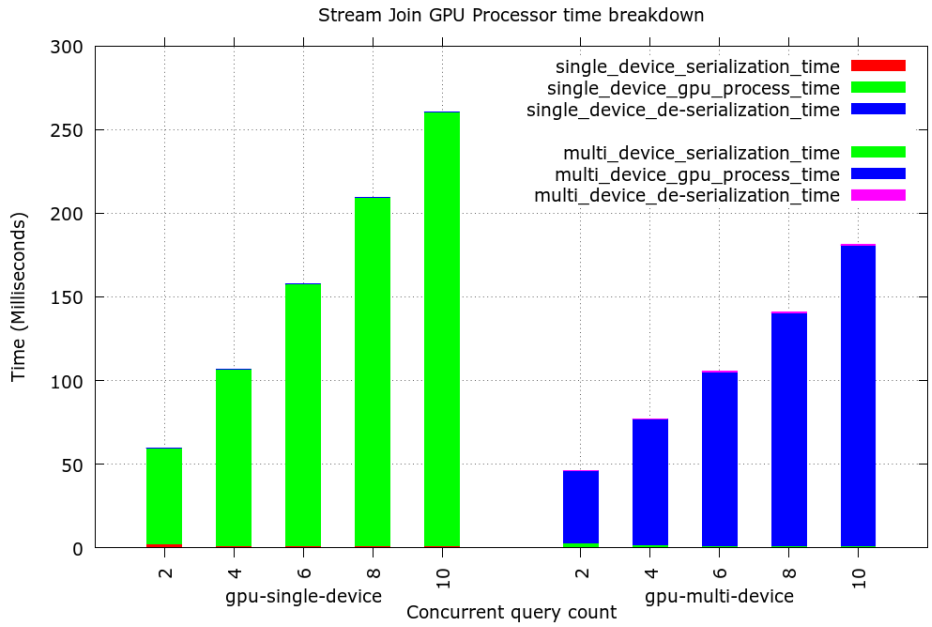


Figure 5.10: GPU Join processor: Processing time breakdown (event batch size 2048 events).

about 3GB/s, it can take maximum of 22 milliseconds. The effect of CUDA memory copy is more visible in this test because of the lack of multiple copy engines in the GPU device we used for this testing. There is only one copy engine in Nvidia GeForce GTX 480 GPU. So all the memory transfers from CPU to GPU and from GPU to CPU happens sequentially.

Table 5.4: Join query GPU processing time breakdown.

GPU Kernel	**Time(%)**	**Average Time**
[CUDA memcpy DtoH]	84.87	8.5689ms
ProcessEventsJoinRightTriggerCurrentOn	6.97	2.8136ms
ProcessEventsJoinLeftTriggerCurrentOn	5.89	2.3815ms
[CUDA memcpy HtoD]	0.70	139.26us
JoinSetWindowState	0.67	135.86us
FilterKernel	0.89	179.513us
[CUDA memcpy DtoD]	0.01	2.2150us

The Figure 5.10 shows the individual time components of GPU processing. This includes;

- time to serialize events in a event batch,
- time to transfer serialized events to GPU memory, time to invoke GPU kernels and processing and time to copy result events back into CPU memory,
- time to de-serialize, create event objects and send to output stream junction

As can be seen in the Figure 5.10, the event serialization and de-serialization takes negligible amount of time when compared to event transfer and actual GPU processing time. The less serialization and de-serialization time is due to the optimizations we have done for these two phases. The event serialization code is dynamically generated at runtime using a Java assembly code generation tool (Javassist) so that it has hard-coded like logic specific for the input events it is serializing. The same optimization is also applied to de-serialization phase. Moreover, the deserialization is done in parallel threads. The number of serialization threads can be configured per-query using query annotations.

5.4.3 Input Event Processing Throughput Analysis

Event processing throughput of GPU Join processor is shown in Figure 5.11. The throughput of processing each event batch is measured at *GpuStreamReceiver* for GPU event processors and at *StreamReceiver* for other event processors, and the average value is presented in the graph. The average input event rate of this workload is 11626 events per second. So if any configuration cannot achieve event processing throughput higher than this value, that configuration is not usable in real-world. In this test, the single threaded mode processing cannot achieve event processing throughput than the input event rate after four concurrent join queries. But the GPU join processor can maintain higher event processing rate than the input event rate for eight concurrent join queries.

The number of concurrent GPU event processors were limited due to memory resource limitation in GPU device. We could only run ten concurrent queries of the Join query defined for this test setup with 2048 input event batch size. Increasing the batch size to 4096, this count further reduced to five concurrent queries.

5.5 Query Mix Analysis

Previous two experiments evaluated the query processing performance of individual event operators. But in real-world use cases event queries in a single execution plan contains mix of different event operators. To evaluate this use case, an experiment was done using several event queries which uses different event operators. There are three event queries consist of different CEP operators we have implemented for GPU processing. These event queries are shown in Listing 5.4.

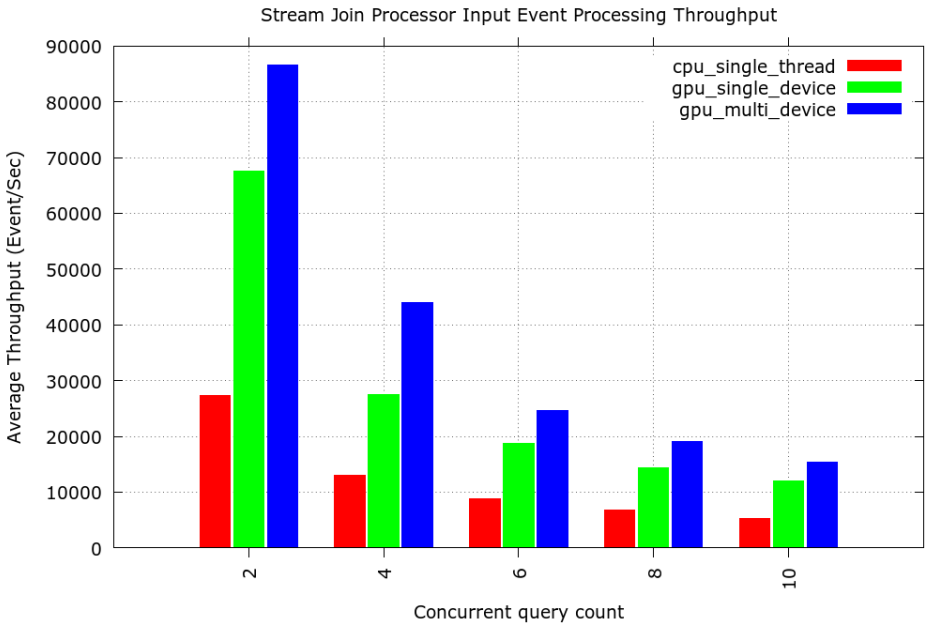


Figure 5.11: Join Query: Input event processing throughput against concurrent query count (event batch size 2048 events).

```
from sensorStream[(sid == '4' or sid == '8'
or sid == '10' or sid == '12')
and ((ts >= 107532955944241161 and ts <= 125572955944241161)
or (ts >= 130866391464034951 and ts <= 148796391464034951))]
select sid, ts, x, y
insert into ballStream;

from sensorStream[sid == '4' or sid == '8'
or sid == '10' or sid == '12']#window.length(200) as a
join sensorStream[sid != '4' and sid != '8'
and sid != '10' and sid != '12' and sid != '105'
and sid != '106']#window.length(200) as b
on a.x == b.x and a.y == b.y and a.ts > b.ts
and (a.ts - b.ts < 1000000000)
select b.sid as psid, a.sid as bsid,
b.ts as pts, a.ts as bts, b.x as px,
b.y as py, a.x as bx, a.y as by
insert into nearBallStream;

from sensorStream[(sid == '4' or sid == '8'
or sid == '10' or sid == '12')
and ((ts >= 107532955944241161
and ts <= 125572955944241161)
or (ts >= 130866391464034951
and ts <= 148796391464034951))]#window.length(10000)
select sid, ts, x, y, avg(v) as avgV
insert into ballStreamAvgs;
```


Listing 5.4: Mix of SiddhiQL queries in a single use case.

The goal of this experiment was to evaluate the performance improvements gained by executing mix of the event queries in an execution plan on GPU environment. From the set of event queries listed in Listing 5.4, the second event query (stream join query) is configured to run on GPU environment. Other two event queries run on CPUs. From previous two experiments, it is observed that event query with stream join operator can be efficiently processed in GPUs. So we have configured only the event query with stream join operator to run on GPU. All the test setup parameters are summarized in Table 5.5. For this experiment, the batch size has to be reduced to 2048 events in order to accommodate maximum number or concurrent queries in our GPU device memory.

Table 5.5: Query mix test setup parameters.

Number of input events	49,576,080
Input event time span	71 minutes and 4 seconds
Average input event rate	11626 events per second
Disruptor buffer size	8192
GPU thread block size	128 threads
GPU process event batch size	2048
GPU Selector work size	50 events
GPU Selector worker count	3
Concurrent use case count	1 to 5

5.5.1 Event Consume Speedup Analysis

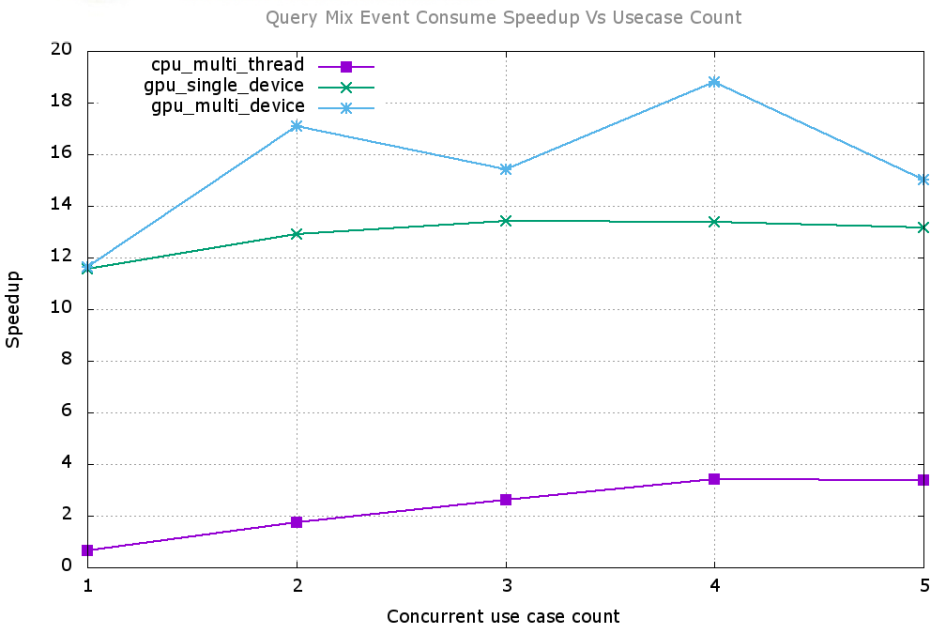


Figure 5.12: Query Mix: Average event consume rate speedup against concurrent use case count (batch size 2048 events).

The Figure 5.12 analyzes the speedup of input event consume rate of set of event queries we have configured. The event consume speedup is plotted against the number of concurrent use case count. We consider a set of event queries defined in a execution plan as a use case.

Single-threaded mode has constantly increasing event queue buildup in the main input event queue so the event publish latency was constantly increasing as shown in Figure 5.13. This is because of the sequential processing of lot of event queries in a single CPU core.

Multi-threaded mode event processing in general has more than two times higher event consume rate than the single-threaded processing. When the number of event queries running in CPU cores increasing gradually, the CPU resources are exhausted and each processing thread has to contest with a lot of other processing threads to get CPU time. With one execution plan running, single-threaded mode has higher event consume rate than multi-threaded mode processing. This was because of the additional queuing overhead incurred by the parallel processing in multi-threaded mode. Once there are enough parallel work, queuing overhead has overcame by the event processing improvements in parallel processing.

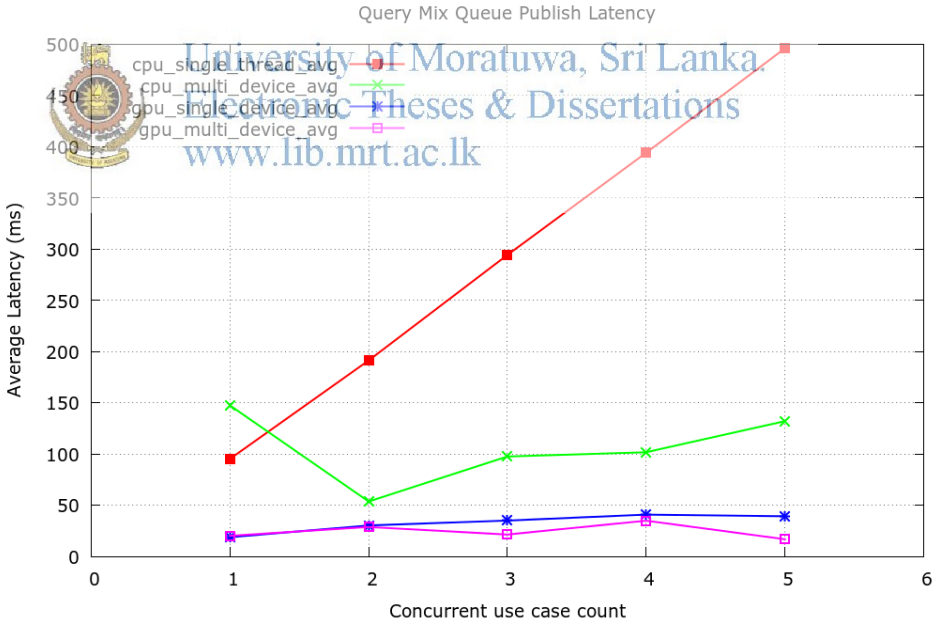


Figure 5.13: Query Mix: Input event queue publish latency against concurrent use case count (batch size 2048 events).

With GPU processing, the same use cases has achieved at least ten times event consume speedup for all the parallel use cases compared to the single-threaded mode. And this is observed for both the GPU single device processing and GPU multiple device processing. As shown by the previous two experiments, stream join operator takes the

most processing time form all the implemented event processing operators and it is the most complex and inherently parallel event processing operator. In average, event query with stream join operator (second event query) has taken about 80% of the total processing time for all the event queries in single-threaded mode. So by running event queries with stream join operator in GPU devices, the other event queries get more CPU time to do their processing. Moreover, our parallel GPU event processing algorithms drastically reduces the event processing time which ultimately helps to achieve higher event processing throughput (see Figure 5.14).

As shown in the Figure 5.13, same as the previous two experiments, input event queue size is drastically increasing in single-threaded mode which significantly increases the pressure on event producers and reduces the event processing throughput. GPU processing has significantly low queue event publish latency because of higher input event processing throughput. Moreover, the input event queue build up is almost constant with the increase of parallel use case count (number of event queries).

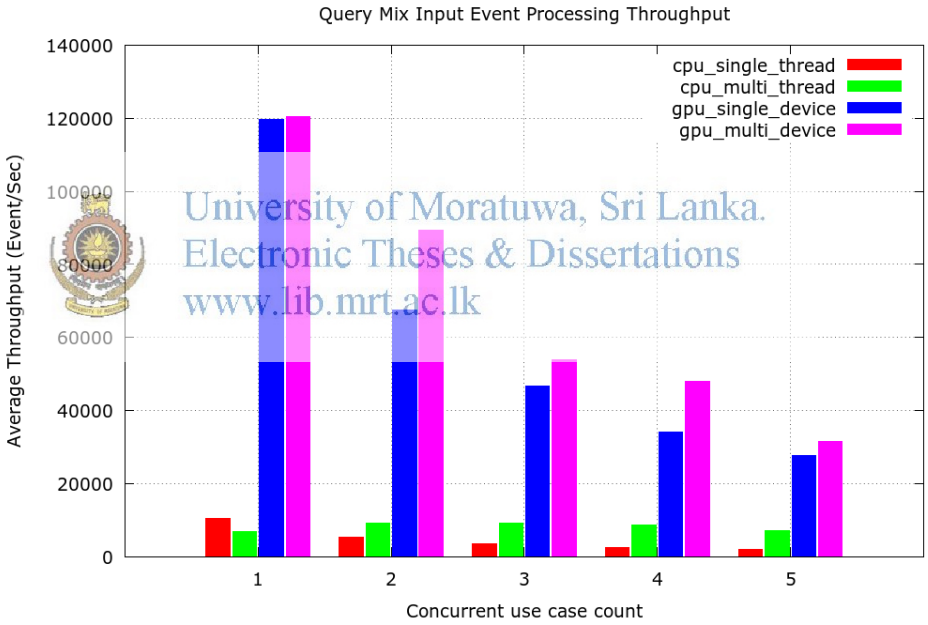


Figure 5.14: Query Mix: Input event processing throughput against concurrent use case count (batch size 2048 events).

This research try to improve performance of event processing CEP queries as general, without targeting any specific use case of complex event processing. Usually when using complex event processing for any specific use case, the queries are specially optimized for that particular use case. Sometimes custom functions or processor extensions are developed to improve performance. In these tests we were not using any optimization on the queries. Instead we were trying to analyze how query processing performance of standard query can be improved just by running it on GPU devices. It is possible

to develop custom CUDA kernels targeting particular event processing use case. For example, it is possible to develop custom CUDA kernels for each event processing use case in 2013 DEBS challenge, where each CUDA kernel is optimized for that particular scenario and data load. This way it is possible to achieve much higher event processing throughput than our approach.

Chapter 6

Conclusions

6.1 Conclusion

First, this thesis has reviewed overall architecture of the Siddhi CEP engine, along with some of the specific features and design characteristics which allow delivering high performance in event processing applications on CPU hardware technologies. Then, it presents our approach on improving Siddhi event processing throughput using GPGPU hardware technologies. We have presented the design and implementation of parallel event processing algorithms for highly used CEP event processors; filter event processor, window event processor and event stream join processor.

In designing GPGPU-based event processing mechanism, we identified the requirement of having a general purpose event processing framework for parallel hardware technologies. Separation of low-level GPU communication and changes to Siddhi CEP into two entities helped us to debug and test functionality separately. Hence, this work has two main contributions; a general purpose event processing framework implementation and new event processing runtime implementation for Siddhi CEP.

The general purpose event processing framework is implemented as a C++ library which encapsulate all the low-level implementation of GPU event processing and provides a well-defined API for use it in Siddhi CEP. Although our main concern was to use this library within Siddhi CEP, this library can be used by the other CEP engines as well. Within the scope of this research we have implemented only three event processing constructs in the library; event filter processor, event window processor and stream join processor. Other event processing constructs can be implemented in the same way extending the library.

In order to utilize GPU for event processing, we have implemented a new Query Runtime, which follows the same interface as standard Siddhi query runtimes but

process all the events for a particular query using GPU hardware. The new query runtime internally uses the event processing library we have developed. A user can decide to run a particular query using GPU at the query define time. It is possible to configure queries which run on CPUs to consume events generated by queries which run on GPU and vice versa.

Designing an event data transfer mechanism between Java and C++/CUDA-C was the main challenge we faced when implementing the event processing library. Performance study on our initial implementations showed a significant overhead in data transfer between Siddhi and GPU event processing library. So the data transfer overhead was negatively affecting the performance gain achieved by the GPU parallel processing. Our early approaches did not achieve any performance speedup due to this overhead. However, through use of direct memory buffer sharing between Java and C++, we eliminated significant fraction of data transfer overhead. Java NIO buffers were used for this purpose which enable us to access low-level memory buffers within Java in the same way it is done in C++. However, there is still a considerable performance overhead associated with the event data serialization and de-serialization.

Since this research is one of the leading edge research in this area, there is no other related research that we can compare our performance results. Moreover, as stated in the Chapter 5, there is no any standard benchmarking platform for event-based systems which is widely accepted by the research community and the industry. DEBS grand challenge workload for 2013 ACM International Conference on Distributed Event-Based Systems was chosen as the evaluation workload since it is publicly available and it represents event stream of a real-world use case. Performance results were measured with this workload for Siddhi single-threaded mode, multi-threaded mode and the proposed approach for GPU event processing. Measured results were compared against the single-thread mode to determine the speedup of event processing throughput.

Use of GPU event processing for simple queries did not achieve significant performance speedup. In fact, Siddhi single-threaded mode can outperform both Siddhi multi-threaded mode and GPU processing in simple queries. As suggested by the performance analysis, to gain a considerable event processing speedup with GPUs, there should be a significant number of concurrent complex event queries. Current Siddhi implementation, even with the multi-threaded mode, could not achieve event processing speedup when there are a lot of concurrent complex event queries. Whereas the event processing in GPUs has achieved significant performance gain with the same workload. This behavior was observed in the experiment with mix of different complex event queries where we could achieve more than ten times event processing speedup with GPU processing compared to single-threaded mode.

For complex event queries which involve many event processors, Siddhi single-threaded

mode cannot deliver significant event processing performance speedup. This behavior was observed in stream join operator test, where with Siddhi single-threaded mode, input event consume and process rate is significantly lower than the input event rate. So there was, on average, more than two times slowdown of event processing and significant event queue was built up in the input event queue.

Moreover, delegating the event processing task to GPUs brings additional advantages when considering the whole system. For example, reducing the CPU load so it can perform other necessary tasks. Since GPU devices are constrained by its memory capacity, configuring all the event queries in a Siddhi instance to run on GPUs is not a viable option. The better strategy should be to delegate event processing of most complex queries to GPUs and run other non complex queries in CPU cores. As of the current state of our research, determining the performance gain achieved by running a particular complex event query in GPUs is not possible. A pre-run with same workload should be done in order to properly identify if running a particular query in GPU can improve its event processing performance.

Performance gain of GPU event processing can vary significantly with different GPU devices. Capabilities and resources available in GPU devices can affect the performance. As we have discussed in Evaluation Chapter, some of the limitations we had in our GPU processing algorithms were due to lack of capabilities in the GPU device we have used for testing. But this should be further verified by proper testing in GPU devices with different capabilities.

The performance improvements of using GPU devices for event processing should be considered with the cost of acquiring and running GPU devices. Today, GPU devices are becoming common in even entry-level PCs which are capable of hundreds of GFLOPS performance. A mid-level GPU device like Nvidia GeForce GTX 970 can be purchased at about $300 and provides 3,494 GFLOPS. The return of investment of GPUs should be calculated for performance speedup gained for each extra dollar spend on GPU devices. Pay per usage services and GPU as a service on cloud servers* can also increase return of investment for GPUs.

In conclusion this work improves Siddhi event processing throughput when there are several complex event queries to be evaluated on high frequent incoming input event stream using a single Siddhi runtime. This work enables Siddhi users to utilize GPU devices attached to their server for event processing without doing major changes to their application. Users are able to gain parallel event processing performance advantages offered by GPGPUs just by configuring event queries to use GPU event processing at the query definition time.

*http://www.nvidia.com/object/gpu-cloud-computing-services.html

6.2 Future Work

We plan to extend our study on following directions.

6.2.1 Implementing GPU Algorithms for Other CEP Operators

This work has proposed several GPU parallel event processing algorithms for event filter operator, event window operator and stream join operator. Apart from these event operators, there are other event operators such as event sequence and pattern operator which can benefit from GPU parallel algorithms. Implementing GPU algorithms for those event operators will enable Siddhi to fully utilize GPU hardware and improve query processing performance in production systems.

6.2.2 GPU Aware CEP Architecture

The performance evaluation results of our work suggest that using GPUs for improving event processing throughput is not usable and profitable in all scenarios. It is only profitable when there is high frequent input stream which need to be processed with very complex event query. As described in Section 6.1, event serialization cost and de-serialization cost governs the profitability of our approach. This can be eliminated, if the CEP architecture is designed with utilizing parallel hardware in-mind. Particularly if event schema is represented in event representation, it is easy to serialize and de-serialize events for transfer them to GPU memory. For example, if attributes of an event is represented in separate variables of a data structure, it is easy to serialize this data structure into a memory buffer. Currently Siddhi uses array of Java Objects to represent event attributes, where serialization algorithm has to determine data type of each element at runtime in order to serialize them correctly. Having a data structure that represent an event schema helps to speedup event serialization and de-serialization, which in return increase event processing throughput in GPU environments.

6.2.3 Distributed GPU Servers

Having a GPU aware CEP architecture can be more beneficial in distributed CEP deployments, where one node is acting as load balancer and the other sub-nodes do the actual event processing. Each nodes have there own CEP instance deployed and running. In sub-nodes, it is possible to develop a light-weight CEP engine that utilize both CPU and GPU hardware, if available. This light-weight CEP engine can be implemented using low level language like C++, and it can easily utilize GPU hardware without any additional overhead.

6.2.4 Automatic Query Configure to Run on GPUs

Current implementation needs users to define whether each query should run on GPU or CPU. This requires user's pre-knowledge about the input event rate and the complexity of the query. Moreover, it may require few trial runs to decide on best configuration for execution plan to get the best event processing performance. It is more beneficial if Siddhi CEP is able to decide, in runtime, by running a particular query in GPU will deliver significant performance gain, if so configure them to run on GPU. This may require running same query in parallel in both CPUs and GPUs and compare the performance. Have a performance metrics of previously executed similar queries can also help to achieve this purpose.

6.2.5 Runtime GPU Kernel Generation

If it is possible to generate CUDA kernel runtime according to the user defined query and compile them at the runtime, more optimized GPU algorithms can be used to improve event processing for specific use cases. Compiling CUDA kernel may have initial cost, but that is negligible if it is a long running event query. Siddhi can generate CUDA kernels using the compiled execution plan. Since the generated CUDA kernel can have hard-coded logic for the particular event query processing, it is possible to get much higher performance than our current approach.

References

[1] J. Gantz and D. Reinsel, "Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East," *IDC iView*, Dec. 2012. [Online]. Available: http://idcdocserv.com/1414.

[2] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "Towards expressive publish/subscribe systems," in *Proceedings of the 10th International Conference on Advances in Database Technology*, ser. EDBT'06, Munich, Germany: Springer-Verlag, 2006, pp. 627–644.

[3] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch, "Distributed complex event processing with query rewriting," in *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems*, ACM, 2009, p. 4.

[4] F. Wang and P. Liu, "Temporal management of RFID data," in *Proceedings of the 31st international conference on Very large data bases*, VLDB Endowment, 2005, pp. 1128–1139.

[5] Y. Simmhan, V. Prasanna, S. Aman, S. Natarajan, W. Yin, and Q. Zhou, "Toward data-driven demand-response optimization in a campus microgrid," in *Proceedings of the 3rd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, ACM, 2011, pp. 41–42.

[6] D. Luckham and R. Schulte, *Event Processing Glossary – Version 2.0*. Event Processing Technical Society, Jul. 2011.

[7] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[8] A. Adi, D. Botzer, G. Nechushtai, and G. Sharon, "Complex Event Processing for Financial Services," in *IEEE Services Computing Workshops, 2006*, Sep. 2006, pp. 7–12.

[9] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, and A. Rasin, "Aurora: a data stream management system," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 666–666.

[10] M. Cammert, C. Heinz, J. Krämer, A. Markowetz, and B. Seeger, "Pipes: a multi-threaded publish-subscribe architecture for continuous queries over streaming data sources," *Department of Mathematics and Computer Science, University of Marburg*, 2003.

[11] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "STREAM: The Stanford Data Stream Management System," Stanford InfoLab, Technical Report 2004-20, 2004. [Online]. Available: http://ilpubs.stanford.edu:8090/641/.

[12] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, and E. Ryvkina, "The design of the borealis stream processing engine.," in *CIDR*, vol. 5, 2005, pp. 277–289.

[13] EsperTech, *Event Stream Intelligence: Esper and NEsper*, http://esper.codehaus.org/, [Online; accessed 10-Sept-2013], 2013.

[14] Oracle Product Management and Development Team, "Oracle Complex Event Processing: Lightweight Modular Application Event Processing in the Real World," Oracle Corporation, Technical Report, Jun. 2009. [Online]. Available: http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html.

[15] *TIBCO StreamBase*, http://www.streambase.com/, [Online; accessed 10-Sept-2013], 2013.

[16] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: towards an elastic stream computing platform for the cloud," in *Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD)*, 2011, pp. 348–355.

[17] G. Cugola and A. Margara, "Complex Event Processing with T-REX," *Journal of Systems and Software*, vol. 85, no. 8, pp. 1709–1728, Aug. 2012, ISSN: 0164-1212.

[18] M. Hirzel, "Partition and Compose: Parallel Complex Event Processing," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '12, Berlin, Germany: ACM, 2012, pp. 191–200, ISBN: 978-1-4503-1315-5.

[19] G. Sharon and O. Etzion, "Event Processing Network-A Conceptual Model," PhD thesis, Technion-Israel Institute of Technology, Faculty of Industrial and Management Engineering, 2007.

[20] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, "Siddhi: a second look at complex event processing architectures," in *Proceedings of the 2011 ACM workshop on Gateway computing environments*, ACM, 2011, pp. 43–50.

[21] WSO2 Inc., *WSO2 Complex Event Processor*, http://wso2.com/products/complex-event-processor, [Online; accessed 10-Sept-2013], 2013.

[22] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[23] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008, ISSN: 1542-7730.

[24] K. Fatahalian and M. Houston, "A closer look at GPUs," *Commun. ACM*, vol. 51, no. 10, pp. 50–57, Oct. 2008.

[25] M. R. Mendes, P. Bizarro, and P. Marques, "A performance study of event processing systems," in *Performance Evaluation and Benchmarking*, Springer, 2009, pp. 221–236.

[26] G. Cugola and A. Margara, "Low latency complex event processing on parallel hardware," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 205–218, 2012.

[27] S. Schneidert, H. Andrade, B. Gedik, K.-L. Wu, and D. S. Nikolopoulos, "Evaluation of streaming aggregation on parallel hardware architectures," in *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*, ACM, 2010, pp. 248–257.

[28] A. Margara and G. Cugola, "High performance content-based matching using GPUs," in *Proceedings of the 5th ACM international conference on Distributed event-based system*, ACM, 2011, pp. 183–194.

[29] O. Etzion and P. Niblett, *Event Processing in Action*, 1st. Greenwich, CT, USA: Manning Publications Co., 2010, ISBN: 1935182218, 9781935182214.

[30] G. Cugola and A. Margara, "TESLA: A Formally Defined Event Specification Language," in *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '10, Cambridge, United Kingdom: ACM, 2010, pp. 50–61.

[31] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," in *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '02, Madison, Wisconsin: ACM, 2002, pp. 1–16.

[32] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, ISBN: 0201727897.

[33] T. Bass, *What is Complex Event Processing?* http://www.thecepblog.com/what_is_complex_event_processing, [Online; accessed 30-Jan-2014], Apr. 2007.

[34] D. S. Rosenblum and A. L. Wolf, "A Design Framework for Internet-scale Event Observation and Notification," in *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC '97/FSE-5, Zurich, Switzerland: Springer-Verlag New York, Inc., 1997, pp. 344–360.

[35] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[36] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching Events in a Content-based Subscription System," in *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '99, Atlanta, Georgia, USA: ACM, 1999, pp. 53–61.

[37] R. R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997.

[38] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 948–960, 1972.

[39] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, and R. Cavin, "Larrabee: a many-core x86 architecture for visual computing," in *Proceedings of the ACM Transactions on Graphics (TOG)*, ACM, vol. 27, 2008, p. 18.

[40] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation—a performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[41] M. Gschwind, "The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor," *International Journal of Parallel Programming*, vol. 35, no. 3, pp. 233–262, 2007.

[42] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine Stream Processor," in *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, IEEE, 2002, pp. 282–288.

[43] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104.

[44] S. Venkatasubramanian, "The graphics card as a stream computer," in *SIGMOD-DIMACS workshop on management and processing of data streams*, vol. 101, 2003, p. 102.

[45] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[46] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch, "Rootbeer: Seamlessly using GPUs from Java," in *Proceedings of the IEEE 14th International Conference on High Performance Computing and Communication and IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, IEEE, 2012, pp. 375–380.

[47] G. Dotzler, R. Veldema, and M. Klemm, "JCudaMP: OpenMP/Java on CUDA," in *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, ACM, 2010, pp. 10–17.

[48] P. Calvert, "Parallelisation of java for graphics processors," *Part II Dissertation, Computer Science Tripos, University of Cambridge*, 2010.

[49] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA," in *Euro-Par 2009 Parallel Processing*, ser. Lecture Notes in Computer Science, H. Sips, D. Epema, and H.-X. Lin, Eds., vol. 5704, Springer Berlin Heidelberg, 2009, pp. 887–899.

[50] jcuda.org, *JCuda*, http://www.jcuda.org/, [Online; accessed 10-December-2013], 2009.

[51] Hoopoe, *jCUDA*, http://www.cass-hpc.com/category/jcuda, [Online; accessed 16-March-2014], 2009.

[52] *JaCuda*, http://sourceforge.net/projects/jacuda/, [Online; accessed 16-March-2014], 2008.

[53] *jacuzzi*, http://sourceforge.net/projects/jacuzzi, [Online; accessed 16-March-2014], 2008.

[54] J. Strnad, "Java on CUDA architecture," 2012.

[55] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of CUDA and OpenCL," *arXiv preprint arXiv:1005.2581*, 2010.

[56] Z. Wang, P. Lv, and C. Zheng, "CUDA on Hadoop: A Mixed Computing Framework for Massive Data Processing," in *Foundations and Practical Applications of Cognitive Systems and Information Processing*, Springer, 2014, pp. 253–260.

[57] D. Chen and H. Peng, "Cuda performance study on hadoop mapreduce clusters,"

[58] *OpenJDK Project Sumatra*, http://openjdk.java.net/projects/sumatra/, [Online; accessed 16-March-2014], 2012.

[59] S. Gupta, *GPU Acceleration Coming to Java, Says IBM Exec*, http://blogs.nvidia.com/blog/2013/09/22/gpu-coming-to-java/, [Online; accessed 16-March-2014], Sep. 2013.

[60] W. Zaremba, Y. Lin, and V. Grover, "Jabee: framework for object-oriented java bytecode compilation and execution on graphics processor units," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ACM, 2012, pp. 74–83.

[61] *Java bindings for OpenCL*, http://jocl.org/, [Online; accessed 16-March-2014], 2009.

[62] *OpenCL bindings for Java*, https://code.google.com/p/javacl/, [Online; accessed 16-March-2014], 2010.

[63] *APARAPI: A Parallel API*, http://developer.amd.com/tools-and-sdks/heterogeneous-computing/aparapi, [Online; accessed 16-March-2014], 2011.

[64] L. Baumgärtner, P. Graubner, M. Leinweber, R. Schwarzkopf, M. Schmidt, B. Seeger, and B. Freisleben, "Mastering Security Anomalies in Virtualized Computing Environments via Complex Event Processing," in *The 4th International Conference on Information, Process, and Knowledge Management, (eKNOW 2012)*, 2012, pp. 76–81.

[65] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs," *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 1906–1916, 2013.

[66] A. Carzaniga and A. L. Wolf, "Forwarding in a content-based network," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, ACM, 2003, pp. 163–174.

[67] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen, "Parallel Event Processing for Content-based Publish/Subscribe Systems," in *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '09, New York, NY, USA: ACM, 2009, 8:1–8:4.

[68] M. Sadoghi, H. Singh, and H.-A. Jacobsen, "Towards highly parallel event processing through reconfigurable hardware," in *Proceedings of the 7th International Workshop on Data Management on New Hardware*, ACM, 2011, pp. 27–32.

[69] H. Inoue, T. Takenaka, and M. Motomura, "20Gbps C-Based Complex Event Processing," in *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, ser. FPL '11, Washington, DC, USA: IEEE Computer Society, 2011, pp. 97–102.

[70] L. Woods, J. Teubner, and G. Alonso, "Complex Event Detection at Wire Speed with FPGAs," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 660–669, Sep. 2010.

[71] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen, "Efficient Event Processing Through Reconfigurable Hardware for Algorithmic Trading," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1525–1528, Sep. 2010.

[72] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads," *Technical paper. LMAX, May*, p. 206, 2011.

[73] JavaCPP, *Samuel Audet*, https://github.com/bytedeco/javacpp, [Online; accessed 10-January-2015], 2012.

[74] *Event Processing Technical Society*, http://en.wikipedia.org/wiki/Event_Processing_Technical_Society/, [Online; accessed 16-January-2015], 2015.