

Low Latency Complex Event Processing on Parallel Hardware

Gianpaolo Cugola^a, Alessandro Margara^a

^a*Dip. di Elettronica e Informazione
Politecnico di Milano, Italy
surname@elet.polimi.it*

Abstract

Most complex information systems are *event-driven*: each part of the system reacts to the events happening in the other parts, potentially generating new events. *Complex Event Processing (CEP)* engines in charge of interpreting, filtering, and combining *primitive events* to identify higher level *composite events* according to a set of rules, are the new breed of Message Oriented Middleware, which is being proposed today to better support event-driven interactions.

A key requirement for CEP engines is low latency processing, even in presence of complex rules and large numbers of incoming events. In this paper we investigate how parallel hardware may speed up CEP processing. In particular, we consider the most common operators offered by existing rule languages (i.e., sequences, parameters, and aggregates); we consider different algorithms to process rules built using such operators; and we discuss how they can be implemented on a multi-core CPU and on CUDA, a widespread architecture for general purpose programming on GPUs.

Our analysis shows that the use of GPUs can bring impressive speedups in presence of complex rules. On the other hand, it shows that multi-core CPUs scale better with the number of rules. Our conclusion is that an advanced CEP engine should leverage a multi-core CPU for processing the simplest rules, using the GPU as a coprocessor devoted to process the most complex ones.

Keywords: Complex Event Processing, Parallel Hardware, Multi-core CPUs, General Purpose GPU Computing

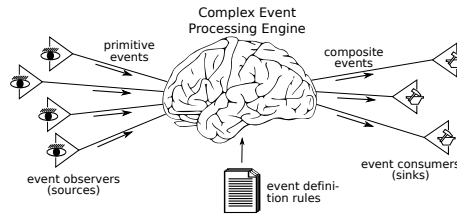


Figure 1: The high-level view of a CEP application

1. Introduction

As observed by [1] and acknowledged by most, the large majority of existing information systems are ultimately *event-driven*. Information flows are triggered by events and generate events. Each part of the system reacts to events happening in other parts and by reacting makes new events happen. This observation originated a large number of works in the area of ICT, looking for the best way to support and automate such event-driven interactions. Message brokers, Message-Oriented Middleware (MOM) [2], and Publish/Subscribe infrastructures [3] were some of the technologies developed to answer this need in the past 20 years. They succeeded in supporting exchange of messages among distributed components, but they failed in capturing the complex relationships among such messages (i.e., among the data they carry or the events they notify), which remain hidden into each component and are potentially lost.

To overcome this limitation two classes of systems were proposed recently: *Data Stream Management Systems (DSMSs)* and *Complex Event Processing (CEP)* systems. Albeit they use a different terminology to describe what they do, they act similarly: both process incoming information as it enters the system, to produce new information according to a set of rules. In principle, “pure” DSMSs differ from “pure” CEP systems as the former (rooted on the relational model) focus on *transforming* the incoming flow of information, while the latter focus on *detecting* patterns of information (i.e., events) [4]. On the other hand, things are much more blurred in reality, as currently available DSMSs (e.g., [5, 6]) enrich their relational core with primitives to detect sequences of incoming

data, while most advanced CEP include the ability to aggregate the data carried by incoming event notifications, to build their output. To avoid confusions and concentrate on a single aspect, in this paper we adopt the CEP terminology (see Figure 1) and focus on the problem of detecting pattern of *primitive events*, generating new *composite events* whose content aggregates and combines the content of the primitive events it comes from. This is done by a *CEP engine*, which interprets a set of *rules* written in an ad-hoc *rule definition language* [4, 7].

As we observed at the beginning, while a CEP infrastructure intrinsically fits those application areas that by their own nature manage events explicitly, like WSN applications for environmental monitoring [8, 9], financial applications, which analyse stock prices to detect trends [10], fraud detection tools, which observe streams of credit card transactions to prevent frauds [11], RFID-based inventory management, which analyses data in real-time to track valid paths of shipments and to capture irregularities [12], most information systems would benefit from the availability of a CEP engine. This is confirmed by the growing number of DSMSs and CEP engines that have been proposed both from the academy¹ and the industry [5, 13, 14, 15, 6, 16] in the last years. These works put in evidence how a key requirement for a CEP engine is low latency processing. Consider for example financial applications for high frequency trading: a faster processing of incoming information may produce a significant advantage over competitors. Similarly, in computerized control systems for industrial automation, the ability to react to events as quickly as possible is crucial to keep control of the automated factory.

In this paper we start from this desire of performance and investigate how a CEP engine may take advantage of parallel hardware to speed up processing. In particular, we consider multi-core CPUs and CUDA [17], a widespread architecture for general purpose GPU programming. We select the key operators

¹The reference conferences for distributed systems and data bases, like ICDCS and VLDB, publish tens of papers each year that describe advances in CEP and DSMSs, while DEBS is a conference entirely devoted to these systems.

offered by almost all existing CEP rule languages, and we describe two algorithms for interpreting them: a traditional one, taken as a reference, and one explicitly conceived for parallel processing. We detail how the former can be implemented on multi-core CPUs and how the latter can be implemented (with several differences coming from the differences in the two architectures) on both multi-core CPUs and CUDA GPUs. Finally, we compare the performance of the three resulting engines under various workloads, to understand which aspects make the choice of each architecture and algorithm more profitable. Our study shows that the use of GPUs brings impressive speedups in all scenarios involving complex rules, when the processing power of the hardware overcomes the delay introduced by the need of copying events from the main to the GPU memory and vice versa. At the same time, multi-core CPUs scale better with the number of rules deployed in the engine. Accordingly, our conclusion is that an advanced CEP engine should leverage a multi-core CPU for processing the simplest rules, using the GPU as a coprocessor devoted to process the most complex ones, thus getting the best from the two worlds.

The rest of the paper is organized as follows: Section 2 presents the operators we consider in our analysis; Section 3 offers an overview of the CUDA architecture and programming model; Section 4 and Section 5 discuss the event detection algorithms and their implementations, while Section 6 evaluates their performance. Finally, Section 7 presents related work and Section 8 provides some conclusive remarks.

2. Rule Definition Language

Over the last few years a large number of CEP engines have been developed, both by the academia and the industry. They differ significantly on several aspects, including syntax, semantics, and expressiveness of the rule definition languages, processing models, and algorithms [4]. To present results that could be easily adapted to a wide range of systems, we focus on the key event processing operators, which are supported by almost all existing engines. Moreover,

to offer a precise and unambiguous definition of these operators we introduce them using TESLA [18], the language adopted by the T-Rex CEP engine [19]. Indeed, TESLA adopts a simple syntax, easy to read and understand, while providing a formal semantics (presented in [18] using temporal logic). This allowed us to check the correctness of the algorithms we developed, while it helps the interested reader in precisely capturing the meaning of each operator.

To introduce the operators we chose, we use an example, part of an environmental monitoring application, where sensors periodically notify their position, the temperature they measure, and the presence of smoke. Now, suppose a user has to be notified in case of fire. She has to teach the system to recognize such a critical situation starting from the raw data measured by the sensors. Depending on the environment and the user preferences, the notion of fire can be defined in many different ways. Consider the following definitions:

D1 Fire occurs when there is smoke and a temperature higher than 45 degrees has been detected in the same area of smoke within 5 min. The fire notification must embed the area of detection and the measured temperature.

D2 Fire occurs when there is smoke and the average temperature in the last 5 min. in the same area of smoke is higher than 45 degrees. The fire notification must embed the area of detection and the measured temperature.

Definition *D1* can be expressed in TESLA using the following rule:

Rule R1

```
define   Fire(area: string, measuredTemp: double)
from    Smoke(area=$a) and each Temp(area=$a and value>45)
        within 5 min. from Smoke
where   area=Smoke.area and measuredTemp=Temp.value
```

TESLA assumes events to be timestamped (either at the source or when they enter the engine) and order them according to such timestamps. In particular, as exemplified above, each TESLA rule defines composite events by: (*i.*) defining their structure, i.e., their *type* and the list of *attributes* they include (in

the **define** clause); (ii.) specifying the *sequence* of primitive events whose occurrence leads to the composite ones (in the **from** clause); and (iii.) defining the actual values of the attributes of the new event (in the **where** clause). Rule R1 also shows how in TESLA the occurrence of a composite event is always bound to the occurrence of a primitive event (**Smoke** in our example), which implicitly determines the time at which the new event is detected. This anchor point is coupled with other events (**Temp** in our example) through *sequence* operators, like **each-within**, which capture the temporal relationships among events, i.e., the relevant *window* into the history of events. Also notice how in Rule R1 event selection is restricted by *parameter* **\$a**, which binds the value of attribute **area** of **Smoke** and **Temp** events.

Although very simple, Rule R1 puts in evidence how the definition of fire *D1* is ambiguous. Indeed, it is not clear what happens if the **Smoke** event is preceded by more than one **Temp** event. In such cases we say that the *selection policy* of the rule is ambiguous [4]. TESLA allows users to precisely define the selection policy they have in mind. By using the **each-within** operator, Rule R1 adopts a *multiple* selection policy: when a **Smoke** event is detected the engine notifies as many **Fire** events as the number of **Temp** events observed in the last 5 min. Other policies can be easily defined in TESLA by substituting the **each-within** operator with the **last-within** or the **first-within** operators. As an example, in presence of three temperature readings greater than 45 degrees followed by a **Smoke** event, Rule R1 would notify three **Fire** events, while Rule R2 below would notify a single **Fire** event carrying the last temperature read.

Rule R2

```
define   Fire(area: string, measuredTemp: double)
from     Smoke(area=$a) and last Temp(area=$a and value>45)
         within 5 min. from Smoke
where    area=Smoke.area and measuredTemp=Temp.value
```

Finally, definition *D2* can be translated in TESLA using the Rule R3 below. It introduces the use of *aggregates* (**Avg** in our example), i.e., functions that extract a value from a set of events. Notice how the value of an aggregate can

be used to filter out composite events that do not match certain conditions ($45 < \$t$ in our example).

Rule R3

```
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and 45 < $t=Avg(Temp(area=$a).value
          within 5 min. from Smoke)
where     area=Smoke.area and measuredTemp=$t
```

The subset of TESLA operators presented above, i.e. *sequences* with customizable *selection policies*, *parameters*, and *aggregates* are among the main building blocks of almost all existing languages adopted by CEP engines. Combined together they offer a good starting point to write a wide range of different rules. The rest of the paper focuses on them.

3. GPU Programming with CUDA

Introduced by Nvidia in Nov. 2006, the CUDA architecture offers a new programming model and instruction set for general purpose programming on GPUs. Different languages can be used to interact with a CUDA compliant device: we adopted CUDA C, a dialect of C explicitly devoted to program GPUs. The CUDA programming model is founded on five key abstractions:

i. Hierarchical organization of thread groups. The programmer is guided in partitioning a problem into coarse sub-problems to be solved *independently* in parallel by *blocks* of threads, while each sub-problem must be decomposed into finer pieces to be solved *cooperatively* in parallel by all threads within a block. This decomposition allows the algorithm to easily scale with the number of available processor cores, since each block of threads can be scheduled on any of them, in any order, concurrently or sequentially.

ii. Shared memories. CUDA threads may access data from multiple memory spaces during their execution: each thread has a *private local memory* for automatic variables; each block has a *shared memory* visible to all threads in the same block; finally, all threads have access to the same *global memory*.

iii. Barrier synchronization. Since thread blocks are required to execute independently from each other, no primitive is offered to synchronize threads of different blocks. On the other hand, threads within a single block work in cooperation, and thus need to synchronize their execution to coordinate memory access. In CUDA this is achieved exclusively through *barriers*.

iv. Separation of host and device. The CUDA programming model assumes that CUDA threads execute on a physically separate *device* (the GPU), which operates as a coprocessor of a *host* (the CPU) running a C/C++ program. The host and the device maintain their own separate memory spaces. Therefore, before starting a computation, it is necessary to explicitly allocate memory on the device and to copy there the information needed during execution. Similarly, at the end results have to be copied back to the host memory and the device memory have to be deallocated.

v. Kernels. *kernels* are special functions that define a single flow of execution for multiple threads. When calling a kernel k , the programmer specifies the number of threads per block and the number of blocks that must execute it. Inside the kernel it is possible to access two special variables provided by the CUDA runtime: the *threadId* and the *blockId*, which together allow to uniquely identify each thread among those executing the kernel. Conditional statement involving these variables can be used to differentiate the execution flows of different threads.

Architectural Issues. The CUDA model provides thread programming at a relatively low level. There are details about the hardware architecture that a programmer cannot ignore while designing an algorithm for CUDA.

The CUDA architecture is built around a scalable array of multi-threaded *Streaming Multiprocessors (SMs)*. When a CUDA program on the host CPU invokes a kernel k , the blocks executing k are enumerated and distributed to the available SMs. All threads belonging to the same block execute on the same SM, thus exploiting fast SRAM to implement the shared memory. Multiple blocks may execute concurrently on the same SM as well. As blocks terminate new

blocks are launched on freed SMs. Each SM creates, manages, schedules, and executes threads in groups of parallel threads called *warps*. Individual threads composing a warp start together but they have their own instruction pointer and local state and are therefore free to branch and execute independently. On the other hand, full efficiency is realized only when all threads in a warp agree on their execution path, since CUDA parallels them executing one common instruction at a time. If threads in the same warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

An additional issue is represented by memory accesses. If the layout of data structures allows threads with contiguous ids to access contiguous memory locations, the hardware can organize the interaction with memory into several memory-wide operations, thus maximizing throughput. This aspect significantly influenced the design of PCM's data structures, as we discuss in the next section.

Finally, to give an idea of the capabilities of a modern GPU supporting CUDA, we provide some details of the Nvidia GTX 460 card we used for our tests. It includes 7 SMs, which can handle up to 48 warps of 32 threads each (for a maximum of 1536 threads). Each block may access a maximum amount of 48KB of shared, on-chip memory within each SM. Furthermore, it includes 1GB of GDDR5 memory as global memory. This information must be carefully taken into account when programming: shared memory must be exploited as much as possible, to hide the latency of global memory accesses, but its limited size significantly impacts the design of algorithms.

4. Processing Algorithms

Most CEP engines, both from the academia [20, 19, 21] and the industry [5], adopt an incremental approach to detect composite events. They model rules as automata, which store the intermediate results derived from the computation of primitive events, until composite events are detected. In the following we de-

scribe our own implementation of this approach, which we call *Automata-based Incremental Processing (AIP)*, and use it as a baseline for our analysis. In [19], we have shown that the performance of AIP are in line (actually better) with those of Esper [5], a widely adopted commercial system which has performance as a key goal. This, together with the fact that AIP processes the TESLA language that we use in this paper as a reference, makes it an ideal baseline for our comparison.

While incremental processing through automata is becoming a standard for implementing CEP engines, it is hard to implement on massively parallel hardware like GPUs. We will come back to this point after presenting AIP in detail, what is important here is to notice that other approaches can be followed to process CEP rules as those presented in Section 2: approaches that better suit the peculiarities of GPUs. In particular, it is possible to accumulate primitive events as they arrive, delaying all the processing to the time when some specific event arrives. Indeed, CEP rules bind the occurrence time of a composite event to the arrival time of the last event in the sequence they define: rules cannot fire before this *terminator* event arrives. The *Column-based Delayed Processing (CDP)* algorithm we propose leverages this fact to reduce the complexity of processing non terminator events, resulting in an algorithm that is easier to implement and better fits the data parallel programming model of GPUs.

The rest of the section presents AIP and CDP in detail, while Section 5 describes how we implemented them on CPU and CUDA.

4.1. AIP Algorithm

AIP is a simplified version of the algorithm implemented in the T-Rex engine [19] and works as follows. First, each rule is translated into an *automaton model*, which is a linear, deterministic, finite state machine. As an example, Rule R1 of Section 2 is translated into the automaton model M of Figure 2. Each event in the sequence defined by R1 is mapped to a state in M , and a transition between two states of M is labeled with the *content and timing constraints* that an incoming event has to satisfy to trigger the transition. Additional constraints

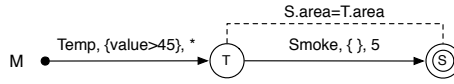


Figure 2: Automaton model for Rule R1

introduced by parameters are represented using dashed lines, as parameter a in Rule R1 that constrains the value of attribute `area` in `Smoke` and `Temp` events.

Algorithm 1 AIP Algorithm

```

1  for each model in getMatchingModels(e)
2  for each state in model.getMatchingStates(e)
3  for each aut in getAutomata(model, state)
4  if (aut.expired()) automata.remove(aut)
5  if (! aut.satParameters()) continue;
6  autDup = aut.dup()
7  autDup.nextState(event)
8  if (autDup.isInAcceptingState())
9    acceptingAutomata.insert(autDup);
10 else
11   automata.insert(autDup)
12 if (model.checkSingleSelection(state)) break;
13 generateCompositeEvents(acceptingAutomata)

```

At the beginning, each automaton model is instantiated in an *automaton instance* (or simply *automaton*), which waits in its initial state for the arrival of appropriate events. When a new event e enters the engine, AIP operates as shown by Algorithm 1. First, it extracts all the automaton models that may be affected by the arrival of e (line 1), and for each model selects all the starting states of transitions that can be triggered by e (line 2). This operation involves checking the type and content constraints that characterize each state.

Potentially, there may be several instances of an automaton model waiting in the same state. For each automaton `aut`, AIP performs the following operations.

- i.* It checks the timing constraints (line 4), eliminating it if it cannot proceed to the next state since the time limit for future transitions has already expired. Notice that an automaton in its initial state is never deleted as it cannot expire.
- ii.* It checks all parameter constraints (line 5); these constraints have to be evaluated on each instance separately since they do not depend only from the automaton model, but also from the content of the events that led the instance to its current state.
- iii.* If all constraints are satisfied, AIP creates a copy of

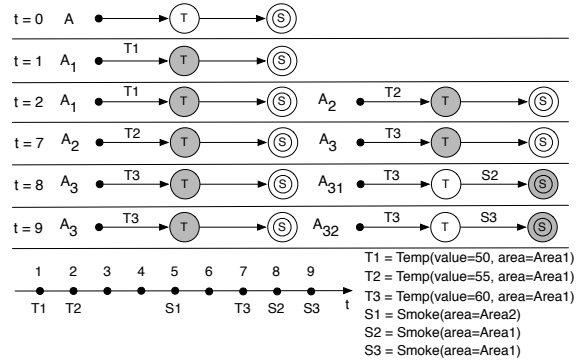


Figure 3: An example of processing using AIP

aut, **autDup** (line 6), and uses e to advance it to the next state (line 7). The original automaton **aut** remains in its state, waiting for further events. *iv*. If the new automaton **autDup** has reached the accepting state, then it is stored inside the set of accepting automata, otherwise it is added to the set of available instances (i.e., into **automata**). *v*. When a single selection policy is adopted it is sometimes possible to skip the analysis of further instances waiting in the same state as soon as one is advanced (line 12). This optimization potentially reduces the number of duplicated automata, reducing the complexity of processing. *vi*. Finally, AIP generates the composite events for all the automata that arrived to their accepting state. The content of the events stored in each automata is used to compute the attribute values of those composite events.

As an example of how the algorithm works, consider Rule R1 and the corresponding model M . Figure 3 shows, step by step, how the set of incoming events drawn at the bottom is processed. At time $t = 0$ a single automaton A of model M is present in the system, waiting in its initial state. Since A does not change with the arrival of new events, we omit it in the figure for all time instants greater than 0. At time $t = 1$ an event $T1$ of type **Temp** enters the engine. Since it matches type, content, and timing constraints for the transition to state T , A is duplicated, creating A_1 , which advances to T . Similarly, at $t = 2$, the arrival of a new event $T2$ of type **Temp** creates a new automaton A_2 from A and moves it to state T . At time $t = 5$ a **Smoke** event arrives from a

wrong area, so it is immediately discarded. At time $t = 7$, A_1 is deleted, since it has no possibility to proceed without violating the timing constraint of its outgoing transition. At the same time, the arrival of event $T3$ generates a new automaton A_3 . At time $t = 8$ A_2 is deleted, while the arrival of an event $S2$ of type **Smoke** from the correct area duplicates A_3 , generating and advancing A_{31} to its accepting state S . This means that a valid sequence, composed by events $T3$ and $S2$ has been recognized. After detection, the automaton A_{31} is deleted. Similarly, at $t = 9$, the arrival of $S3$ causes the creation of automaton A_{32} and the detection of the valid sequence composed by $T3$ and $S3$.

4.2. CDP Algorithm

While the AIP algorithm processes rules incrementally, as new events enter the engine, the CDP algorithm takes the opposite approach: it stores all events received until a terminator is found. To simplify the analysis, instead of keeping a flat history of all received events, each rule R organizes them into columns, one for each primitive event appearing in the sequence defined by R . As an example, consider the following rule:

Rule R_4

```
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
```

It defines a sequence of three primitive events of type **A**, **B**, and **C**, so CDP creates three columns (see Figure 4), each labeled with the type of the primitive events it stores and with the set of constraints on their content. The maximum time interval allowed between the events of a column and those of the previous one (i.e., the window expressed through the ***-within** operator) are modeled using a double arrow. Similarly, additional constraints coming from parameters are represented as dashed lines. Notice that the last column reserve space for a single event: the terminator.

When a new event e enters the engine, CDP operates as shown by Algorithm 2. First, it extracts all the rules, and all the columns in each rule, whose

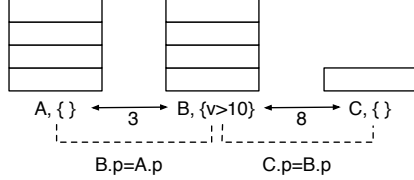


Figure 4: Columns for Rule R4

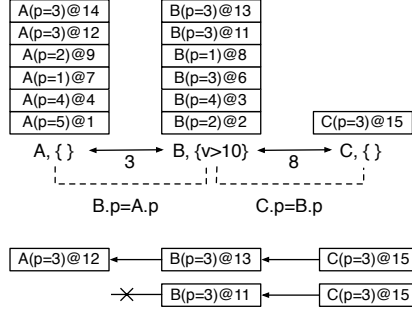


Figure 5: An example of processing using CDP

type and content constrains are satisfied by e (lines 1 and 2), and adds e on top of them (line 3). If among the matched columns there is an ending one, the processing of the events stored so far starts. Processing is performed column by column, from the last to the first one, creating *partial sequences* of increasing size at each step. First, CDP deletes old events, i.e., events that cannot participate in a valid sequence since they do not satisfy the timing constraints expressed by the rule. This is performed recursively by the `deleteOldEvents` function (line 5). Second, CDP starts computing valid sequences (line 7). This is also performed recursively, by the `computeSequences` function. At each step, CDP compares the partial sequences computed so far with the events in each column. When an event satisfies timing and parameter constraints it is used to create a new partial sequence. Finally (line 8), valid sequences returned by the `computeSequences` function are used to generate new composite events.

To better understand how the algorithm works, consider again Rule R4 and the situation in Figure 5, where the events stored in each column are represented with their type, their value for the attribute p , and their timestamp (which we assume integer, for simplicity). The event $C(p=3)@15$ was the last entering the engine. Since it is a terminator for Rule R4, it starts the processing algorithm. Its timestamp (i.e., 15) is used to compute the index of the first valid element in Column B, i.e., $B(p=1)@8$, as it results by noticing that the window between Columns B and C is 8. Previous events are removed from Column B, while the timestamp of $B(p=1)@8$ is used to remove elements from Column A whose

Algorithm 2 CDP Algorithm

```
1 for each rule in getMatchingRules(e)
2   for each column in getMatchingColumn(rule)
3     column.add(event)
4     if (column.isLast())
5       deleteOldEvents(column)
6       partialSequences.insert(e)
7       sequences = computeSequences(partialSequence, column)
8       generateCompositeEvents(sequences)
9
10 deleteOldEvents(col)
11   if (col.isFirst()) return
12   col.getPreviousCol().deleteOlderThan(col.getFirstTS - col.getWin())
13   deleteOldEvents(col.getPreviousCol())
14
15 computeSequences(partSeqs, col)
16   if (col.isFirst()) return partSeqs
17   previousColumn = column.getPreviousColumn()
18   for each p in partSeqs
19     for each ev in col.getPreviousCol().getEvents()
20       if (p.checkParameters(ev))
21         newPartSeqs.insert(createPartSeq(p, ev))
22       if (checkSingleSelection()) break
23   computeSequences(newPartSeqs, col)
```

timestamp is lower than 5. The remaining events are evaluated to detect valid sequences. First, Column B is analyzed: events B(p=3)@13 and B(p=3)@11 are both valid. Both are selected, since a multiple selection policy is defined between events B and C. This generates two partial sequences $\langle C(p=3)@15, B(p=3)@13 \rangle$ and $\langle C(p=3)@15, B(p=3)@11 \rangle$, as shown at the bottom of Figure 5. Event B(p=1)@8 is not selected, since it violates the constraint on attribute p. The two partial sequences above are used to select events from Column A. Sequence $\langle C(p=3)@15, B(p=3)@13 \rangle$ selects event A(p=3)@12, which is the only one satisfying both its timing and parameter constraints. Indeed, the event A(p=3)@14 is not valid since its timestamp is greater than the timestamp of B(p=3)@13. On the contrary, sequence $\langle C(p=3)@15, B(p=3)@11 \rangle$ does not select any event as none of those in Column A satisfies its timing and parameter constraints. At the end of the processing, the only valid sequence detected is $\langle C(p=3)@15, B(p=3)@13, A(p=3)@12 \rangle$, so a single composite event is generated.

4.3. Computing Aggregates

In describing the AIP and CDP algorithms we did not mention how aggregates are computed. Indeed, the two algorithms do not differ in this regard, as both of them calculate aggregates only when a valid sequence is detected. In principle, AIP could calculate them incrementally but we expect most application scenarios to present workloads in which a small number of valid sequences are extracted from a large amount of primitive events. Under these conditions, postponing the computation of aggregates reduces the total amount of work to do. Accordingly, our approach is the following: for each rule R , and for each aggregate a in R , a different column is created to store the events matching the type and content constraints of a . When a valid sequence for R is detected the column for a (and those for the other aggregates of R) is processed as follows.

- i.* The timestamp of the terminator is used to determine, looking at the windows in R , the events to consider according to pure timing constraints, deleting the others.
- ii.* The values and timestamps of the events participating in the sequence are used to select events in the column according to timing and parameter constraints.
- iii.* the selected events are used to compute the value of the desired aggregate (e.g., sum, average, etc.).

5. Implementation

GPUs offer a lot of computational power but the CUDA programming model only fits data parallel algorithms on pre-allocated memory areas. Unfortunately, the AIP algorithm does not fall in this category. Each automaton differs from the others and requires different processing, while new automata are continuously created and deleted at runtime. For these reasons in the following we describe how AIP can be implemented on CPUs only, while we consider both CPUs and GPUs for CDP.

5.1. Implementing the AIP Algorithm

When a new event e enters the engine, the AIP algorithm performs three operations: *i.* it selects relevant automata; *ii.* it deletes useless automata; *iii.*

it duplicates and advances automata using e . Our implementation focuses on optimizing these operations. First of all, for each automaton model m (i.e., for each rule) we group the instances of m according to their current state s . This simplifies the lookup operations in lines 1 and 2 of Algorithm 1. Automata in the same state s are stored in chronological order, according to the timestamp of the event used to trigger the last transition, and they are processed in order, starting from the most recent. When we first encounter an automaton that is too old to satisfy the timing constraints for the next transition we can safely stop and delete all the remaining automata.

Moreover, since instances of the same automaton model m share the same structure, the same type and content constraints on states, and the same timing constraints for state transitions, we store all these information once and refer to them, through pointers, in instances. This way we reduce memory occupation and minimize the effort in duplicating instances, which only store the set of pointers to the events used to arrive to their current state².

5.2. Implementing the CDP Algorithm on CPU

The CDP algorithm may be straightforwardly implemented on the CPU following the abstract description presented in Section 4. When an event e enters the engine, a single copy of e is saved, while columns store pointers to the events they include. This brings two advantages: on one hand, it reduces the amount of memory used; on the other hand, it avoids duplication of data structures, thus making it faster to add the same event to multiple columns.

As we discussed in Section 4, sequence detection is performed by analyzing columns in order, combining events of a column with the partial sequences produced at the previous step. Selecting events to form partial sequences involves evaluating timing constraints and parameters. Since columns store events in timestamp order, evaluation of timing constraints is implemented as a binary search of the first and last events satisfying the constraints. This reduces the

²For further details on how AIP is actually implemented the interested reader may see [19].

number of events to be considered for evaluating parameters. A further optimization is possible in presence of a single selection policy, by stopping processing as soon as an event matching the constraints on parameters is found.

5.3. Implementing the CDP Algorithm on CUDA

Due to the peculiarities of CUDA, implementing CDP on GPUs is not so straightforward as it was for CPUs. First of all we had to re-think the data structures used to represent columns. Indeed, memory management is a critical aspect in CUDA: the developer is invited to leverage the fact that CUDA assembles together (and computes in a single memory wide operation) concurrent memory access to contiguous areas from threads having contiguous identifiers in the same warp. Using pointers to events inside columns, as in the implementation on CPU, would lead to memory fragmentation, making it impossible to control memory accesses from contiguous threads. Accordingly, in the CUDA implementation columns do not hold pointers to events but copies of them.

Moreover, since the GPU memory has to be pre-allocated by the CPU (and allocation has a non-negligible latency), we implemented each column as a statically allocated circular buffer. We also choose to perform some operations, those that would not benefit of a parallel hardware, directly on the CPU, which keeps its own copy of columns. In particular, when an event e enters the engine and matches a state s for a rule r , it is added to the column for s in the main memory. Then e is copied asynchronously to the GPU memory: the CPU can go on without waiting for the copy to end. If e is a terminator for r , the CPU uses the information about windows to determine which events has to be considered from each column. We delegate this operation to the CPU since it requires to explore columns in strict sequence (the result of the computation on a column is needed to start the computation on the previous one), while each column can be efficiently processed using a sequential algorithm, i.e., a binary search. Once this operation has ended, the CPU invokes the GPU to process the relevant events from each column. We implemented two different kernels, optimized respectively for multiple selection and single selection policies.

Multiple selection policy. When using a multiple selection policy to process a column c , each partial sequence generated at the previous step may be combined with more than one event in c . Our algorithm works as follows:

- it allocates two arrays of sequences, called seq_{in} and seq_{out} , used to store the input and output results of each processing step. Sequences are represented as fixed-size arrays of events, one for each state defined in the rule;
- it allocates an integer $index$ and sets it to 0;
- at the first step seq_{in} contains a single sequence with only the last position occupied (by the received terminator);
- when processing a column c , a different thread t is executed for each event e in c and for each sequence seq in seq_{in} ;
- t checks if e can be combined with seq , i.e., if it matches timing and parameter constraints of seq ;
- if all constraints are satisfied, t uses a special CUDA operation to atomically read and increase the value of $index$ (`atomicInc`). The read value k identifies the first free position in the seq_{out} array: the thread adds e to seq in position c and stores the result in position k of seq_{out} ;
- when all threads have finished, the CPU copies the value of $index$ into the main memory and reads it;
- if the value of $index$ is greater than 0 it proceeds to the next column by resetting the value of $index$ to 0 and swapping the pointers of seq_{in} and seq_{out} into the GPU memory;
- the algorithm continues until $index$ becomes 0 or all the columns have been processed. In the first case no valid sequence has been detected, while in the second case all valid sequences are stored in seq_{out} and can be copied back to the CPU memory.

To better understand how this implementation works consider the example in Figure 6(a). It shows the processing of a rule R defining a sequence of 3 primitive events. Two columns have already been processed resulting in six

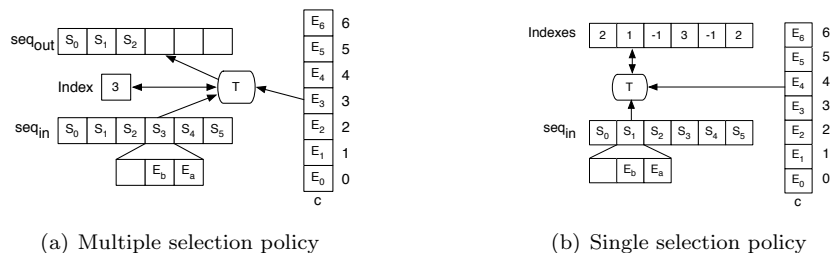


Figure 6: CDP algorithm on CUDA

partial sequences of two events each, while the last column c to be processed is shown in figure. Since there are 6 sequences stored in seq_{in} and 7 events in c , our computation requires 42 threads. Figure 6(a) shows one of them, thread T , which is in charge of processing the event E_3 and the partial sequence S_3 . Now suppose that E_3 satisfies all the constraints of Rule R, and thus can be combined with S_3 . T copies E_3 into the first position of S_3 ; then, it reads the value of $index$ (i.e., 3) and increases it. Since this last operation is atomic (`atomicInc`) T is the only thread that can read 3 from $index$, thus avoiding memory clashes when it writes a copy of S_3 into the position of index 3 in seq_{out} .

In our implementation we adopt a bi-dimensional organization of threads. All threads inside a block are organized on a single dimension and share the same element in seq_{in} . Depending from the size of seq_{in} , more than one block may work on the same element. To optimize memory accesses threads of the same block having contiguous identifiers are used to process contiguous elements of the column.

Single selection policy. When using a single selection policy to process a column c , each partial sequence generated at the previous step can be combined with at most one event in c . Accordingly, the processing algorithm is changed as follows: instead of a single $index$, we define an array of $indexes$, one for each sequence in seq_{in} . As in the case of a multiple selection policy, each thread checks whether it is possible to combine an event e from column c with one partial sequence in seq_{in} . Consider the example in Figure 6(b). Thread T is in charge of processing event E_4 and sequence S_1 . Now assume that E_4 satisfies

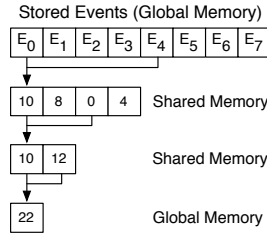


Figure 7: Computing aggregates on CUDA

all constraints for Rule R. Assume the `last-within` operator is adopted. In this case thread T calls an atomic function (`atomicMax`) that stores in the index associated to the sequence S_1 the maximum between the currently stored value (i.e., 1) and the index of E_4 in the column (i.e., 4). All positions in *indexes* are initially filled with the value -1 . When all threads have completed their processing, each position of *indexes* contains the index of the last event in column c that can be combined with the corresponding sequence, or -1 if no valid events have been found. At this point, seq_{in} is updated by adding the selected events inside partial sequences and by removing the sequences with a negative value in the *index* array. Notice that in this case we directly update seq_{in} with no need to define an additional array for output. The same algorithm also applies to the `first-within` operator: it is sufficient to modify the initial value in *indexes* and to compute the minimum index instead of the maximum.

5.4. Computing Aggregates on CPU and CUDA

To compute aggregates on the CPU we navigate through all events stored for it, selecting the relevant ones (i.e., those that match constraints on parameters, if any) and calculate the aggregate function on their value. With CUDA, the process is performed in parallel by using a different thread to combine couples of stored events. All threads in a block cooperate to produce a single value, using shared memory to store the partial results of the computation.

To understand how the algorithm works, consider Figure 7. Assume for simplicity that 8 events (E_0, \dots, E_7) have been stored for the aggregate, and that the function to compute is the sum of a given attribute att for all events

satisfying some constraints. We use a single block composed of 4 threads (t_0, \dots, t_3). Each thread t_x reads two events stored in the global memory, in position x and $x + 4$: for example, thread t_0 reads events E_0 and E_4 , as shown in Figure 7. This way, contiguous threads read contiguous positions in global memory. Each thread t_x checks whether the events it is considering satisfy the constraints defined by the rule. If this is not the case the events participate in the aggregate with a value of 0 (neutral for the sum), otherwise the value of *att* is used. Then, each thread t_x computes the sum and stores the result into the shared memory, in position x . The rest of the computation is performed in the shared memory: at each step the size of the array is halved, with each thread summing two values. This way, at each step, also the number of active threads is halved. At the last step, only one thread (t_0) is active, which stores the final results into the global memory, where the CPU retrieves it.

This algorithm must carefully consider the constraints imposed by the CUDA architecture. In particular, we have to limit the number of threads per block so that all the values stored during processing fit the limited amount of shared memory. With our reference hardware (GTX 460) we do not hit this limit until each value exceed 48 bytes. Below this value we may use the maximum number of threads per block, i.e., 512. This means that each block may analyze up to 1024 events. If the number of events stored for the aggregate is greater we use more blocks, thus producing more than one result. In this case, we apply the algorithm recursively, by considering the partial values produced at the previous step as input values for the next step, until a single value is produced.

5.5. Managing Multiple Rules with CUDA

As we said, in the current CUDA implementation, columns include copies of events. While this choice is fundamental to obtain good performance it wastes memory. The amount of memory actually required for each rule depends from the number of states it defines, from the number of aggregates it includes, from the maximum size of each column, and from the size of events (at least the part relevant for the rule). In our tests, complex rules with a relevant history of one

million events consume up to 100MB of GPU memory. This means that current GPU models, which usually include from 1 to 2GB of memory, can manage 10 to 20 of such complex rules. Notice that this represents a worst case scenario, since it is unusual for a rule to involve so many events and since we do not consider the (common) case of rules sharing events of the same type (i.e., sharing the same columns). Moreover, for simplicity our code statically allocates columns as circular buffers, which we sized big enough for the worst scenario. A better but more complex solution would be to manage GPU memory in chunks of fixed size, allocating them dynamically (from the CPU) as new events arrive and have to be copied to columns whose size limit has been reached. In any case, the CUDA implementation of CDP has to check whether the GPU memory is sufficient to store all deployed rules. If it is not, information about rules is kept in the main memory and copied to the GPU only when a terminator is detected, before starting sequence detection and aggregates computation. While this solution removes the limitation on the maximum number of rules that the engine can manage, it introduces some extra overhead due to the additional copies from the CPU to the GPU memory. In Section 6 we analyze how this impacts performance.

5.6. Use of Multi-Core CPUs

In principle, both AIP and CDP could leverage the availability of multiple CPU cores to process each rule using multiple threads (e.g., as the GPU does). On the other hand, most CEP scenarios involve multiple rules: the easiest (and most efficient) way to leverage multiple cores in this scenarios is to process each rule sequentially by paralleling the work among different rules. This is the approach we took in implementing our multi-threaded version of both AIP and CDP, in which each thread (from a thread pool of fixed size) processes a subset of the deployed rules. During our analysis, we also tried to use the same approach with CUDA, by letting different CPU threads to invoke different CUDA kernels for different rules, but we did not achieve any relevant improvement in performance. Indeed, each kernel uses a number of blocks that is large enough

Number of rules	1
Adopted rule	Rule R5
Length of sequences	3
Number of aggregates	1
Number of event types	3
Number of attribute values	50000
Size of Windows	100000

Table 1: Parameters in the Base Scenario

to fully exploit the GPU hardware resources. Different kernels launched by different CPU threads in parallel tend to execute sequentially. On the other hand, this multi-threaded solution could be used to exploit multi-GPU configurations, by processing rules concurrently on different GPUs.

6. Evaluation

Evaluating the performance of a CEP engine is not easy as it is strongly influenced by the workload: the type and number of rules and the events to process. Unfortunately, to the best of our knowledge, there are no publicly available workloads coming from real deployments. This issue is well recognized within the community, as demonstrated by the survey recently promoted by the Event Processing Technical Society [22], with the aim of understanding how event processing technologies are used.

To address this issue, we decided to use a large number of synthetic workloads, exploring the parameter space as broadly as possible. The paper presents the most notable results. Our analysis starts from a base scenario whose parameters are shown in Table 1 and examines how changing these parameters one by one (while the others remain fixed) impacts performance. To better analyse how the form and complexity of rules impacts performance, our base scenario includes a single rule:

Rule R5

```
define CE(att1: int, att2: int)
from C(att=$x) and last B(att=$x) within 100000 from C
and last A(att=$x) within 100000 from B
```


where `att1=$x` and `att2=Sum(A(att=$x).value within 100000 from B)`

After analyzing how the characteristics of such rule impact performance, we will test the impact of multiple rules. We will show that GPUs scales nicely in the complexity of rules, but they have problems when the number of rules grows.

During our tests we submit only primitive events whose types are relevant for the deployed rule (i.e., A, B, and C). We assume that (i.) primitive events are uniformly distributed among these types; (ii.) each primitive event includes three integer attributes; (iii.) the values of attributes are uniformly distributed in the interval [1, 50000]; (iv.) primitive events enter the engine one at each clock tick and they are timestamped by the engine with such clock tick.

Analysis of the workload. Rule R5 considers a sequence of three states, one for each event type, joined by a constraint on the value of attribute `att`, which requires an aggregate to be computed. This is in line with several real world cases. As an example, stock monitoring applications usually: (i) detect trends (by looking at sequences of events); (ii) filter events that refer to the same company (using parameter constraints); (iii) combine values together (using aggregates). Similarly, a monitoring application like the one we presented in Section 2 uses rules very similar to R5 to detect relevant situations.

Our base scenario includes a window size of 100000 timestamps (i.e., clock ticks). Again, this is in line with several real world cases where the detection of a relevant situation potentially demands to analyze a large number of events occurred in the past. Stock monitoring applications work on large volumes of data, including information from different companies. All this data remains relevant for relatively long amount of time (for example to detect trends) and has to be processed to detect which stocks (i.e. companies) satisfy the constraints expressed by rules. Similarly, to detect fire (see Definition *DI* in Section 2) the engine has to process events coming from the entire network (potentially a huge number of events) to find the area where smoke and high temperature occur.

Starting from the base scenario above we measured the performance of our algorithms when changing the following parameters: (i.) the number of primi-

tive events in rules (i.e., the length of sequences), (*ii.*) the size of windows, (*iii.*) the number of values allowed for each attribute, (*iv.*) the number of aggregates defined in each rule, and (*v.*) the number of rules deployed in the engine. Our base scenario uses the aggregate to define a value inside the generated complex event **CE**, but it does not use it to filter out sequences. We also explored the latter case by changing the rule and varying the percentage of sequences filtered out by the aggregate after detection. Finally, since the selection policy may significantly influence processing time, all tests have been repeated twice, once using the **last-within** operator and once using the **each-within** operator.

Experiment setup. During our tests, we initialize the system by submitting a number of events equal to the window size. At this point the engine starts discarding events that violate timing constraints and we start our measures, submitting 100000 primitive events and calculating the average time required by the different algorithms to process each of them. Since all events entering the engine are captured by the deployed rule, we are measuring the average time needed by a rule r to process an event e that is relevant for r . We are ignoring the time needed to select the set of rules interested in e since we found it to be negligible, even with a large number of deployed rules [19].

Tests were executed on a AMD Phenom II machine, with 6 cores running at 2.8GHz, and 8GB of DDR3 RAM. The GPU was a Nvidia GTX 460 with 1GB of GDDR5 RAM. We used the CUDA runtime 3.2 for 64-bit Linux platforms. Nowadays the GTX 460 is considered a low level, cheap graphic card (less than 160\$ in March 2011). Nvidia offers better graphic cards and also cards explicitly conceived for high performance computing, with more cores and memory [23].

Base scenario. Figure 8 shows the processing time measured in the base scenario. If we consider the two algorithms running on the CPU, we observe that CDP performs slightly better than AIP, independently from the selection policy. More interesting is the comparison of the CPU vs. the GPU running the same CDP algorithm. In such scenario the use of the GPU provides impressive speedups: more than 30x with a multiple selection policy and more than 25x

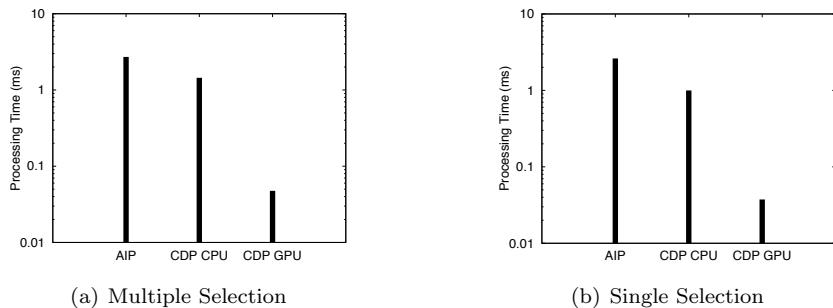


Figure 8: Base Scenario (Avg. Proc. Time for One Event)

with a single selection one. In all cases, we measure a relatively small difference between the results obtained with a multiple selection policy and with a single selection policy. Indeed, the base scenario makes use of a large number of values for each attribute, making the constraints on parameters difficult to be satisfied, and thus limiting the number of valid sequences detected even in presence of a multiple selection policy.

Length of sequences. Figure 9 shows how the performance changes with the number of states in each sequence. During this test we keep all parameters fixed, as defined in Table 1, and we only change the length of sequences, and the number of event types accordingly, so that each event entering the engine is always captured by one and only one state of the rule. Figure 9(a) shows the processing time of our algorithms taken separately, while Figure 9(b) shows the speedup of each algorithm w.r.t. AIP (our reference for comparison) under the same selection policy. Since we do not change the size of windows, increasing the length of the sequence results in lowering the number of events to process for each state. This explains why the average time to process an event decreases when the length of sequences grows. Looking at Figure 9(a) and comparing lines with the same pattern (i.e., same algorithm), we notice that, as in the base scenario, the difference when moving from the single to the multiple selection policy is limited. Figure 9(b) confirms the results of the base scenario: CDP performs better than AIP but it is the usage of the GPU which provides the biggest advantages. The figure also shows that the speedup of CDP w.r.t. AIP

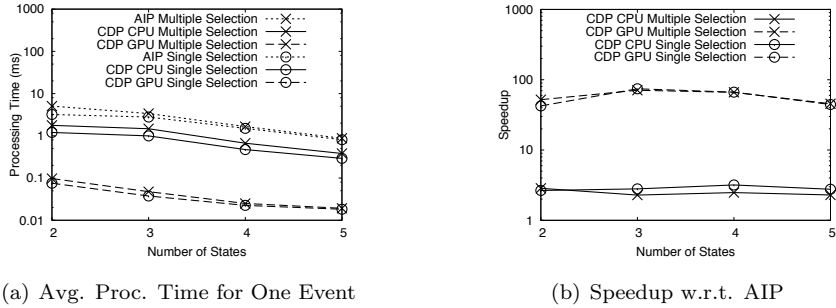


Figure 9: Length of Sequences

does not change significantly with the length of sequences.

Size of Windows. The size of windows is probably the most significant parameter when considering the use of GPUs. Indeed, this parameter affects the number of events to be considered at each state. While the CPU processes those events sequentially, the GPU uses different threads running in parallel. On the other hand, there is a fixed cost to pay in using the GPU, i.e., to transfer data from the main to the GPU memory and to activate a CUDA kernel. As a result, using the GPU is convenient only when there is a significant number of events to process at each state. Figure 10(a) summarizes this behavior: on the one hand, the cost of the algorithms running on the CPU grows with the size of windows, as expected. On the other hand, the cost of the CDP algorithm running on the GPU is initially constant at 0.017ms (it is dominated by the fixed cost associated with the use of CUDA) and it starts growing only when the number of available cores is not enough to compute events entirely in parallel. This trend is faster under a multiple selection policy, which uses more threads and produces more composite events to be transferred back to the main memory. With our base scenario, the smallest size of windows that determines an advantage in using the GPU is 4000. With a sequence of 3 states this results in considering 1333 events in each state, on average. This is an important result, since it isolates one dimension to consider when deciding the hardware architecture to adopt. If the CEP engine is used for applications whose rules need to store and process a small number of events for each state then it is better to use a CPU, otherwise

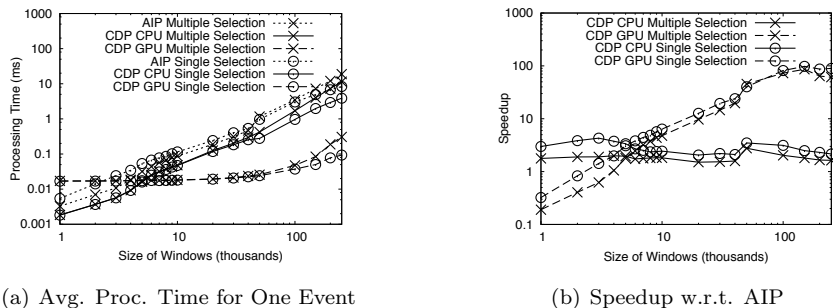
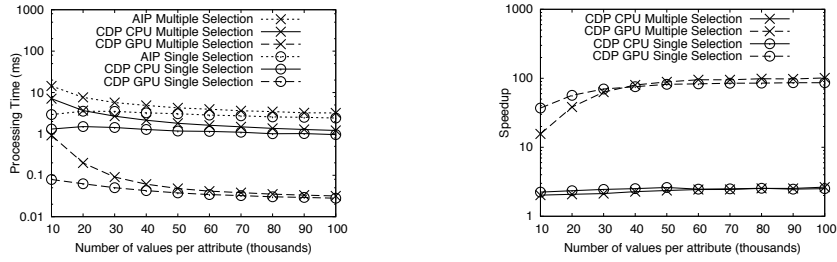


Figure 10: Size of Windows

a GPU is the best choice. Looking at Figure 10(b) we observe that the speedup of CDP running on the CPU w.r.t. AIP remains constant with the size of windows, while, as already noticed, the GPU performs better and better as the size of windows grows. With a window of 250000 events and a multiple selection policy the speedup offered by the GPU is close to 100x against AIP and close to 35x against CDP. Similar values hold for the single selection policy.

Number of values. Another factor that significantly influences the performance of our algorithms is the number of primitive events filtered out by constraints on parameters, which, in our workload, is determined by the number of values allowed for each attribute. Figure 11(a) shows that, under a multiple selection policy, a higher number of values results in lower processing times. Indeed, when the number of values grows, less primitive events satisfy the constraint on parameter $\$x$ of Rule R5, which results in detecting less composite events. The GPU implementation is the one that mostly benefits from this aspect, since it has to transfer fewer composite events back to the main memory, through the (relatively) slow PCI-e bus. On the GPU, the same behavior is registered under a single selection policy. The same is not true for AIP and CDP running on the CPU, which exhibit constant processing times under a single selection policy. Indeed, when there is no need to perform (slow) memory transfers, the advantage of reducing the number of composite events detected is balanced by the greater complexity in detecting them: with a few events matching the rule constraints, more and more events have to be processed before find-



(a) Avg. Proc. Time for One Event

(b) Speedup w.r.t. AIP

Figure 11: Number of values

ing the one that fires the single detection. The speedup graph (Figure 11(b)) confirms the considerations above. The GPU is more influenced than the CPU by a change in the number of attribute values.

Number of aggregates. Figure 12 shows how the number of aggregates that must be computed for a rule influences the processing time. During our analysis we kept fixed (to the value of 0.33, as in the base scenario) the probability for a primitive event of being relevant for each aggregate, independently from the number of aggregates defined in a rule. Figure 12(a) shows that, both under single and multiple selection policies, increasing the number of aggregates only marginally impacts performance. Indeed, in our scenario few composite events are captured at each interval, and the computation of aggregates is started only when a valid sequence is detected. The greater cost of computing 0 vs. 3 aggregates in this few cases explains why performance (marginally) degrade when the number of aggregates to calculate grows. Figure 12(b) shows that the GPU implementation is more affected by an increased number of aggregates: indeed, even if the GPU computes the aggregates faster than the CPU, copying the primitive events to the columns storing data for aggregates increases the number of memory transfers from the main to the GPU memory, which we already noticed being a bottleneck for CUDA.

Use of multi-threading. All previous tests considered a single rule, and consequently a single thread of execution on the CPU. Here we study how performance changes when multiple rules are deployed and a pool of threads

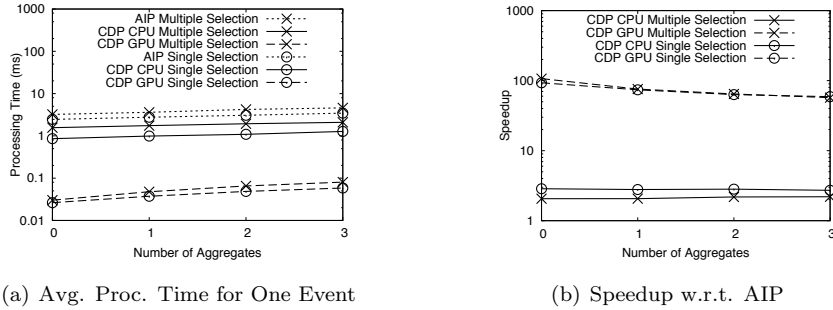


Figure 12: Number of aggregates

is used to process them in parallel. During this analysis we are interested in capturing the (common) case when a primitive event entering the engine is relevant only for a subset of the deployed rules. Accordingly, we consider rules having the same structure of Rule R5 but using different event types, in such a way that each primitive event entering the engine is captured by 1/10 of the rules. We will see how this choice impacts the performance on the GPU in the next section, here we are interested in preliminarily studying the use of multi-threading on a multi-core CPU. In particular, Figure 13 shows the speedup of the multi-threaded CPU algorithms w.r.t. the single-threaded case, when the number of deployed rules grows. For the multi-threaded case we used a thread pool whose size was experimentally determined to best match the number of rules and the number of available cores (6 in our test system). Both AIP and CDP obtain an advantage from the use of multi-threading when the number of rules increases: on our 6 cores CPU the maximum speedup we could achieve is slightly below 2.5x. Notice that with a small number of rules (below 10), the single-threaded implementation performs slightly better than the multi-threaded one, due to the overhead in synchronizing multiple threads.

Number of rules. After analyzing the influence of multi-threading we are ready to test the behavior of our algorithms (running the faster, multi-threaded version of code on the CPU), when the number of rules grows. Figure 14(a) shows that the performance of all our algorithms increases linearly when the number of rules to process grows (a linear function is a curve in a logarithmic

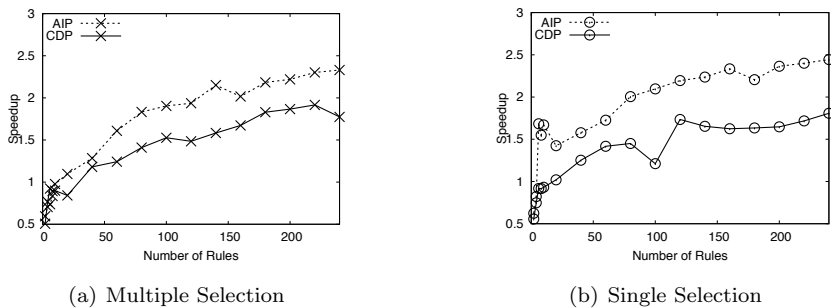
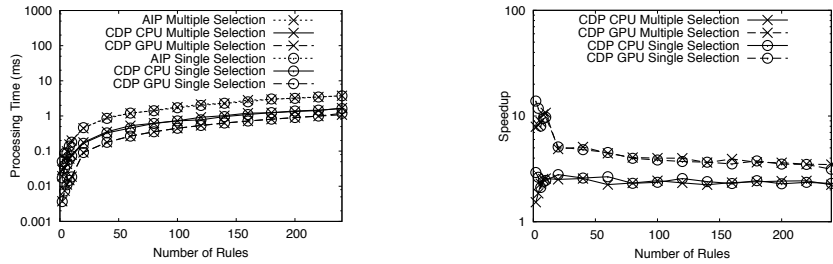


Figure 13: Use of multi-threading

graph). More interesting is Figure 14(b), which compares the CDP algorithm running on CPU and GPU w.r.t. the AIP algorithm. First of all we notice that the speedup gained by using the GPU is lower than that measured in our base scenario, even with a few rules. This behavior can be explained by remembering that we moved from a scenario where each primitive event entering the engine is relevant for the only rule available, to a scenario in which the same events are relevant for only 1/10 of the rules. With a fixed size of windows and a growing number of rules, this means that each rule captures much less primitive events than in the base scenario, i.e., less events have to be stored and processed for each rule. As we observed while analyzing the influence of the size of windows on performance, this phenomenon advantages the CPU more than the GPU. Moreover, the reduced processing complexity also reduces the differences between the single and the multiple selection policy in all algorithms, as shown both by Figure 14(a) and 14(b). Figure 14(b) highlights another aspect: the GPU speedup quickly drops when more than 10 rules are deployed into the engine. This can be explained by remembering what we said in Section 5.5: the 1 GB of RAM available on our GPU is enough to store the events relevant for at most 10 different rules. More rules force CDP to continuously move data from the main to the GPU memory, with an evident impact on performance: the speedup of the GPU significantly drops when moving from 10 to 20 rules, both under single and multiple selection policies.

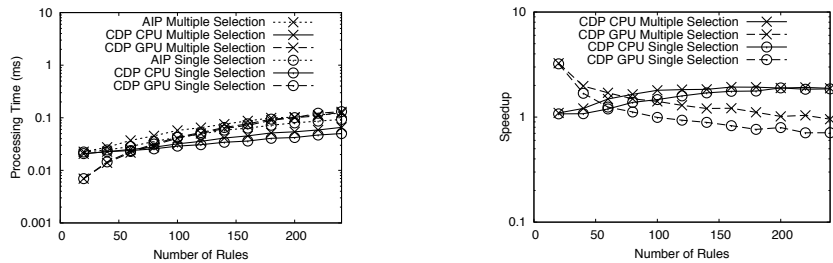
To better understand the actual limits of the CUDA architecture, we re-



(a) Avg. Proc. Time for One Event

(b) Speedup w.r.t. AIP

Figure 14: Number of rules



(a) Avg. Proc. Time for One Event

(b) Speedup w.r.t. AIP

Figure 15: Number of rules (simple rules)

peated the experiment above by further decreasing the number of rules influenced by each event and hence the number of events to consider when processing each rule. We considered a scenario where each primitive event is captured by only 1/100 of the available rules. To balance the effect of this change on the number of composite events captured, we also reduced the number of possible attribute values to 1000. We let the number of deployed rules go from 20 (twice those that may enter the GPU memory) to 250. Intuitively, this scenario is challenging for CUDA because it uses a large number of “simple rules”, i.e., rules that require few events to be processed at each terminator. Figure 15 demonstrates that this is indeed a very tough scenario for CUDA. Independently from the selection policy adopted, from 50 rules and above, the CDP algorithm runs faster on the CPU than on the GPU. This is not the first time we see the CPU outperform the GPU in our tests. The same happened for a single rule when we decreased the size of windows. Even in that case the rule became “simple”

as it involved few events. On the other hand, in that scenario there was a bound: the size of windows cannot become negative. Moreover, the (absolute) processing times were very small, so the (relative) advantage of the CPU was not relevant, in practice. This is not the case here. The number of rules may grow indefinitely, at least in theory, and the more rules we have the better the CPU performs w.r.t. the GPU, the longer are the (absolute) processing times. This means that the (relative) advantage of the CPU grows and becomes relevant also in practical, absolute terms. We may conclude that handling a large number of rules represents a real issue for CUDA.

Selectivity of aggregates. In all the tests so far, we used aggregates as values for the generated complex events, but we did not use them to filter out sequences. We analyze this case here, by changing the form of rules. In particular, we relaxed the constraints on parameters and the size of windows to detect a large number of valid sequences (we adopted a multiple selection policy), while we used an aggregate to filter out part of them. This workload is challenging for the CDP algorithm running on the GPU. Indeed, detecting valid sequences and evaluating aggregates for each of them are operations performed separately and in order by the GPU. As a consequence, increasing the number of detected sequences also increases the amount of (sequential) work to be performed. Moreover, each composite event generated must be copied from the GPU to the CPU memory, introducing additional delay. To highlight this aspect, we changed the number of sequences filtered out by the aggregate, thus changing the number of events generated at the end of the process. Figure 16 shows the results we obtained. First of all we observe how the increased complexity of the workload significantly increases the processing time of all algorithms w.r.t. the base scenario. Secondly, as we expected, this scenario is challenging for the GPU, which is still performing better than the CPU (both AIP and CDP algorithms) but with a smaller speedup w.r.t. the base scenario. On the other hand, the fact that the GPU still performs better than the CPU is a sign that the advantages it provides in reducing the processing time dominates the cost of transferring

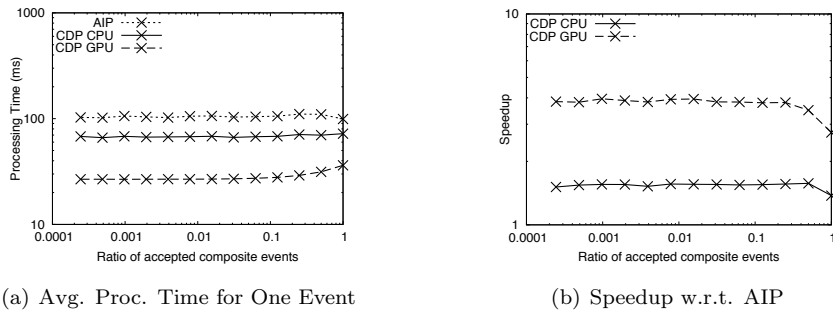


Figure 16: Ratio of accepted composite events

data around. Finally, we notice that the more composite events are accepted (i.e., pass the filter of the aggregate) the lower is the advantage in using the GPU as the cost of memory transfers increases.

7. Related Work

In the last few years a number of CEP systems [1, 7] and accompanying rule languages have been proposed in the literature. For an extensive survey on CEP systems, and data stream processing in general, the reader may refer to [4], where we analyze in great details more than 35 systems. In general, it is possible to roughly divide existing works into two main groups: on one hand there are systems explicitly designed to detect occurrences of complex event patterns, often including temporal relationships [24]; on the other hand there are systems that more generically perform on-line analysis of streaming data to extract new knowledge. In recent proposals, especially those coming from the industry [5, 6], the boundaries between the two classes are fading.

In this paper we considered a set of operations that are provided by almost all existing systems. Since the research area is relatively new, no standard or reference rule language has been proposed. For our analysis, we decided to adopt TESLA [18], which allowed us to test our algorithms using a highly expressive language. Indeed, most of the languages proposed for event processing are extremely simple, enabling only patterns derived from regular expressions, without explicit constructs to define timing constraints on sequence, programmable selec-

tion policies, or aggregates [25, 26, 27]. Moreover, to the best of our knowledge, other expressive rule specification languages proposed in the literature [28, 27] present a general structure and a set of operators that are very close to those defined in TESLA. Accordingly, the work presented in this paper can be easily applied to them.

As for the various algorithms for event processing presented in the literature, those using automata are the majority [25, 26, 27, 5]. Despite the differences in the languages they consider, they all share many similarities with the AIP algorithm adopted in this work. This aspect is explored in details in [19], where we compare AIP with a mature commercial product, showing how its performance is in line (and even better) with a state of the art. Some systems [11, 25] focus on large scale scenarios and try to reduce latency and bandwidth usage by distributing the processing load and moving the computation as near as possible to the sources of events. This is orthogonal with respect to the considerations examined in this paper, which may apply to those systems, also. In particular, we plan to complement the work presented in this paper with our own protocol for distributed processing [29].

The adoption of GPUs for general purpose programming is relatively recent and was first allowed in late 2006 when Nvidia released the first version of CUDA [17]. Since then, commodity graphics hardware has become a cost-effective parallel platform to solve many general problems, including problems coming from linear algebra [30], image processing, computer vision, signal processing [31], and graphs algorithms [32]. For an extensive survey on the application of GPU for general purpose computation the reader may refer to [33]. To the best of our knowledge, our work is the first one exploring the use of GPUs to process event notifications, detecting complex events. The Streambase [6] commercial CEP engine claims to include CUDA accelerated modules, but unfortunately the details of such modules are not publicly available. Finally, in [34] the authors adopt CUDA to speed up the operation of computing aggregates. Their work differs from ours, since they focus on periodic, rather than detection-based, computation: this assumption significantly changes the

problem and allows different kinds of optimizations. Interestingly, the authors include the Cell architecture in their analysis. Cell includes both a PowerPC core and many SIMD cores, all of them sharing the same global memory: porting our algorithm on it could increase performance, removing the cost of memory copies through the PCI express bus.

8. Conclusions

In this paper, we studied how a CEP engine may leverage parallel hardware architectures to improve performance. In particular, we considered the most common operators for CEP (i.e., sequence detection, parameter evaluation, and aggregates computation) and two CEP algorithms that take two opposite approaches: one adopts the today's standard way of processing events incrementally by using automata, while the other delays processing as much as possible. We discussed how they can be implemented on multi-core CPUs and CUDA GPUs, and compared the resulting engines using a large number of workloads, to identify the aspects that make the use of an architecture more profitable.

Apart from the specific considerations on how CEP may leverage parallel hardware, the main conclusion we come was that the large number of processing cores offered by modern GPUs makes them more suitable to process complex rules, storing and analyzing a huge number of primitive events: during our analysis we registered speedup of more than 40x for such rules. However, the programming model and hardware implementation of CUDA introduce an overhead that makes the use of GPUs inconvenient when rules are very simple. Moreover, handling a large number of rules represents a significant issue when using CUDA due to the limited memory available on the GPU. In conclusion, we believe that GPUs should be used as coprocessors, processing only a subset of the rules: the most complex ones. Simple rules can be easily handled by the CPU, making use of multi-core hardware. We plan to explore this route in our T-Rex engine, in the future.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom; and by the Italian Government under the projects PRIN D-ASAP.

References

- [1] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [2] G. Banavar, T. D. Chandra, R. E. Strom, D. C. Sturman, A case for message oriented middleware, in: *Proceedings of the 13th International Symposium on Distributed Computing*, Springer-Verlag, London, UK, 1999, pp. 1–18.
URL <http://dl.acm.org/citation.cfm?id=645956.675943>
- [3] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, *ACM Comput. Surv.* 35 (2003) 114–131.
- [4] G. Cugola, A. Margara, *Processing Flows of Information: From Data Stream to Complex Event Processing*, ACM Computing Surveys.
- [5] Esper, <http://www.espertech.com/>, visited Oct. 2011 (2011).
- [6] Streambase, <http://www.streambase.com/>, visited Oct. 2011 (2011).
- [7] O. Etzion, P. Niblett, *Event Processing in Action*, Manning Publications Co., 2010.
- [8] K. Broda, K. Clark, R. Miller, A. Russo, SAGE: A Logical Agent-Based Environment Monitoring and Control System, in: *Proceedings of the European Conference on Ambient Intelligence, AmI '09*, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 112–117.

- [9] Event Zero, <http://www.eventzero.com/solutions/environment.aspx>, visited Jan. 2011 (2011).
- [10] A. Demers, J. Gehrke, M. Hong, M. Riedewald, W. White, Towards Expressive Publish/Subscribe Systems, in: Y. Ioannidis, M. Scholl, J. Schmidt, F. Matthes, M. Hatzopoulos, K. Boehm, A. Kemper, T. Grust, C. Boehm (Eds.), *Advances in Database Technology - EDBT 2006*, Vol. 3896 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2006, pp. 627–644, 10.1007/11687238_38.
- [11] N. P. Schultz-Møller, M. Migliavacca, P. Pietzuch, Distributed complex event processing with query rewriting, in: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS '09*, ACM, New York, NY, USA, 2009, pp. 4:1–4:12.
- [12] F. Wang, P. Liu, Temporal management of RFID data, in: *Proceedings of the 31st international conference on Very large data bases, VLDB '05*, VLDB Endowment, 2005, pp. 1128–1139.
- [13] IBM, Business event processing white paper, websphere software (2008).
- [14] Oracle, <http://www.oracle.com/technologies/soa/complex-event-processing.html>, visited Oct. 2011 (2011).
- [15] Tibco, <http://www.tibco.com/software/complex-event-processing/businessesvents/default.jsp>, visited Oct. 2011 (2010).
- [16] M. Ali, An introduction to microsoft sql server streaminsight, in: *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application, COM.Geo '10*, ACM, New York, NY, USA, 2010, pp. 66:1–66:1.
- [17] CUDA, http://www.nvidia.com/object/cuda_home_new.html, visited Jan. 2011 (2011).

- [18] G. Cugola, A. Margara, TESLA: a formally defined event specification language, in: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS '10, ACM, New York, NY, USA, 2010, pp. 50–61.
- [19] G. Cugola, A. Margara, Complex Event Processing with T-Rex, Tech. rep., Politecnico di Milano (2010).
- [20] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, W. White, Cayuga: a high-performance event processing engine, in: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, SIGMOD '07, ACM, New York, NY, USA, 2007, pp. 1100–1102.
- [21] A. Adi, O. Etzion, Amit - the situation manager, The VLDB Journal 13 (2004) 177–203.
- [22] E. P. T. Society, <http://www.ep-ts.com/>, visited Jan. 2011 (2011).
- [23] Nvidia Tesla: High Performance Computing, http://www.nvidia.com/object/tesla_computing_solutions.html, visited Jan. 2011 (2011).
- [24] G. Mühl, L. Fiege, P. Pietzuch, Distributed Event-Based Systems, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
- [25] G. Li, H.-A. Jacobsen, Composite Subscriptions in Content-Based Publish/Subscribe Systems, in: Middleware, Springer-Verlag New York, Inc., 2005, pp. 249–269.
- [26] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, W. White, Cayuga: a high-performance event processing engine, in: SIGMOD, ACM, New York, NY, USA, 2007, pp. 1100–1102.
- [27] J. Agrawal, Y. Diao, D. Gyllstrom, N. Immerman, Efficient pattern matching over event streams, in: SIGMOD, ACM, New York, NY, USA, 2008, pp. 147–160.

- [28] D. Gyllstrom, J. Agrawal, Y. Diao, N. Immerman, On Supporting Kleene Closure over Event Streams, in: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 2008, pp. 1391–1393.
- [29] G. Cugola, A. Margara, RACED: an adaptive middleware for complex event detection, in: Proceedings of the 8th International Workshop on Adaptive and Reflective Middleware, ARM '09, ACM, New York, NY, USA, 2009, pp. 5:1–5:6.
- [30] J. Krüger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, in: ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, ACM, New York, NY, USA, 2005, pp. 234–242.
- [31] J. D. Owens, S. Sengupta, D. Horn, Assessment of Graphic Processing Units (GPUs) for Department of Defense (DoD) Digital Signal Processing (DSP) Applications, Tech. rep., Department of Electrical and Computer Engineering, University of California, (2005).
- [32] P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: Proceedings of the 14th international conference on High performance computing, HiPC'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 197–208.
- [33] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, T. Purcell, A Survey of General-Purpose Computations on Graphics Hardware, Computer Graphics Volume 26.
- [34] S. Schneidert, H. Andrade, B. Gedik, K.-L. Wu, D. S. Nikolopoulos, Evaluation of streaming aggregation on parallel hardware architectures, in: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS '10, ACM, New York, NY, USA, 2010, pp. 248–257.