Preface

In the following thesis I consider the realm of Complex Events Processing, the answer to the necessity of extracting new information from continuous streams of events data flowing from peripherals observers to a central system, extracting patterns of events temporally bound. More in details I will focus on TRex, a CEP middleware based on the TESLA formal language, developed by Cugola and Margara [7] [5].

The core component of this middleware is the CEP engine; the kind of information processed and algorithm employed by the engine are suitable to offer a great level of parallelism. As a result in this thesis I will describe the development of a new engine for TRex that uses NVIDIA CUDA [4] to exploit the processing power of modern GPUs to improve CEP performance. My work extends a test project [6] developed by Cugola and Margara that already focused on this goal.

The resulting engine, called GTRex, proved to be very efficient in many scenarios and with the same API and feature set of the default TRex engine, CPU based.

This allowed me to modify the TRex middleware to integrate both engines at the same time. The two can now be used interchangeably applying each of them to the rules it executes more efficiently.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Complex Event Processing is an important topic in information technology: it analyzes streams of information coming from multiple sources to extract new, more complex, knowledge. Many CEP middlewares, taking different approaches, have been developed as a solution to the CEP problem; one of these is TRex2, developed by Cugola and Margara [5].

At the same time modern graphics cards with powerful Graphics Processing Units (commonly abbreviated GPU), are nowadays installed in every computer sold, and using their capabilities for general purpose computation is quickly becoming a common technique. NVIDIA is the major player in this field, thanks to it proprietary technology called CUDA, that greatly simplifies the usage of their GPUs to make generic computations not related to the visualization of an image on the computer display.

In this thesis indeed CUDA will be exploited to boost the performance of the TRex2 core engine; CHAPTER 1 starts with an exemplified introduction to the CEP problem and continues with the description of the current version of the TRex2 middlware and, with more details, of its core engine.

CHAPTER 2 instead gives an overview of the typical architecture of a modern GPU, explains the principles of GPU programming and shows the programming model and some of the features of CUDA.

CHAPTER 3 is where the actual work done in this project is described: it contains the core algorithms of the GPU powered version of the engine as well as many descriptions of the solutions adopted to make everything work.

CHAPTER 4 analyzes with many benchmarks trying different workloads the performance gains obtained over the standard CPU version of TRex2. Each test is presented with a pair of graphs and an interpretation of the result.

Chapter 1

Complex Event Processing

1.1 Introduction

Sometimes information systems have to deal with big quantities of flowing data that enter the system continuously; as example fraud detection tools, financial applications and even airlines must promptly take actions based on the information they gather in real time. Being able to timely process this data to extract new knowledge is not trivial, and is the main goal of Complex Event Processing.

Recently, different approaches to the problem have been studied: some rely on the solid base of DBMS (DataBase Management Systems), introducing some modifications to better adapt those systems to the world of live streams of data, others are built from the beginning around the idea of event notifications, observers and sinks.

As example, Data Stream Management Systems allow data to be processed in streams, while it flows through the system, with continuous query execution. They operate on windows of events, where a window is a finite collection of events temporally bound. These systems operate mainly on pure relational tables, where they index the incoming data before the computation begins. This kind of computation is indeed unnatural and not enough flexible to support the constructs needed by a complex event processing language, that selects complex patterns of events bound by temporal relations.

On the other hand there are systems built around a native concept of

event notification. In these systems (like the one shown in FIGURE 1.1) many components collaborate exchanging information about events occurring: some of these components publish notifications about events happened (they are the observers), while others (the consumers) subscribe to notifications of events they are interested in. The whole system relies on a message based communication (the boxes in the picture). The dispatcher is in charge of routing the events to the subscribers. In this scenario a CEP engine (the brain in the centre of the picture) generates complex events based on combinations of events notifications as described in specific rules, installed in the engine.

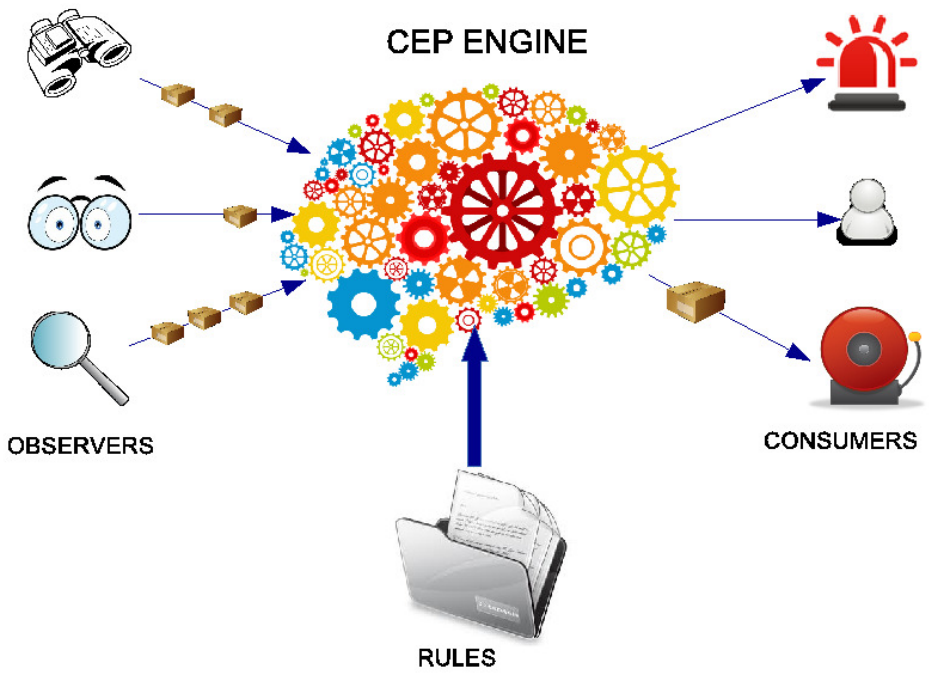


Figure 1.1: Overview of a CEP application.

TRex2, the CEP middleware presented and enhanced in this thesis, is based on the latter kind of systems. One basic example, that will guide us throughout the thesis, is the following possible scenario in which a CEP middleware acts: in an automation system there are N tanks that contain some kind of liquid. Each tank has its own valve that can be opened and closed remotely. In order to make the system safe, an alarm state must be notified in the following cases:

- a specific tank is not empty within 10 minutes after a valve open command has been issued for the same tank
- a specific tank is empty and not any valve open command has been issued in a 10 minutes interval before the empty tank notification
- the maximum level of liquid in a specific tank is greater than a specified treshold within 1 minute from a valve open command for the same tank
- the level of liquid for a specific tank is increasing in the last 10 readings and a valve open command has been issued within 1 minute

In this example each of the four data extraction paths is specified in a different rule; the sensors for the level of the liquid installed in the tanks are the observers, while the alarm systems are the sinks.

As the will see below TRex2 has observers, an engine and a dispatcher; rules for the TRex2 engine are written in the TESLA specification language, that will be introduced in the next section.

1.2 TESLA

TESLA is a complex event specification language; it has been proposed by Cugola and Margara in 2010 [7]. It combines a simple and clean syntax with an high expressiveness and flexibility. It provides content and temporal constraints, parameterization, negations, sequencies, aggregates, timers and fully customizable policies for events selection and consumption. With these features it fits the requirements needed to express all the four formulations of the example problem. It considers incoming data items as notifications of events and defines rules declaring how they combine to build new complex events. *Events* happen instantaneously at some point in time and are observed by sources, which encode them in event notifications. Each event has a type, a timestamp and a set of attributes, identified by their name and information kind.

In the automation system example a reading for the level of liquid in a tank is an event with the attributes Value, stating the level of the liquid, and TankID, representing the tank for which the level has been read. Also

the valve open commands issued can be thought at as events, with a TankID attribute representing the target tank.

As example the event stating that at time 10 a liquid level of 5 for tank number 1 has been read is formatted as:

$$\text{Level@10}\{\text{TankID} = 1, \text{Value} = 5\}$$

while an event stating that at time 3 a valve open command for tank 1 has been issued can be formatted as:

$$\text{Open@3}\{\text{TankID} = 1\}$$

Parameters express relations between attributes of different events: the fact that the the tank for which the level of liquid is read and the valve open command has been issued must be the same, as example, is expressed through a parameter on the TankID attribute of the *Open* and *Level* events, that must be equal.

Negations instead state what should not happen at some time in the events sequence: as example the second formulation of the example above specifies that there must not be any Open event in a 10 minutes range before the Level event.

Aggregates allow the computation of a single value as a function of the values of attributes of the events respecting the aggregate constraints; as example the maximum of levels read in a time interval can be computed as a max function on the values read.

Selection Policy is an important aspect in the rule definition, as it affects deeply how the simple events are used and combined to form complex events; let's see what it does with the help of the following general structure of a TESLA rule:

Rule Definition

define	$CE(Att_1 : Type_1, ..., Att_n : Type_n)$
from	$Pattern$
where	$Att_1 = f_1, ..., Attn = f_n$
consuming	$e_1, ..., e_n$

Define describes the complex event created by the rule, with its name and its attributes.

The *pattern* clause is probably the most important, because it describes the pattern of events that form the complex event. The temporal relation be-

tween them, the static constraints on their attributes as well as the selection policy are defined here.

The first event defined in the pattern clause is called *terminator*; it is indeed the event that closes a sequence of events that can create a new complex events and whose timestamp is used as timestamp for the event to be created. As we will see later, it is very important especially in the CDP [sec 1.4.2] algorithm because it is the event that triggers the key compuation for the creation of new complex events.

Each event in the clause following the terminator is preceeded by the *selection operator*: the selection operator can be one out of each-within, last-within or first-within. It allows to define precisely and without ambiguity which simple events must be selected for the creation of the complex event.

Finally all the events of the pattern clause are translated in *states*, that will become important when talking about the implementations of the CEP engines.

The *where* clause instead specifies parameters binding different events, while the *consuming* one specifies which events will be consumed, thus deleted, once they have participated to the creation of a complex event.

The first formulation of the automation example above can thus be expressed with the following TESLA rule:

Rule R1

define	$Alarm(TankID : int)$
from	$Level(TankID = \$x, Value < 5)$
	and last $Open(TankID = \$x)$ within 10 from Level
where	$TankID = \$x$

Note that the selection policy for the Open event is of the last-within kind. This means that if there are more than 1 Open events occurred in the last 10 minutes when a terminator arrives, only the last valid one will be selected, and only one complex event will thus be created. With an each-within selection instead each Open event would have caused the creation of 1 complex event.

1.3 TRex Middlware

TRex is a complete middleware for complex event processing. It is based on a simple publish-subscribe architecture [5] It has a server that initializes the computation engine, deploys rules and listens for new subscriptions and events notifications. These are provided by java clients, that use specific TRex libraries. Clients send subscription packets (SubPkt) and event notifications (PubPkt) to the server, that accordingly, and using the rules deployed, creates and publishes complex events (still represented as PubPkt) to the subscribed clients, as shown in FIGURE 1.2. Still the most critical component in this system is probably the engine: it is the most impacting component on overall performance of the middleware, when it is not limited by other factors like poor network connections.

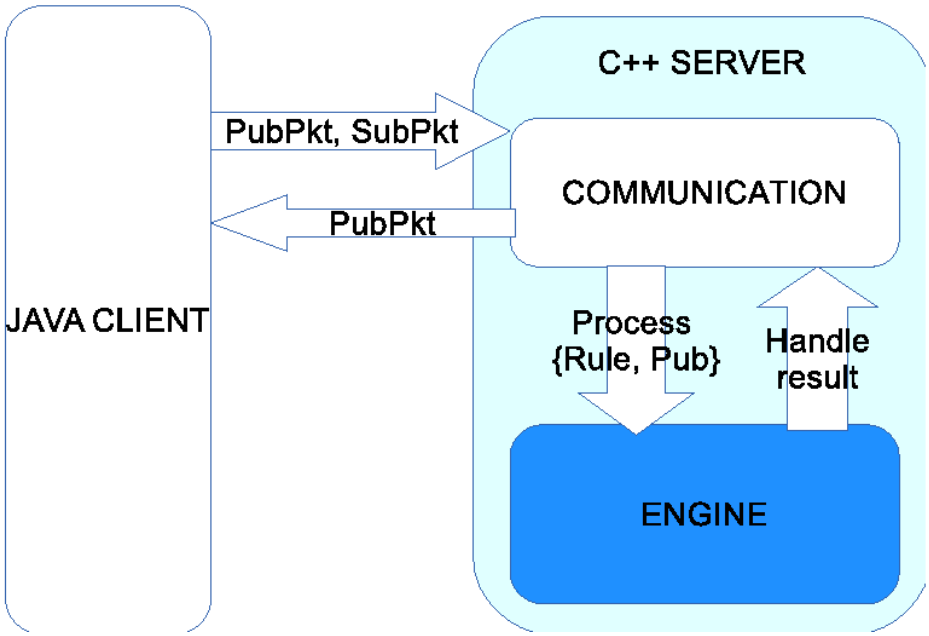


Figure 1.2: Overview of the middleware.

1.4 TRex Engine

Three versions of the engine have been created before my project took off: two of them run on the host CPU and employ different algorithms, AIP and CDP, while a third one, still based on the CDP algorithm, used CUDA

to speed up the computation, but was a very limited test project with only a small subset of functinalities. Currently the AIP version has been dropped, because it demonstrated to be less efficient than the CDP one; still I will give a basic introduction to the AIP, as it represents the typical approach to rule processing adopted by commonly available CEP engines.

1.4.1 Automata Incremental Processing

The AIP approach is probably the most natural one when creating an algorithm to process complex events; it is also used by Esper [1], that is the most used open source CEP system. It models rules as linear, deterministic and finite automata, which evolve with the events entering the engine.

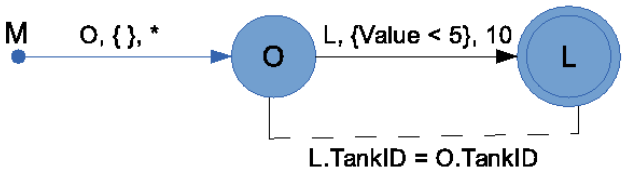


Figure 1.3: Example of the AIP model of rule R1.

FIGURE 1.3 is an example of such rule model for rule R1. To keep track of the history of events arrived, instances of the automata are created; each instance that is not in a final state represents a partial sequence.

A partial sequence is a collection of events that partially satisfies a specific rule installed in the engine. Still it can't trigger the creation of a new complex event since there are still missing packets to complete the sequence required by the rule.

At the beginning all the automata have a single state, the inital one (M in FIGURE 1.3). When a new event enters the engine the computation starts checking all the available automata and deleting the ones that clearly can't evolve any further because their timing constraints can't be compatible with any event occurring from that moment on. Then the kind and the parameters of the new event are checked against the transition constraints of the current state of the automata. These constraints depend on the rule that the automata is following and the parameters of the events that brought to that specific automata up to that moment. As example, in FIGURE 1.3, a transition from the state M is triggered when an event of kind O is received

at any time and with any attribute value. The transition from state O to state L is allowed only when an event Level arrives with a V attribute lower than 5 and with a TankID equal to that of the O event. If everything is compatible, a new automata is created from the previous one, that is not deleted because it may still evolve differently with future events. When an automata reaches its ending state, the corresponding complex event is generated and the automata is deleted.

Table 1.1: Example of packets arriving

Time	Event
1	O{TankID = 3}
2	O{TankID = 4}
4	O{TankID = 1}
5	O{TankID = 5}
7	O{TankID = 3}
12	L{TankID = 3, V=1}

Consider as example the events from table 1.1, showing a possible occurrence of incoming events; they produce the computation showed in FIGURE 1.4. At time 1 the event O{TankID = 3} arrives and a new instance of an automata is created. The same happens at time 2, when there will be 2 automata instances waiting for a possible terminator event L. At time 12, when a terminator arrives, the instances created at time 1 and 2 previously will be deleted, since they will be too old to trigger any new complex event. On the other hand there will be an automata, created at 7, that will evolve in its final state. That automata will create a new complex event.

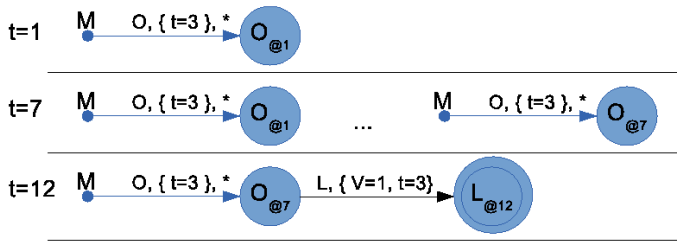


Figure 1.4: Example of evolution with the events from table 1.1.

If a multiple selection policy is adopted, then all the automata must be checked for every new event. Instead, with a single selection policy, the computation triggered by the arrival of the last terminator event can be

stopped as soon as the first valid automata evolution is found, as long as they are checked in the order requested by the selection policy. The algorithm is quite simple, but, even if implemented efficiently, can consume quite a lot of memory, and is not as efficient as the CDP algorithm.

1.4.2 Column-based Delayed Processing

The CDP approach is the opposite when compared to AIP: when a new event is notified the engine does nothing but storing it for a delayed computation. The representation of events and partial sequences is, indeed, completely different from the AIP. The basic data structure employed is a column: one column stores all the received events that are relevant for a specific state of an installed rule, as shown in FIGURE 1.5; the columns are bound with constraints on the timestamps and on the attributes of their events. Moreover the dimension of the columns reserved for the terminator events is only 1: this is because at any time only 1 terminator at maximum will be stored in the engine.

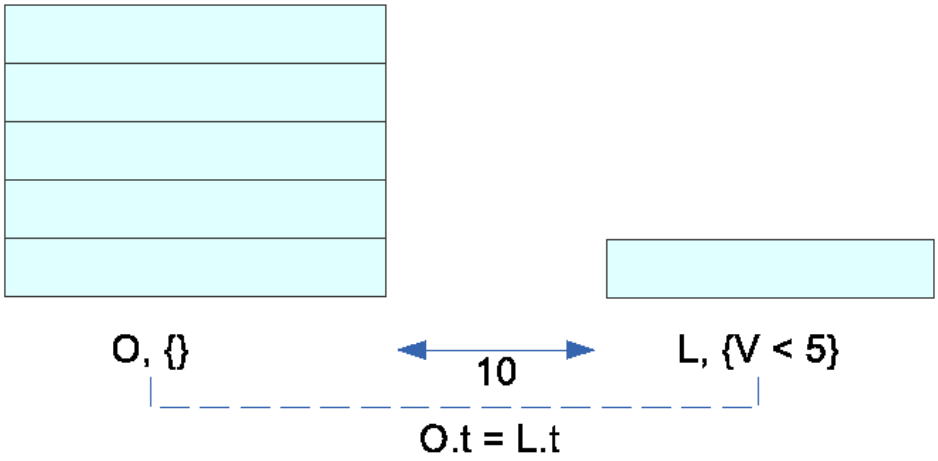


Figure 1.5: CDP data structures.

Only one physical copy of the event is stored, so that the events in the columns are in reality only pointers to the unique original event. As said before the computation is delayed, and is launched only when a terminator event is collected, because a terminator is the necessary condition for a complex event to be created. So when a new event arrives it is checked against all the rules installed and added to the columns of the corresponding

states declared in the rule, if any. In this phase also the static constraints on their attributes are checked. If one of these states is a terminator one, the computation for the related rule is launched, as shown in algorithm 1.4.1.

- line 5: old events in the columns that can't combine with any newly generated event are deleted
- line 7: all the columns are checked in turn and partial sequences are created; a partial sequence is the corresponding element of the automata of the AIP approach: it is a sequence of primitive events that up to the state currently being checked form a valid sequence for a complex event. Again, if a full valid sequence is found, a corresponding complex event is generated. Events in the columns are stored in order from the oldest to the newest, so that the computation for a specific state, when a single selection policy is applied, can be stopped as soon as the first valid element is found and added to the partial sequence. With a multiple selection policy, instead, all the elements of a column must be checked in any case.
- line 8: if there are valid sequences the corresponding complex events are created

FIGURE 1.6 shows an example of computation for rule R1. There are two columns, one for the event type O and one for the terminator event L, with a size of 1.

At time 12 the terminator arrives and the computation starts: the O column is analyzed in order, from the last or the first event depending on wether the selection policy is first, last or each within. The event with timestamp 1 is immediately discarded and deleted, since it won't be compatible with any future terminator. The rest of events are all analyzed, but only O(t=3)@7 meets all the requirements and is used to form a valid sequence. From this valid sequence the complex event A(t=3) is created, with the same timestamp of the terminator in its sequence.

Computing Aggregates

The computation of aggregates specified in a rule is performed at the end of the work cycle related to the arrival of an event, since it is only needed

Algorithm 1.4.1: CDP Algorithm on CPU

```
1  foreach rule in getMatchingRules(e) do
2  |  foreach column in getMatchingColumn(rule) do
3  |  |  column.add(e)
4  |  |  if column.isLast() then
5  |  |  |  deleteOldEvents(column)
6  |  |  |  partialSequences.insert(e)
7  |  |  |  sequences = computeSequences(partialSequence.column)
8  |  generateCompositeEvents(sequences)

9  deleteOldEvents(col)
10    if col.isFirst() then
11 |  return
12
   col.getPreviousCol().deleteOlderThan(col.getFirstTS()-col.getWin())
13    deleteOldEvents(col.getPreviousCol())

14 computeSequences(partSeqs, col)
15    if col.isFirst() then
16 |  return partSeqs
17    previousColumn c̄olumn.getPreviousColumn()
18    foreach p in partSeqs do
19 |  foreach ev in col.getPreviousCol().getEvents() do
20 |  |  if p.checkParameters(ev) then
21 |  |  |  newPartSeqs.insert(createPartSeq(p,ev))
22 |  |  if checkSingleSelection() then
23 |  |  |  break
24    computeSequences(newPartSeqs, col)
```

when new complex events are found. It is performed sequentially on the events requested by the rule definitions. The events relevant for a specific aggregates have their own columns assigned, just like the events used for the computation of the states transitions. The algorithm loops through them and, if the parameter verification succeeds, their attribute are processed as needed to complete the computation.

Computing Negations

Negations, in the CDP algorithm, are checked within the computation routine. Indeed when a new candidate primitive event for a partial sequence

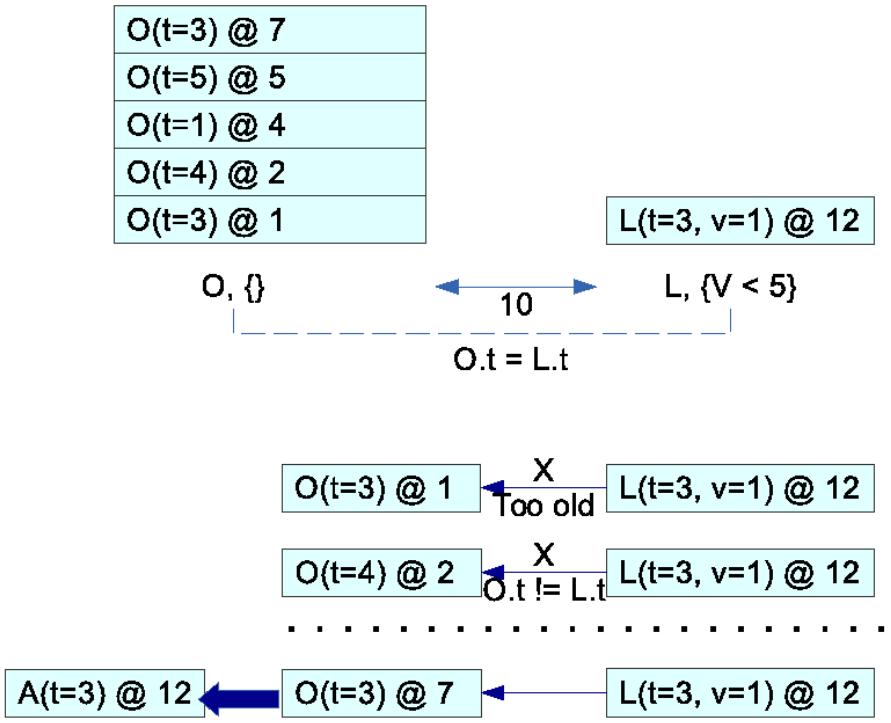


Figure 1.6: Example of CDP processing.

is found, negation constraints are sequentially checked, looping over the negations column, and eventually invalidate the new possible partial sequence, that is deleted. Again, just like aggregates, events relevant for a negation are stored in a dedicated column.

Exploiting multi-core CPUs

Exploiting modern CPUs having multiple computational cores efficiently is quite easy, at least in theory, in a CEP engine. Indeed a very simple form of parallelism is exposed by the fact that usually an engine must handle multiple rules; so both AIP and CDP can easily distribute the compuation designating different CPU threads to different rules. Each thread can then perform its jobs without affecting or being affected by the others, thus exploiting all the physical cores available, as long as there are enough rules to occupy them.

Chapter 2

General Purpose computation on GPU

2.1 Introduction

In this chapter I will introduce Graphics Processing Units and describe in details the General Purpose computing on GPU programming model, with its advantages and disadvantages. I will focus on CUDA, since it has reached a great success and maturity, and is the recommended choice for general purpose GPU programming.

2.2 Graphics Processing Units

A GPU (Graphics Processing Unit) is a specialized processor designed to efficiently make the computations necessary to create an image to be displayed. It is the core of a graphics card, that also incorporates dedicated memory chips and the circuitry to allow communications between the GPU and the rest of the computer system, like the host CPU and DMA controllers.

From the very first models, developed in the first '80s, graphics cards were designed with very specific functionalities; they had custom microchips implementing a very limited set of functionalities, for which they were great. On the other hand they couldn't do anything but what they were designed for. They were mainly used to draw simple geometric shapes like lines and circles first, and for playing 3D games or high-end 3D rendering then.

With the advent of the OpenGL API and similar functionality in DirectX, GPUs added programmable shading to their capabilities. Each pixel could now be processed by a short program that could include additional image textures as inputs, and each geometric vertex could likewise be processed by a short program before it was projected onto the screen.

Indeed the workload for the creation of a 3d scene is more or less the same for every pixel and vertex of the scene, so that an architecure with hundreds or thousands of simple computation units performing floating point algebra following the same rules is the ideal solution. The costs could be kept low because the basic architecture was very simple.

NVIDIA was the first to produce a chip capable of programmable shading, the GeForce 3 (code named NV20). By October 2002, with the introduction of the ATI Radeon 9700 (also known as R300), the world's first Direct3D 9.0 accelerator, pixel and vertex shaders could implement looping and lengthy floating point math, and in general were quickly becoming as flexible as CPUs, and orders of magnitude faster for image-array operations. Pixel shading is often used for things like bump mapping, which adds texture, to make an object look shiny, dull, rough, or even round or extruded.

In 2006, though, NVIDIA released the G80 GPU. It was the very first GPU with programmable generic stream processing units. Thus the G80 could be exploited to make general purpose computation, not necessarily related to the creation of an image to be displayed.

Actually there are two alternatives to develop gpu enabled programs: OpenCL [3], and NVIDIA CUDA [4]. While the former generates code usable on a wide range of devices such as GPUs from different manifacturers and multi core CPUs with different architectures, the latter can only do it for NVIDIA GPUs; still, being targeted for a narrow scope of devices, it is more optimized to exploit all the capabilities of the specific hardware. I will analyze only the NVIDIA framework, which is the most used and probably the easiest to start gpu development with, but the ideas showed in this thesis should apply to any of these parallel development environments, with exception of the CUDA only dynamic parallelism (cap 3), that will be supported soon anyway also by OpenCL.

2.2.1 NVIDIA GPU architectures

Before showing the programming model, it is important to describe a bit the hardware that will run the CUDA programs, since it is fundamental to analyze and understand the performance of the programs that they execute.

Since CUDA has been very appreciated in different fields of the information technology, from financial to physics and medicine, NVIDIA continued to improve the general purpose computing capabilities of its GPUs with every new architecture, and even producing some specific models designed specifically for this purpose, without any capability of showing an image on a display. From 2006, when it was introduced, there have been mainly three significant architecture advancements, G80, Fermi and Kepler, as described in the next sections. Within a single family, moreover, the same architecture is configured in different ways: as example the GPU could be sold with different numbers of active cores, with faster or slower memory, with different core clocks and so on to cover all the slices of the market. Still the capabilities of the GPU within the same family don't vary.

G80

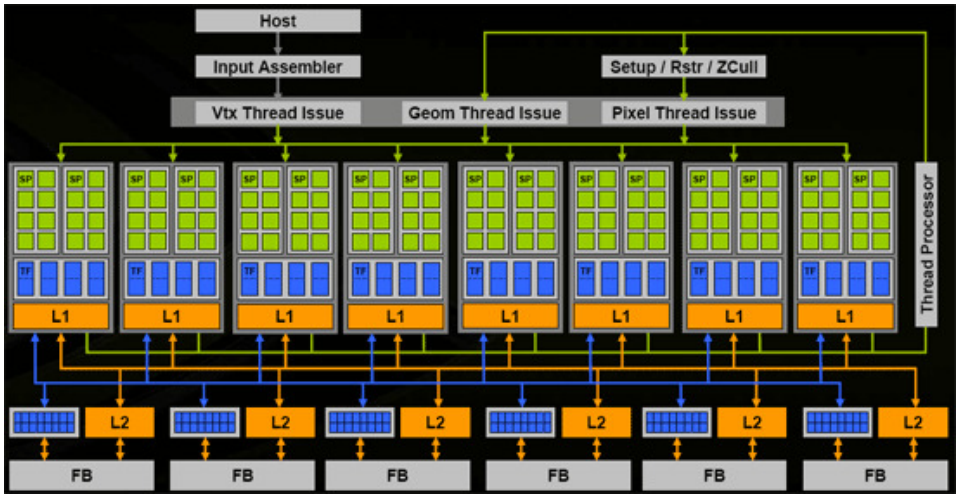


Figure 2.1: Sketch of the NVIDIA G80 streaming processors array architecture.

The G80 chipset is the first that unifies all the shaders (pixel, vertex, geometrix or physics) under one computational unit, called Stream Processor,

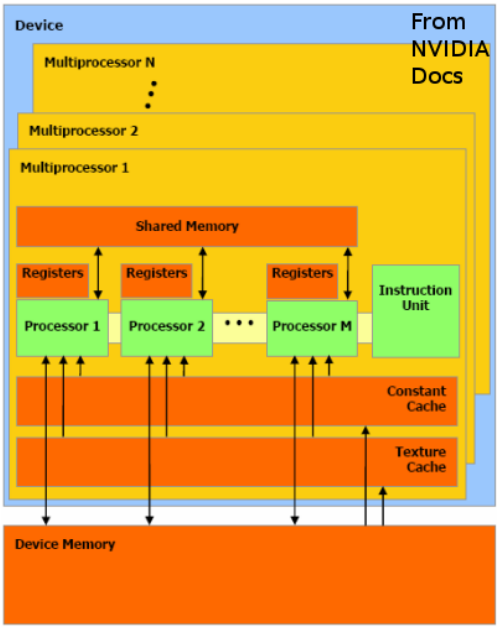


Figure 2.2: Sketch of the model of a shader processor in the G80/G92 gpus.

that is the core of CUDA. As we see in FIGURE 2.1, in one GPU there is an array of identical SPs.

In the sketched fig. 2.2 the layout is made clearer: there are many streaming multiprocessors (SM), where each has a number of SPs, each with its own **Registers**, one common **Shared Memory** and one **Instruction Unit**, that is in charge of mapping active threads on the available SPs.

Within the SM there are also the read only **Constant Memory** and the Texture Memory. The device memory, called **Global Memory** resides outside the multiprocessor, being effectively the DRAM of the board, available to every multiprocessor. The image also shows that the only memory locked to one SP is the register space, while each other memory is shared between all the SP of one multiprocessor or between different SMs too. A SP is a fully pipelined single-issue processing core with two ALUs and a single floating point unit (FPU). Single integers operations are performed with a 24bit precision.

SM are in turn grouped by two or three in Texture Processor Clusters (TPC), that includes support for Texture processing, though these features are seldom used for general purpose computing and will not be investigated in this thesis.

Evolutions: Fermi, Kepler

Figure 2.3: Sketch of the NVIDIA GF100 streaming processors array architecture.

The Fermi architecture made its debut in the end of 2009 and greatly enhanced the GPGPU capabilities, as a response to the notable interest gained by CUDA. The GPUs used in chapter 4 to test the performance of GTRex are both based on this architecture, that is today largely diffused in most personal computers. Fig 2.3 shows the new blocks (still called SM, streaming multiprocessors), now with 32 or 48 cores each. SPs are now called Cuda Cores, showing the intent of NVIDIA to push and promote the CUDA technology. With a total of 16 SM, GF100, the reference GPU for the Fermi family, can reach 512 cuda cores. Each cuda core executes 1 warp of 32 threads.

One of the most interesting extra of the new core is the presence of an L2 cache between the SM and the global memory that relaxes the constraints to obtain coalesced memory accesses with the first generation GPU and also

allows a more efficient usage of the global memory. This is indeed a very important aspect as will be shown below.

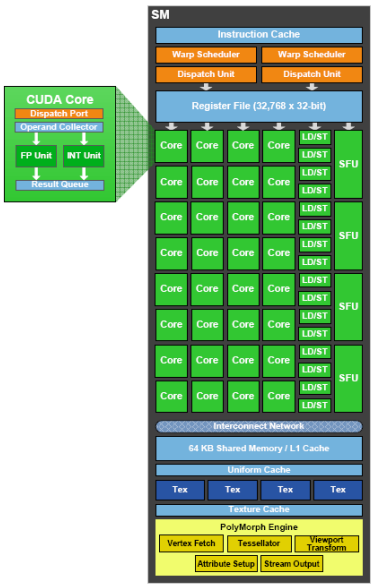


Figure 2.4: Sketch of a multiprocessor on the GF100 gpu, with the new cuda core.

Fig 2.4 shows the new SM, even if it remains quite similar to the preceeding ones. The interesting news are the number of registers available, now 32768 per SM and the new INT unit to make operations with integers natively. This feature is another example of enhancements introduced specifically for the GPGPU, since vertex and pixel computations rely on floating point math.

Finally, with Kepler, released in 2012, the GPGPU capabilities of the NVIDIA GPUs were enhanced even more, with a brand new SM (shown in FIGURE 2.5), now called SMX, grouping 192 Cuda Cores and bringing a new specialized unit to allow dynamic parallelism, a major enhancement for CUDA that will probably make even more profitable the usage of CUDA to accelerate parallel applications.

On the other hand the GPUs based on Kepler are still high priced and not so much diffused, so that I hadn't the chance to try this innovation for GTRex. More considerations on possible Kepler advantages for the project are shown in the conclusions of this thesis.

Figure 2.5: Sketch of a multiprocessor on the GF100 gpu, with the new cuda core.

2.3 CUDA

The CUDA toolkit is composed of a compatible driver, a closed source compiler (nvcc) and a set of libraries to interact with the gpu from within a normal C program. Many wrappers exist that allow the usage of cuda in different programming environments, but it is designed for C and C++. To interact with the graphic hardware there are two methods, not compatible one with another. They are **C for CUDA** and **Driver API**. While the first is simpler to use, the latter exposes more options and low level functions, giving a better level of control over the hardware. For the project the C for CUDA approach was used, since it allows to write a smaller and simpler code granting at the same time all the features exploited by the engine. In the following I will not analyze in details all the specific functions to call in order to initialize the environment; all those informations can be found in the NVIDIA CUDA Programming Guide [2]; I will try to give an overview of the basic principles to understand in order to program with CUDA.

From now on I will call **host** code the program that will run on the CPU, while the program run by the graphic card will be referred to as **device**.

2.3.1 Programming model

The most important concept in the development of a cuda application is the **kernel**: it describes the job that will be done by the gpu. It is the equivalent of the main() function of the host code. The keyword to declare such a function is **__global__**, and it must return a void type. From a kernel the programmer can call other functions run by the gpu and declared as **__device__** that don't need to necessarily return a void.

The memory space accessed by the kernels is completely separate and independent from the memory of the C program using CUDA, at least in the most general and more efficient case where the unified address space is not used. Thus, in order to give the GPU the data to work on, specific functions to copy memory areas to and from the GPU must be used.

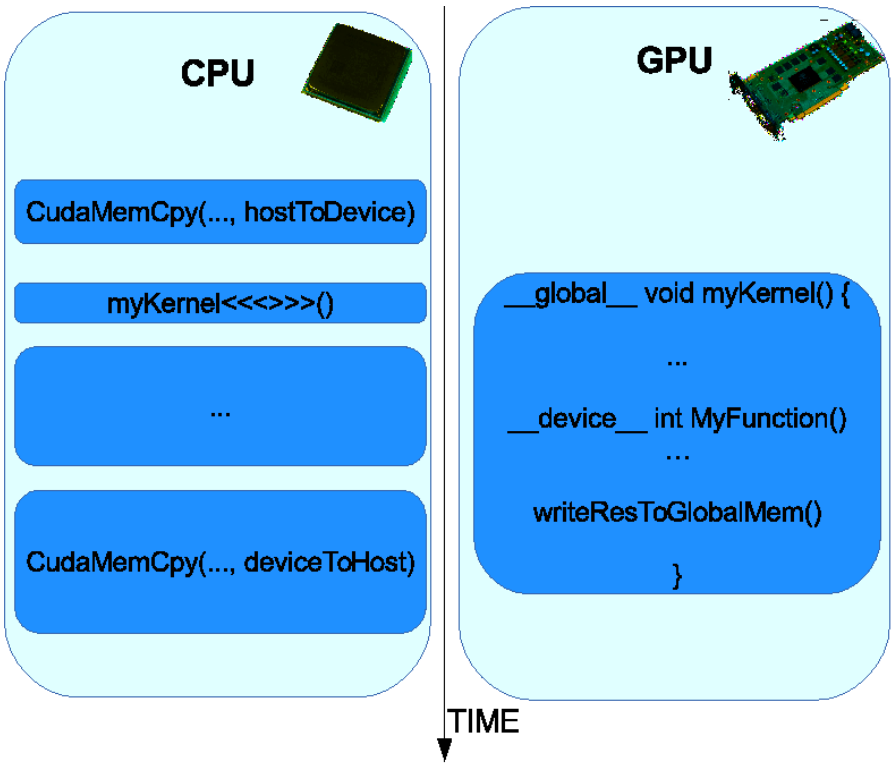


Figure 2.6: Sketch of the typical workflow in a CUDA enabled program.

FIGURE 2.6 represents the typical workflow: the program at a certain point asks the CUDA driver to copy some data to the device, then it invokes the kernel that will be executed on the GPU. The kernel launch request on the host side is asynchronous and typically very quick. From this moment the CPU and the GPU work in parallel. On the GPU the kernel can call other functions that will be executed on the device and share the memory space with the callee; at the same time the CPU can do anything else. Finally the host program asks for the resulting data to be copied back to the host memory. This call instead is typically blocking, since the completion of the kernel is necessary for the data to be copied back to be valid and consistent. In this way the memory copy from the device will end a bit later than the end of the GPU computation, because of the time necessary to phisically move the data.

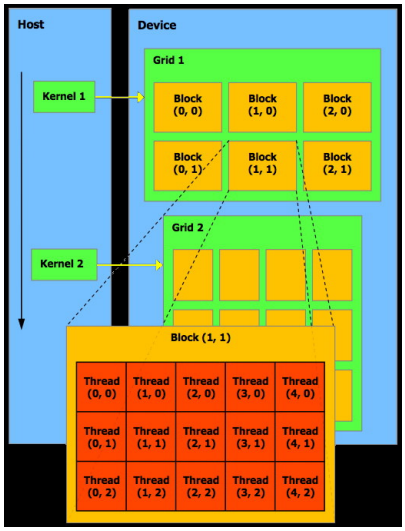


Figure 2.7: Sketch of the grid of threads in cuda.

2.3.2 Computing resources organization

The amount of parallel processing units on a GPU is great, but coordinating and managing all them is not trivial: the following hierarchical model is used.

From the smallest to the biggest, the massive amount of computing units is grouped as follows: **thread**, **warp**, **block**, **grid**, as shown in FIGURE 2.7. More in details 32 threads form a warp, a custom number of threads form a

block and a custom number of blocks compose the grid, that characterizes the kernel launch.

- thread: it's the basic computational unit; this means that the code of the kernel will be effectively executed by each thread. Each thread is uniquely identified in a kernel launch thanks to the intrinsics **threadIdx**, **blockIdx** and **blockDim**. All these three-dimensional variables have **x**, **y** and **z** subfields. Indeed the threads can be organized in multi dimensional grids. As example, in the common convention, the first thread of a 1 dimensional grid will be the one having threadIdx.x and blockIdx.x equal to 0

- warp: the warp abstraction is hidden to the developer, since there's not any routine or API call to modify a warp anyhow. Indeed it is directly bound to the physical configuration of the GPU. It's important to know of its existence because it is a set of 32 threads that in a given moment will be performing exactly the same instruction, since it is supplied by a single instruction issuer. So, when a kernel includes some divergent paths, if all the threads of one warp will take the same path it would not harm the result too much, but this will be shown in SECTION 2.3.5

- block: the block dimension must instead be set by the developer, and can be read from the kernel code with the blockDim variable. It is a 1,2 or 3 dimensional group of threads that share the same shared memory and that will be scheduled for execution at the same time by the driver. The optimal choice of the dimension of one block is a multiple of 32 - that is a multiple of the dimension of a warp -, because otherwise there would be for sure some shared processors idling for the duration of the entire kernel.

- grid: the biggest grouping entity is the grid. It is only 1 in a kernel; like for the threads and blocks, though, it provides the **gridSize** variable, that represents the number of blocks per dimension (x, y and z like before) and can be accessed from each thread. This comes useful when the developer has to map different memory areas to each thread within his kernel.

One trivial example for the grid organization, where only one dimension is defined for each entity: if we launched 10 blocks, each with 32 threads, we would have a total of 320 threads numbered from 0 to 319, with blockIdx.x varying from 0 to 9 and threadIdx.x varying from 0 to 31 and blockDim.x fixed at 32. Each thread could compute its index and be uniquely identified as $idx = threadIdx.x + blockIdx.x * blockDim.x$.

2.3.3 Memory spaces

The memory area where data is copied to from the host code is physically resident on the GPU DRAM; there are many reasons for which it shouldn't be used as the default memory for the code execution and there are better alternatives offered by CUDA, that gives different kinds of memory (shown in fig. 2.8) to rely on, each with different pros and cons. The following list enumerates the memory spaces available starting from the "outer", phisically the furthest from the SP or Cuda Core, to the inner, the nearest.

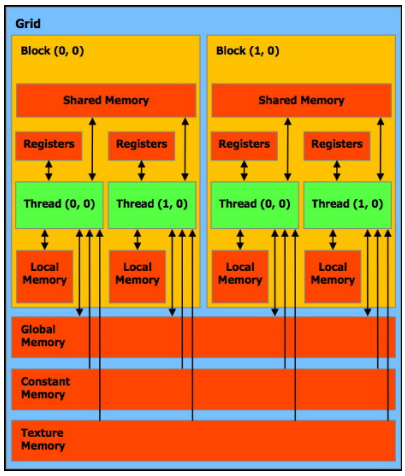


Figure 2.8: Memories available in the device code.

- **Global memory**: it is phisically the DRAM of the graphic card, so it is the biggest space available in the device code. In fact almost the whole quantity of ram (a little space for the displayed image must not be requested) can be assigned to a single pointer. It has a huge bandwidth (more than 80GB/sec), but also a huge latency (around 400-600 clock cycles). This means that it is perfect to fetch very big and contiguous memory areas, but it is not recommended for a random

access to few bits per time. In the Fermi and newer architectures this disadvantage is partially masked by the shared cache to global memory. This is the only memory that makes possible the return of values from the GPU to the host.

- **Constant memory**: as the name itself indicates, this memory is accessible in a read only way from the device code, and it can be written only from the host code. It is limited in size (64kbyte), but, being cached, it is useful when many threads concurrently access the same memory location.

- **Shared memory**: this small sized memory (from 16kb up to 48kb) is shared between all the threads of one block. It is very fast, but the programmer should take care to reduce concurrent access to the same memory block, otherwise the performance could drop.

- **Registers**: they are tied to one specific thread during the execution, so that they are really fast and they can't suffer from concurrent access, but they are limited and an excessive usage can restrict the number of concurrent threads really executed at the same time. Their number increases from G80/G92 to GT200 and to GF100 and GK110, and starts from a minimum of 8192 to a maximum of 65536.

So the general solution to exploit efficiently the resources of the GPU is: copy the data to the global memory, let each thread read the needed data in its registers or in shared memory, wether the threads of a block should cooperate on the same data, and finally write back the results to the global memory, from where it will be copied to the host memory.

Memory access coalescing

Because of the limits of the global memory, that is mainly a huge latency, it is important to organize access to this memory so that with a single transaction a lot of data can be fetched (remember also that it has an incredible bandwidth). The requirements for this to happen have relaxed from generation to generation of gpus, but I will now show the requirements needed to achieve it on every gpu family, which have been used in the current project. From the CUDA programming guide, we know that the global memory access

by all threads of a half warp is coalesced into one or two memory transaction if it satisfies the following three conditions:

- each thread must access either 4, 8 or 16 bytes words, respectively coalesced into a 64, 128 or 128 bytes memory transaction
- all 16 words must lie in the same segment of size equal to the memory transaction size, or twice the memory transactio size when accessing 16byte words
- threads must access words in sequence: the k^{th} thread must access the k^{th} word of the memory segment

. If one of these conditions is not met, then 16 different memory transcations are issued, with a throughput significantly reduced. Again, with Fermi and Kepler these conditions have been relaxed a lot, so it is much easier to write an efficient code without all these constraints.

2.3.4 GPU Occupancy

As said above the resources of the GPU are limited, and they will affect the real level of parallelism that a specific kernel will reach: since every kernel will need a predefined number of register per thread to work, and sometimes also shared memory, there will be often a cap to the maximum number of threads that will effectively run in parallel for a given kernel. As example if we have a kernel that needs 30 registers and runs on a G92 gpu, depending on the configuration of threads per block the occupancy of the gpu will range from 17% to a maximum of 33%. In fact with the G92 chipset each multiprocessor has 8192 registers, so each multiprocessor will run $8192/30 = 273$ threads at maximum, divided in blocks of 128 threads each result in 2 active blocks per multiprocessor. But the theorical limits of the gpu allow it to run 768 threads per multiprocessor, so the occupancy is $256/768 = 0.33$. Besides this, the more multiprocessor a gpu has, the more threads it will really run in parallel (in the example 256 more threads per multiprocessor). It is also important to note that a different configuration can change the occupancy of the gpu: as example if for the same kernel we would have launched 192 threads per block, then only one block of threads could have run at a given moment on one multiprocessor, and the occupancy

would have dropped to $192/768 = 0.25$. For this kind of considerations there is the CUDA Occupancy Calculator, a tool released by nVidia that will plot the real occupancy of a given GPU with the register/shared memory usage set by the user. Moreover the number of register used by one thread can be capped giving some instruction to the nvcc compiler, but it is important to note that not always an higher occupancy will bring better performance, because to reduce the number of registers some data can be shifted to the global memory, that has an high latency.

2.3.5 The SIMD architecture

There is a big difference between writing a working code and writing an optimized code. On the gpu the hardware limits make a fine development essential to achieve the results expected from the huge computational power available. It is suggested to read the CUDA Best Practices Guide from nVidia to get some useful tricks in the optimization of the code. This is for sure the most important concept to understand in order to write cuda enabled applications with good results. What SIMD means is Single Instruction Multiple Data. This means that many computational units need to execute exactly the same instruction every other unit is doing on its own chunk of data. In fact one instruction issue unit make each thread of a specific warp perform the same instruction; as a result if only one thread of a warp needs to do a different task the parallelism is broken: in an unpredictable way first some units than some others stop working actively and just complete some dummy loops waiting to receive some good instructions, while the remaining part do some useful work. It is easy to realize that is far from the ideal case, and the resulting performance of the program is heavily compromised. Anyway sometimes it will be probably impossible to completely avoid this to happen, so in those cases the best options is to reduce the most you can the portions of code that may result in divergent paths to the shader processor of the same multiprocessor. Note that this doesn't mean you should always avoid conditional statements in your device code. In fact if one conditional statement gives the same result for all the threads launched with a specific kernel, then the performance is not affected, since there is no divergence in the execution path of the different threads. Also, when possible, the nvcc compiler will automatically avoid divergence with an equivalent machine

code.

2.3.6 CPU-GPU parallelism

One advantage of the gpgpu that has been extensively used in this project is the ability to concurrently run different instructions on the gpu and on the cpu. In fact when the programmer launches a kernel from the device code using indifferently the C for CUDA or the runtime API, almost all the functions used to interact, under some circumstances, with the GPU are asynchronous. This means that as soon as a memory copy or a kernel launch is requested the algorithm on the cpu can go on, without waiting for the job of the gpu to be completed. All the operations, unless otherwise specified, are still scheduled sequentially by the CUDA driver.

In order to synchronize the host program with the GPU workflow there are specific functions that can be used, both to stop the execution of the host program until the GPU has done all that has been scheduled and to poll the execution state without blocking the host program.

Chapter 3

GTrex implementation

3.1 Introduction

In this chapter I will show the key result of the project: a new GPU accelerated processing engine for TRex. I will start with an overview of the entire engine and of the algorithm that controls the GPU computation, then I will describe the algorithm employed for the core processing tasks, that was already existing in the test project done by Cugola and Margara [6]; finally I will show many other enhancements that optimized the code in terms of performance and features.

3.2 The new server

In the first place it is important to say that this project involved only the engine component of the middleware. In fact the new CUDA powered engine can be installed in the middleware without any modification to the rest of the system, excluding of course some code to actually activate and take advantage of it. So the communication, queue control and management components of the system are exactly as they were before. Using the new engine is as simple as declaring a new GPU Engine and pass to it the rules and event notifications, exactly in the same way done with the CPU engine. Figure 3.1 depicts the new server, where two independent engines are created: one is the old plain CPU engine, the other is the one developed in this thesis. The interface and the interaction between the two engines and the rest of the middleware is exactly the same for both the engines. The main algorithm

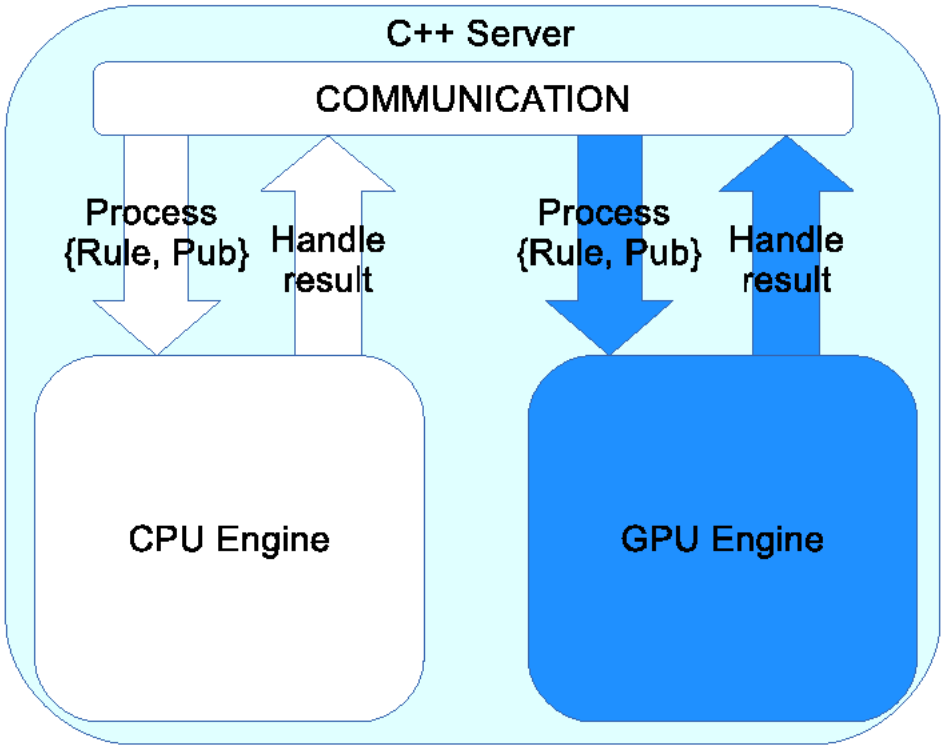


Figure 3.1: Overview of the new TRex Server.

employed by this GPU engine is the CDP introduced in CHAPTER 2.

3.3 CPU code

The GPU engine itself is modularized, as shown in FIGURE 3.2: there is a GPUEngine class that is instantiated only once during the initialization of the server of the middleware. The engine then initializes the memory manager; this is also a unique component, that is created once during the initialization of the engine and controls all the GPU memory requested by the program. Then, for each rules installed in the engine, a GPUProcessor is created: it takes care of the management part, that is coordinating the kernels executed on the GPU with rest of the work to be done to make everything work. The GPUProcessor also creates its CudaKernel, that is a class compiled by the NVIDIA CUDA compiler that actually holds the CUDA kernel code and launches computation when requested by the GPUProcessor. When an event notification is received, the GPUEngine passes it to all

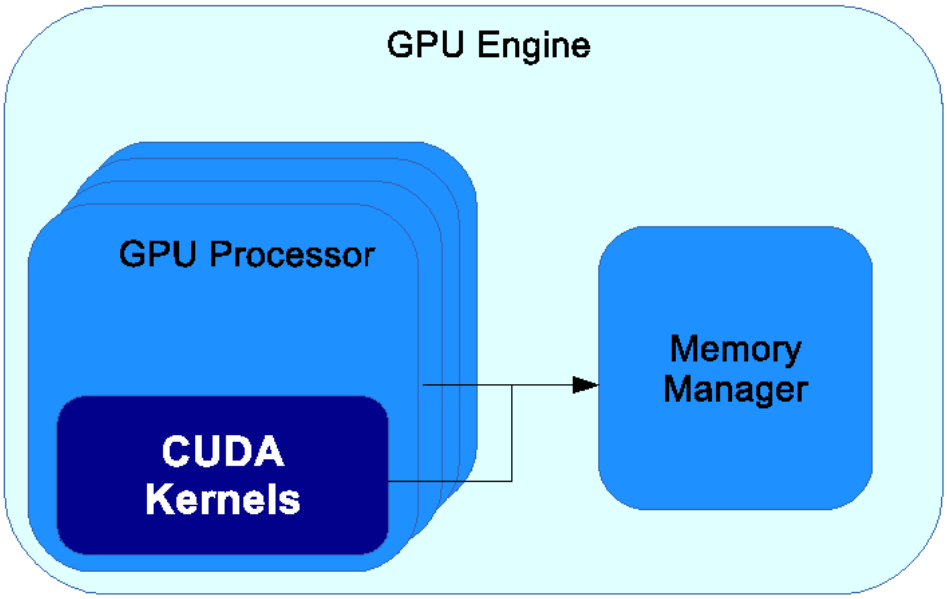


Figure 3.2: Overview of the GPU Engine.

its processors, that eventually store it, in case it is relevant for its rule. If the event is a terminator, the real computation that creates new valid sequences is triggered.

The main idea is the one described in algorithm 3.3.1. Note that this code actually runs on the CPU, and only the functions labelled with <> are invocations of GPU kernels.

The computation begins with a single partial sequence containing one event: the terminator. Then the program loops through all the states of the deployed rule (line 2); at each step a new state of the sequence is analyzed to look for new potential events to be added to the partial sequences constructed up to that moment. The switch at line 7 is used to determine the best path to take for the specifications of the state that will be analyzed: the selection policy, the presence of negations and even the size of the kernel to be launched determine which kernel will be used. Moreover the information needed to set up the GPU computation are set by the prepareForNextLoop function. This must be called for the first time before the first round of the loop (line 1), but it is then computed on the CPU while the GPU is busy at line 23 for the following states. Indeed an explicit synchronization with the GPU is requested at line 25, so that, if the CPU ends its part before

the GPU, it waits for the results on this blocking call. The number of valid partial sequences found during the computation of a state is read back from the GPU memory at each step (line 24): if it is 0 the computation can be stopped since there won't be any valid complete sequence in any case. There is the risk that the computation performed at line 23 is then thrown away when the GPU returns 0 results (line 27), but since the job is done in parallel this doesn't have consequences on the performance. The partial sequences eventually returned at each loop will be one event longer than those given as input at the beginning of the loop.

From the algorithm there are also many other aspects worth noting. One interesting aspect is that there is already the usage of the MemoryManager that will be introduced later in this chapter: it is identified by the *mm* pointer, and offers wrappers for memory operations to the GPUProcessors and CudaKernels. In order to identify the calling rule, the *mmToken* is created when one GPUProcessor subscribes to the memory manager, during the initialization phase, and is assigned to the GPUProcessors, that, then, must use it when calling a memory manager method.

If at the end of the loop resultsSize is a positive number, partialSequences are copied back to the host memory and analyzed to compute their aggregate functions and create the resulting complex events. These resulting sequences are processed sequentially and in order by the CPU, but the aggregate computation is performed by an efficient CUDA reduction kernel, described in SECTION 3.5.3. The creation of complex events instead is left to the CPU, but it is a very easy and light task, and it's also partially overlapped with the GPU execution.

3.4 CDP on CUDA

The CDP [seen in SECTION 1.4.2] is the only algorithm that can be efficiently adapted to the CUDA environment, because of its simple data structures and its delayed and parallelizable computation that can be handled by the GPU. The first thing to take care of, though, is that it would be very unefficient to stick with columns of pointers as happened in the CPU implementation. In fact that would cause a memory fragmentation leading to terrible performance on th GPU. As a result in this implementation the

Algorithm 3.3.1: Overview of the handling code

```
1  loopKind = prepareForNextLoop();
2  described foreach state in states do
3      cudaMemsetAsync(mm->getResultsSizePtr(mmToken), 0);
4      if size==0 then
5          resultsSize = 0;
6          return;
7      switch loopKind do
8          case MULTIPLE
9              computeComplexEventsMultiple()<>;
10         case SINGLE
11             computeComplexEventsSingle()<>;
12         case MULTIPLENEG
13             prepareNegationsInfo();
14             computeComplexEventsMultipleWithNegations()<>;
15         case SINGLEGLOBAL // SINGLENEG
16             cudaMemsetAsync(mm-
               >getCurrentResultsPtr(mmToken), 0);

17             prepareNegationsInfo();
18             computeComplexEventsSingleG()<>;
19             mm->swapPointers(mmToken);
20             reduceFinal()<>;
21             mm->swapPointers(mmToken);
22     if state > 0 then
23         loopKind = prepareForNextLoop();
24     e=cudaMemcpyAsync(resultsSize,
       mm->getResultsSizePtr(mmToken));
25     cudaDeviceSynchronize();
26     if resultsSize==0 then
27         return;
28     else if resultsSize>0 then
29         mm->swapPointers(mmToken);
```

columns hold copies of the events, with their attributes and timestamps. On the other hand the CPU still has to delete old events and set parameters for the CUDA kernels, so the timestamps of the events must be kept also in the host memory. For the sake of simplicity from now on when talking about events on the host memory I am indeed referring to their timestamps, that is the only information actually stored.

When a non terminator event enters the engine the algorithm is similar to the standard version: it still adds the event to the related columns, that are resident on the GPU global memory. When a terminator arrives the program deletes old events analyzing the timestamps on the CPU memory and starts the computation. This is where the CUDA kernels are launched and the main computation takes place. Informations about deleted events, as well as any other information needed to complete the computation, is passed to the CUDA kernel with specific parameters.

As example. remembering the history of events presented in the table 1.1, at time 12 the computation is started; the first two events, with a timestamp lower or equal to 2, are deleted, while the other events are copied to the device memory. At this point the appropriate kernel for the specifications of the given state is launched to analyze the events and find new partial sequences.

Algorithm 3.4.1 shows a simplified kernel for the computation of a multiple selection state without negations.

Algorithm 3.4.1: CDP Algorithm on GPU, Multiple Selection

```
1  x = blockIdx.x*blockDim.x+threadIdx.x;
2  y = blockIdx.y;
3  timestamp = stack[x].timestamp;
4  previousTimestamp = prevResults[y].infos[referredState].timestamp;
5  valid = (timestamp < previousTimestamp && timestamp >
   previousTimestamp-win);
6  if checkParameters(x, y)==0 then
7  |  valid = 0;
8  if valid!=0 then
9  |  writeIndex = atomicAdd(currentIndex, 1);
10 |  currentResult[writeIndex].infos[state] = stack[x];
11 |  copyPreviousEvents(writeIndex);
```

This is the most general form of the algorithm. It doesn't compute

negations and it has some simplifications, but the basic idea is there. A 2D grid of CUDA thread blocks is launched, and within each block there is an array of threads. So each thread will have a unique (x,y) pair: x identifies the new primitive event that will be analyzed from the thread, while y identifies the partial sequence found so far that will be checked against the primitive event. The thread performs a check on the timestamps at line 5, on the parameters at line 6, and, if everything matches, it writes the new partial sequence: this will be like the one that the thread used as input with the addition of the primitive event x. The write is done in a dedicated memory area called currentResult, at lines 10-11 of the algorithm.

To do this it uses a special function called atomicAdd (line 9): since all the lines of code are executed in parallel by all the threads within a block, in order to be able to write an ordered output sequence without memory conflicts, some kind of synchronization is needed between threads. This is where atomicAdd comes useful: it increments the variable pointed by the first argument by a quantity specified by the second argument, and it does it so that no other thread can interfere during the operation. Finally it returns the new value, as it is after the addition. This means that, thanks to this function, each thread will have a unique incremental writeIndex used to write without conflicts to the output array.

At the end of the computation the CPU code swaps currentResult and prevResults as seen before, so that the output sequence of a step becomes the input of the next one.

This is the most general form of the algorithm: different code paths have been written in practice, to optimize as much as possible the code produced by the CUDA compiler.

FIGURE 3.3 is an overview of the algorithm and its data structures with the events from table 1.1 at the beginning of the computation at time 12. In the input array of partial sequences there is only the terminator event received, L. In the column for the current state, instead, there are the 3 O events that haven't been deleted. In this case 1 block with 3 threads is launched. These threads will have indexes (0,0), (1,0) and (2,0), where the first number is their index within their block and second number indicates the block in which they reside. Thread (0,0) and thread (1,0) terminate their execution when checking the parameters of the events, since they don't match

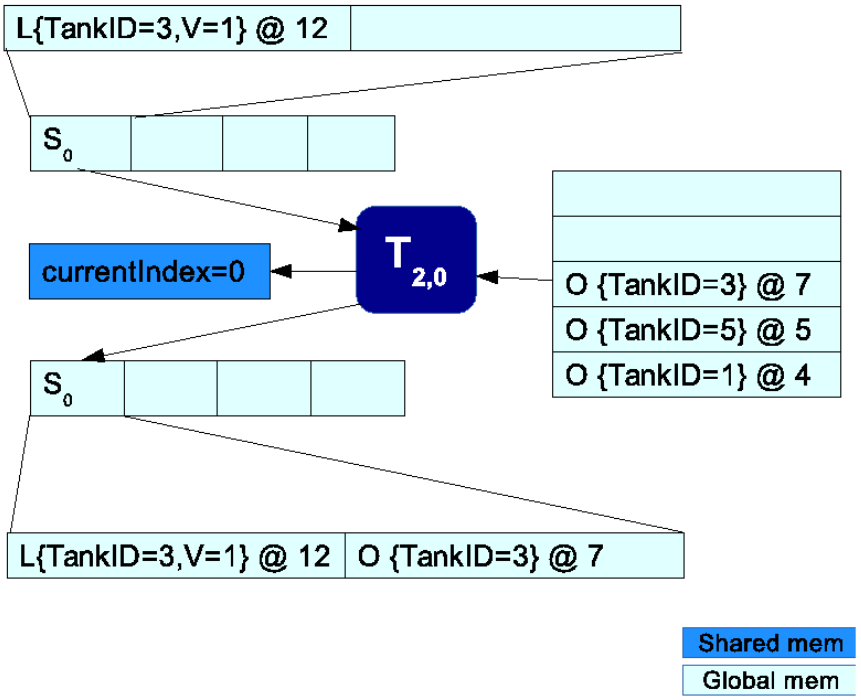


Figure 3.3: Algorithm execution example, Multiple Selection.

the TankID attribute of the terminator received. Thread (2,0), instead, will succeed. At this point it will increment the currentIndex variable, reading at the same time the value 0; finally it will write the new valid partial sequence found in position 0 in the output array. If, for example, 2 threads survived the parameters checks and had to write the new partial sequence, one of the two would have found the value 1 in currentIndex and would have written its partial sequence in position 1 in the output array and so on.

In case of a single selection operator, instead, it is a bit more tricky, as shown in algortihms 3.4.2 and 3.4.3.

The first part of the algorithm is identical to that of the multiple selection case; the differences are all in the code that handles the write of the result to the global memory. Indeed when the single selection policy is used, 1 input partial sequence can be combined with at most 1 simple event from the column; thus many threads will have to compete so that only one, that with the oldest or the newest event from the column, survives and writes its result to the global memory. One important aspect in this version is that the results are not correctly written in an ordered array ready to be copied

Algorithm 3.4.2: CDP Algorithm on GPU, Single Selection, last-within

```
1  x = blockIdx.x*blockDim.x+threadIdx.x;
2  y = blockIdx.y;
3  timestamp = stack[x].timestamp;
4  previousTimestamp = prevResults[y].infos[referredState].timestamp;
5  valid = (timestamp < previousTimestamp && timestamp >
   previousTimestamp-win);
6  if checkParameters(x, y)==0 then
7  |  valid = 0;
8  if valid!=0 then
9  |  oldMax = atomicMax((int *)&maxIdxlocal, x);
10 |  if x < maxIdxlocal then
11 |  |  return;
12 |  oldMax = atomicMax((int *)&(maxTS[y]), x);
13 |  __threadfence();
14 |  if maxTS[y] == x then
15 |  |  currentResult[y].infos[state] = stack[x];
16 |  |  copyPreviousEvents(writeIndex);
```

on the host memory as happened in the multiple selection case with a simple atomic instruction. Indeed the atomicMax function is provided by CUDA, but simply using that with the single selection case, when more than one compuational block is launched, would lead to wrong results. This is caused by the scheduling of the NVIDIA driver: different blocks don't run in parallel for sure, it depends on many aspects like the physical availability of resources on the GPU chip.

So, using an approach similar to that used in the multiple selection case, it may happen that more than 1 result is considered valid. As example one event could be the last at a certain point of the computation, thus it may be written to the output sequence; but, in a block that still hadn't reached that point of the computation, there may be another event with a greater timestamp: this would be again considered last and written to the output sequence.

In the end a kernel wide synchronization barrier is needed; unfortunately CUDA doesn't offer anything like that, and the only method to work around this problem is to launch two different kernels: in this way the second kernel will be executed only after the completion of the first one.

So the algorithm 3.4.2 does the following:

- line 9-10: a local max is determined within each block thanks to a temporary __shared__ variable and to the atomicMax function, that behaves just like the atomicAdd shown before but computes the maximum. In this way only 1 thread per block should survive. This preliminar filter is important for performance reason: reducing the number of surviving threads here reduces the number of conflicting memory access in the global memory in the next lines, that are much more significant in terms of performance

- line 11-12: the surviving threads repeat the operation, this time though on a __global__ variable that is accessed by all the blocks of the kernel. The write location, though, is identified by the y variable, so that only threads analyzing the same partial sequence will compete for the addition of a new event

- line 14: at this point only the wanted event should have been written to the currentResult[y] location.

In this way, on the other hand, the resulting memory structure isn't compatible with the rest of the algorithm: an ordered array of partial sequences is needed. This is the role of the reduction kernel 3.4.3.

Algorithm 3.4.3: Reduction method used for Single Selection

```
1 x = threadIdx.x + blockIdx.x * blockDim.x;
2 if prevResults[x].infos[state].timestamp == 0 then
3 |   return;
4 writeIndex = atomicAdd(currentIndex, 1);
5 copyPreviousEvents(writeIndex);
```

The method to distinguish valid partial sequences that must be returned to the host is simple: the memory area where they are stored is zeroed with a specific cudaMemset function, and the informations are written over only if the requirements checked in algortihm 3.4.2 are ok, as per line 15-16; so it is enough to check if the timestamps of the events are greater than 0 to make sure that the related partial sequence must be kept (line 2 of algorithm 3.4.3). In fact 0 shouldn't be a valid timestamp for any real event.

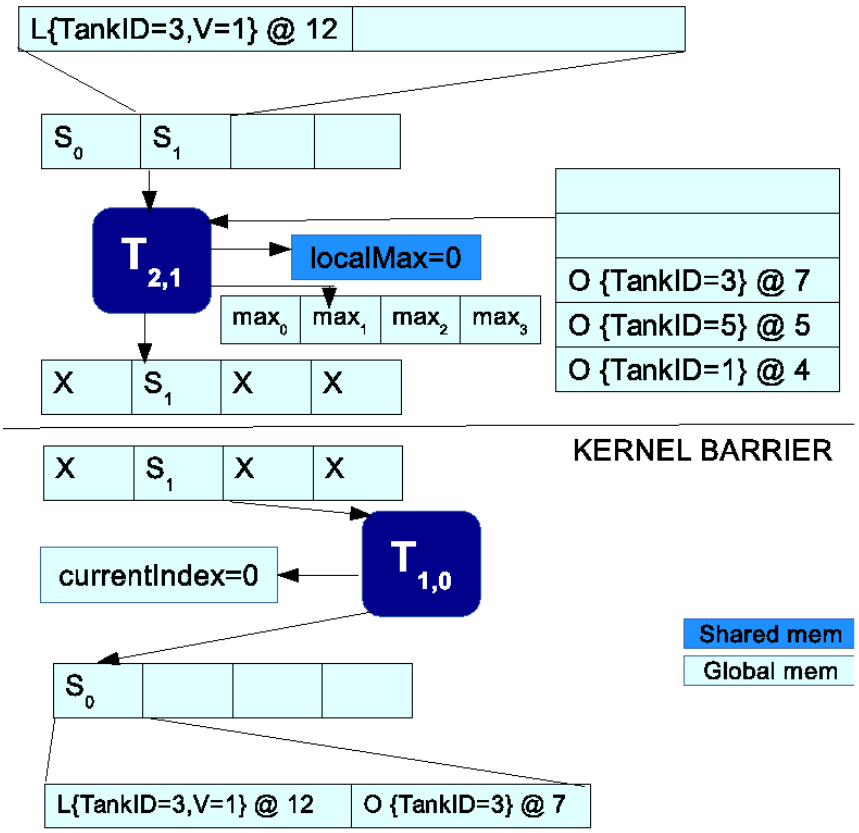


Figure 3.4: Algorithm execution example, Single Selection.

In order to comprehend better the algortihm consider FIGURE 3.4, where, one more time, the packets from table 1.1 at time 12 have been considered along with a last-within selection policy. This time suppose that the input partial sequence is in position 1 (thus in position 0 there will be another input partial sequence, but we're not interested in that). The first part is just like in the multiple selection case. When thread (2,1) convalidates its simple event this time it first checks to be the last within its block, with the help of the atomicMax function called on the shared variable *localMax*. In this example localMax will be updated form the initial value of 0 to the value 2 by thread (2,1).

In case it's not it ends its computation, otherwise it performs the second check, again with the atomicMax function, but this time on a global memory variable, still initialized to 0. In this way all the threads checking the same input partial sequence will compete on the same global variable. If one more

time the threads finds itself to be the last it writes the partial sequence in the same position where the input sequence was (position 1 in the example). In this way when this threads ends the array of partial sequences will have in each position a valid partial sequence combined with the appropriate last simple event or a partial sequence having an event with timestamp 0. Thus the second kernel launched with a 1-dimensional block configuration will simply have to check this timestamp and reorder the array in the same way done in the multiple selection case, with the *currentIndex* stored in global memory so that the computation is correct even with more than 1 block of threads.

Note that the simple version reported here computes only the last-within operator; in case of a first-within the greater-than operators become less-than, and the atomicMax function becomes atomicMin.

Finally note that the single selection case can work in this way only if the columns store events in order, so that, a thread with a greater index will have for sure an event occured later in time. Indeed using the timestamps themselves to choose the first or the latest one would be theoretically right, but would also mean using atomics with 64bit arguments in shared memory, and this is possible only with the latest hardware supporting cuda capabilities 3.5. On the other hand the assumption should always be respected, since the timestamp of an event entering the engine should always greater than that of the precedent one.

One more consideration about performance is that if only one block is enough to take care of all the new events, there is no need for global synchronization among the kernel launch, so a special kernel is used that completes the computation immediatly, like in algorithm 3.3.1 at line 11.

3.5 Compatibility

In order to make the draft of the GPU code work as a complete GPUEngine many aspects have been taken care of, like the following:

- memory structures
- negations support

- attributes could only be of INT kind, leaving out FLOAT, BOOLEAN and STRING kinds
- aggregate computation could handle only the SUM function, leaving out MAX/MIN/AVG/COUNT operators
- the consuming clause was not supported at all

3.5.1 Memory Structures

The first step was to adapt the code of the GPU test project to use the structures used in the TRex2 project. This, though, was not possible without some modifications. In fact the CPU code relied a lot on pointers and C++ specific libraries. Both these elements were incompatible with the requirements of a CUDA implementation, but can be handled by the host code that manages the GPU usage. So in the end the solution of keeping the original class structure for the events was choosed. When an event has been accepted and must be copied to the GPU, it is converted in a new static data structure that perfectly fits the GPU requirements. This is done on the fly when needed, but it demonstrated to be very fast, thus not being a considerable overhead for the project. Also the need for statically allocated memory areas brought to the usage of circular buffers for columns, so that the same memory area can be used to hold infinite events, if its maximum capacity is not surpassed in any moment by the size of the window of events.

3.5.2 Negations

One entirely new, non trivial problem, was that of negations. Events relevant to the negations defined in a rule are store in dedicated columns, just like any other event. The first implementation followed right away the CPU one: the correctness of a potential partial sequence was checked after the sequence itself was created. This is the simplest approach, and it can even be easily parallelized over the possibly big number of negations that have to be checked against the sequence.

On the other hand, though, the idea works correctly on multiple selection predicates, but not on single selection ones. Indeed, if the first possible sequence found is invalidated by a negation, then the computation is not

over in the single selection case: it must continue until a valid sequence is found or there are no more sequences to test.

There were two possibilities to rewrite the negations checking code to work around this problem:

- let each thread of the kernel loop over all the negations to validate the new event during the creation of the partial sequences, in the same kernel
- launch a new kernel that fully exploited the great parallelism offered by the GPU to check all the negations against all the partial sequences and all the new candidate events to signal the non usable ones.

Both these approaches have advantages and disadvantages. The former doesn't require a specific launch configuration, thus it can be coalesced in the kernel that performs the creation of new partial events, and it does exactly the least amount of work necessary to complete the computation. On the other hand it performs a loop with a non predictable length, keeping each thread warp being occupied by the longest loop in the warp even in case of a shorter loop.

The latter instead completely avoids any kind of loop, making a huge usage of the parallelization; this launches a great number of threads and blocks. Indeed one dimension states which partial sequence will be checked, another one states the new simple event that will be checked for addition to the partial sequence, while the last dimension states which negation event will be matched with the partial sequence and the new event. In this case though there is some amount of work that is not necessary; in fact all the combinations of events and partial results are checked with all the negations received, but just one valid negation is sufficient to invalidate a partial sequence. It also requires a specific launch configuration with a 3d grid of blocks, that can quickly grow a lot, and, because of the single selection requirement of having already filtered out the incompatible events when choosing the first or the last one, it must be launched before the kernel that creates new partial events. This means again work thrown away, in case the related event doesn't pass the timestamp or parameter test.

In the end both approaches have been implemented, and it is possible to switch between them setting a preprocessor definition at compile time, but

the former one, that loops over negations during the computation, demonstrated to be more efficient in most test cases. The algorithm 3.5.1 is a simplified version of the GPU function that is called from the kernels computing new partial sequences. It takes the new possible event ev1, the negations related to the state of the rule being computed in that moment and the related partial sequence, called here prevResults, as inputs and returns a bool: if true the partial sequence must be invalidated, while it's ok otherwise.

Algorithm 3.5.1: GPU function that controls negations

```
1  foreach n in negations do
2      negTimestamp = stack[n].timestamp;
3      maxTS = prevResults.infos[negations[n].upperId].timestamp;
4      if negations[n].lowerId < 0 then
5          minTS = maxTS - negations[n].lowerTime;
6      else
7          minTS = ev1.timestamp;
8      if negTimestamp <= maxTS && negTimestamp > minTS then
9          if checkParameters(ev1, n)==0 then
10             return false;
11 return true;
```

3.5.3 Aggregates

Aggregate computation was also already present in the test project, but only for the SUM function and the int data type. It is still almost the same; each aggregate is computed exploiting parallelism but sequentially, one after another, for each new complex event generated by the computation. The kernel used is the typical reduction one, promoted also by NVIDIA to solve a wide range of problems exploting the GPU parallelism, modified to handle parameter checking.

Let's see how the algorithm works with the help of FIGURE 3.5.

First it is important to say that the number of threads per block must be a power of 2, and it must be greater than half the number of elements to be considered in the computation. In the example there are 7 events selected for the aggregate, so 1 block with 4 threads is needed.

Each thread t_x reads two events from the global memory from the indexes

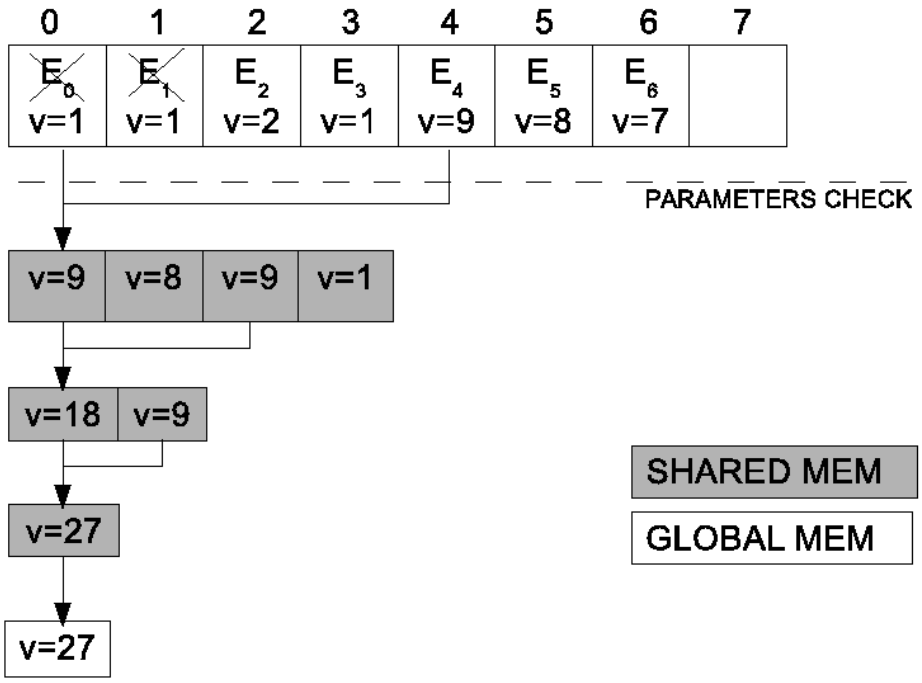


Figure 3.5: Workflow of a reduction kernel.

x and $x + blockDim.x$, so that memory access are contiguos and coalesced as much as possible. For each of these events the parameters are checked in the same way they were checked in the kernels that find new valid partial sequences: if the parameters check doesn't succeed, the value of the event read is changed with the neutral value for the function to be computed. In the same way, if the index of the element to be read is out of the bounds of the array of events to be reduced, the value is replaced with a neutral element. Back to the example: the function to be computed is a sum: t_0 reads e_0 and e_4. The parameters of e_0 don't match, so the v attribute of e_0 is set to 0, that is the neutral element for the sum. The same thing is done by all the active threads; when t_3 has to read from index 7, since there is no valid event in position 7, it uses again the neutral value.

Finally each thread t_x computes the function of the its pair of values and stores it in position x. As example t_0 computes $0 + 9 = 9$ and stores it in position 0.

The algorithm continues halving the number of threads at each step until only 1 thread survives and copies the final value from position 0 in the shared

array to the global memory.

In this scenario the main limit to the number of threads per block that can be launched is represented by the quantity of shared memory available: to make the program work on any Fermi based GPU, the limit has been set to 512. So if there are more than 512 events to be computed for the aggregate more than 1 block is launched, obtaining a number of partial results equal to the number of blocks launched (each block will produce one partial result). Depending on the number of these partial results, the reduction can be finished on the CPU, with a simple sequential computation over the array of partial results, or launching again a reduction kernel, that this time won't check parameters correctness.

Note that this algorithm could correctly be executed entirely on global memory; copying the values in shared memory and working there is done for performance reasons. Extending the support for the remaining functions, that are COUNT, AVG, MIN, MAX, and for the float data type was quite trivial. For the MAX and MIN functions the neutral value is respectively a very big positive or negative number, while to COUNT and AVG functions simply need to have a count of the valid events read. This is accomplished with an atomicAdd on a shared variable within the block. Also the support for parameter on aggregates has been added. This is another quite a simple thing: the computeVal function has been extended to check all the parameters, and in case one doesn't match, the value of the returned attribute is set to 0.

3.5.4 Consuming clause

Another feature missing in the sample project was event consuming. Unfortunately this can't be parallelized in any way and must be handled entirely on the CPU. The events are checked and deleted on the host memory only, and the corresponding memory page (on next section) on the device is flagged as invalid, so that not any CUDA operation is fired until a new terminator for the related sequence arrives. One problem arose during the development of this aspect was that of events uniqueness. On the cpu code it was simple to discern two packets of any kind simply checking their address in memory; in fact the code is written to store only one single copy of each event entering the engine to reduce memory usage, and to use pointers to events when

needed. This was not the case with the GPU implementation. Data must be really copied from the host to the device memory for it to be accessible from a CUDA kernel, unless the very slow host mapped memory is used, but this can be a good idea only if the GPU is an IGP, not the case we're interested in. So, to state if two events are the same or not, the code checks its characteristics: the timestamp first and the event type first, followed by all their attributes, in terms of kind and value. This should be ok as long as two identical events enter the engine: in that case the behaviour of the engine is undefined and wrong results may be observed. On the other hand, at least theoretically, this particular situation should never happen. Extending the code to properly handle also this case is very trivial, but requires an ID field to be added to the event class, thus requiring at leadt 4 more bytes for every single event, so I decided not to take this path and leave the possibly broken events comparison algorithm.

3.6 Memory manager

As pointed out by Cugola and Margara in their paper [6] one of the pitfalls of the test project was memory usage: the really trivial memory allocation used, along with the lack of big memory availability on GPUs, made it impossible to store more than about ten rules at the same time on the GPU. To overcome this limit I added a brand new paged memory manager to the project, with the goal of a smarter memory allocation allowing more rules to be stored on the device at the same time, more transfer efficiency and also bringing support for memory swapping between the device and the host memory.

3.6.1 Pagination

The first goal of the memory manager was to overcome the limitations imposed by the statically allocated memory areas. This is done thanks to the pagination of the memory used by the engine. Let's make an example: if the user wanted each rule handled by the TRex engine to be able to cope with 100k events in a single window, in the original GPU implementation he had to statically allocate during the initialization phase the memory needed to store 100k events for each column of each rule installed, without even

knowing if that memory would have really been used thereafter. As example a rule with a large window and few incoming events would result in a huge memory waste. Now instead the maximum size of an events window, that can still be specified, doesn't imply any prior memory allocation.

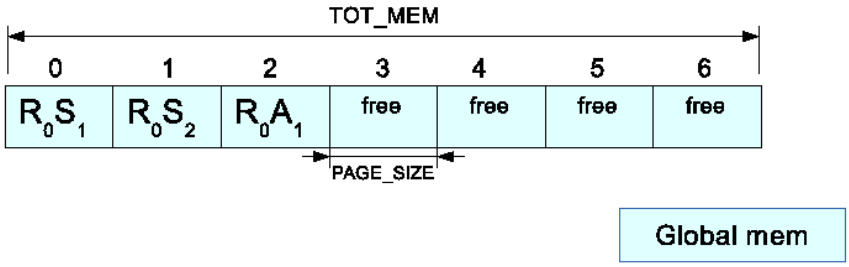


Figure 3.6: Paged memory.

A page of memory is actually a purely abstract concept. As shown in FIGURE 3.6 the physical memory of the GPU instead is allocated all at once, with one single CudaMalloc call, during the initialization phase of the engine. The size of the allocation is specified in the MAX_SIZE definition, thus it is completely under the user control. The address returned by the allocation function is then stored in the constant memory on the GPU. This is indeed the ideal usage of this kind of memory in CUDA: small amount of data accessed concurrently by many threads, that will be able to read values in a very short time thanks to the cache of the constant memory. Then the block of allocated memory is splitted in pages, where each page holds exactly the number of events specified by the PAGE_SIZE flag. In the beginning all the memory pages are free; when a new rules is installed in the engine, the related processor makes a subscription to the memory manager, and is provided with one memory page for each state of its sequence and for each aggregate or negation that it holds. As example in the FIGURE rule R_0 has 2 states and 1 aggregate, and it is the first subscribing to the memory manager. It gains access to the first 3 pages, while the others remain free for other rules or if R_0 will need more pages. Indeed if and when more pages will be needed the memory manager will automatically assign a new page to the rule, if there is any available. All the memory operations are executed by the memory manager, that decides the best path to take to optimize resources usage. A specific memory structure of the memory manager keeps track of the mapping between the page indexes for a rule and page indexes in the

main memory allocation. FIGURE 3.7 is an example of such data structure for the rule R_0 of the previous example: the rule has 2 states, 1 attribute and 0 negations; the maximum allocation size set for a single column equals 3 times the size of a single page, so that 3 indexes are allowed for each columns of each kind.

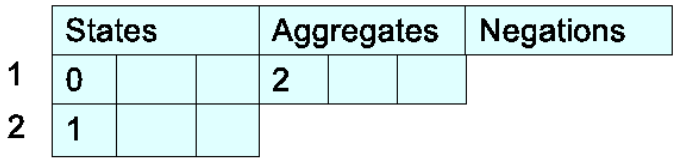


Figure 3.7: Paged memory.

With such a data structure for every rule installed in the engine, it is easy to keep track of the pages assigned to different rules and it is quick to access the correct event knowing the rule ID, the kind of the column wanted (state, aggregate or negation) and the index of the event. Accessing the wanted event is as simple as doing a division and a modulo operation. Assume for example a page size of 10 events; event 13 will be in page $13/10 = 1$, in the rule domain, in position $13 \mod 10 = 3$. Once the page number (1) and the event offset (3) are known, along with the base allocation address obtained in the initialization phase, it is simple to compute the memory address.

Once a page has been assigned to a rule, it is never released until the corresponding GPU processor quits and calls the unsubsribe method of the memory manager. Luckily this is not a problem and even if an oversized maximum allocation size per column has been defined, in case it is needed the page won't remain unused, thanks to the swapping feature implemented in the project.

3.6.2 Swapping

Pagination itself is already very useful to reduce improper memory usage, allowing the definition of theoretically big events' stacks without wasting a lot of memory space in case the real number of event is substantially smaller. Still it is also the prerequisite for another great feature to optimize memory usage by the engine, that is memory swapping.

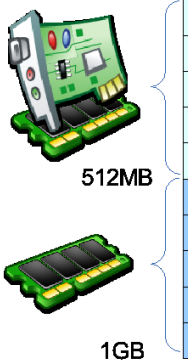
Moreover when old events are filtered out because they go out from the window, the corresponding space is not freed, that is the corresponding mem-

ory page in not released by the owning rule. This may be a waste if the window is small when compared to the maximum allocation size parameter, beacuse in that case only a small portion of the requested memory space would be actively used. One solution could be to dynamically release memory pages when no more used, but this would also introduce complications and would require new data structures to keep track of the exact memory allocation requested by each rule. One simple workaround would be to get rid of the static maximum allocation size parameter and better size the maximum number of events per stack to suite the related window size. This could be easy if more constraints on events timing would be introduced.

Swapping memory overcomes this limit: in fact each device memory page can be shared by more than 1 rule.

The basic idea is to exploit the quantity of host memory avialable on modern computers: it is usually much greater that the GPU memory, and can even be expanded easily and without big economic efforts. So, at compile time, in GTRex, it is also possible to specify the host memory multiplicator parameter. This states the number of times that the total GPU memory used is allocated on the host computer. As example setting the GPU memory to 512MB and the host memory multiplicator to 2, 512MB of ram will be used on the GPU and 1GB of ram on the host. As a consequence each memory page on the GPU could be shared by up to 2 different rules. It is important for the sharing rules to be different because the computation for a single rule is completed atomically, without other rules operations being processed in the meantime, and all the events belonging to the specific rule must be available on the physical GPU memory for the computation. With the underlying structures for memory pagination it was quite easy to add this new feature; basically 2 new fields were needed to map virtual pages, allocated only in the host memory, to real pages in the GPU memory and to signal to which rule the events in the device memory belong. FIGURE 3.8 shows a schema of the new structure.

In the depicted example the there are 2 rules installed in the engine. 512MB of GPU ram are used and the host memory multiplicator is set to 2, while the page size is set to 1024. In a possible situation rule R1 obtained 4 memory pages from the memory manager, while 3 pages are needed by R2. Since the GPU memory can host up to 5 pages, the computation wouldn't be



ID	GPU_ADDR	RULE_ID	EVT_INDEX	ON_GPU
0	0	1	[0-1023]	2
1	1	1	[1024-2047]	1
2	2	1	[2048-3071]	1
3	3	2	[0-1023]	-1
4	4	2	[1024-2047]	2
5	0	2	[2048-2047]	
6	1	-		
7	2	-		
8	3	1	[3072-4095]	
9	4	-	-	

Figure 3.8: Example of a paged memory configuration

possible without swapping. In the scene depicted when R1 asks the fourth page, the free pages were finished; so the memory manager loops through all the virtual pages (with indexes 5-9) until it finds the first not already assigned to rule R1 and not already shared by 2 other rules. Page 8 meets the requirements and is given to R1. On the other hand page 8 is allocated in the same space of page 3 in the device memory.

When an event enters the engine and is relevant for any deployed rule it is copied and kept on the host memory. Also the memory manager checks if the page where the new event is copied is already loaded in the device memory and, if that is the case, performs immediatly an asynchronous memory copy. In this way it is never necessary to copy back events from the device memory to the host one.

When a terminator event is received and the computation must start, the memory manager makes sure that all the related events are really on the device memory in the address where the CUDA kernels expects them to be. This is accomplished checking all the required pages: if they are already on the device memory there's nothing to be done, otherwise the copy is immediately asked and performed before the computation begins.

To keep track of the rules to which the events in the pages in the device memory belong the onGpu field is used, as shown in the FIGURE 3.8 example. -1 is used to invalidate the page so that is will be copied again to the device memory whatever rule will be using it next. It is used as example when consumed events are deleted. In the example, if rule R1 receives a terminator, memory manager copies the whole page 0 from the host memory

to the device memory page 0, then it copies the host page 8 to the device page 3 and then the computation can begin. If instead of a terminator a normal event was received, as example to be added to page 8, the memory manager would have only added the event to the host memory, since page 3 is not assigned to rule R1 in the same moment.

Altough quite complex, this memory swapping mechanism proved to work quite well (as shown in SECTION 4.13.2) and allows consistent memory saves, allowing even GPUs with 512MB of ram (under the current average), to store and manage a great number of rules concurrently.

Chapter 4

Experimental results

4.1 Introduction

In this chapter I will illustrate a big selection of test conducted to evaluate the performance of the GTRex engine opposed to those of the original one based on the CDP algorithm on CPU. Evaluating the performance of a CEP engine is not easy as it is strongly influenced by the workload, characterized by the type and number of rules and the events to process. Unfortunately there are no publicly available workloads coming from real deplyments, so the tests performed here have been created ad hoc to test the engine in different situations over a wide parameters space and to test the engine where it could show its limits.

4.2 Configurations

The tests have been done on 2 different computers: one with a low end graphics card (table 4.1) and one with an mid range one (table 4.2). Excluding the graphics cards, the two configurations are comparable: 4gb of ram, that is the quantity available on the lower specs pc, is enough for any kind of test computation done in this chapter, and the host CPU is very similar in terms of architecture and clock frequency, but has more physical cores on the higher end configuration. Thus the results of the CPU engine running on the two configurations are very similar when only 1 rule is processed, and in the graphs they have been unified, for the sake of simplicity.

The GPU engine used is the result of this project, the brand new CUDA

powered engine for TRex, while the CPU engine is the latest version of the CDP based TRex engine, that makes use of multiple CPU cores and showed good results when many rules are installed in the engine. It is the default engine currently used in the TRex middleware.

Table 4.1: Components of PC1

PART	MODEL/QUANTITY
CPU	AMD Phenom II x4 965 @3400MHz
RAM	4GB DDR3
GPU	GeForce GT520 512MB (GF110)
OS	Ubuntu 13.04 64bit

Table 4.2: Components of PC2

PART	MODEL/QUANTITY
CPU	AMD Phenom II x6 1055t @2800MHz
RAM	8GB DDR3
GPU	GTX460 1024MB (GF110)
OS	Ubuntu 12.04 64bit

4.3 Workload

The base workload is exactly the one used by Cugola and Margara in their paper [6] describing the first GPU implementation of the TRex engine.

Rule R2

define	$CE(att1 : int)$
from	$C(att = \$x)$ and last $B(att = \$x)$ within 1000000 from C
	and last $A(att = \$x)$ within 1000000 from B
where	$att1 = Sum(A(att = \$x).value$ within 1000000 from B)

This simple rule considers a sequence of three states, one for each event type. Two parameters are defined on B and C so that all the events composing a complex one have the same value for the integer attribute att. Also an aggregate is defined to compute the sum of the attributes. The total number of events given to the engine is 100000, and they are equally distributed over all the definitions of the rule. Moreover the window size is set so that

not any event is deleted because of its timestamp during the entire computation. As a result in the base scenaraio each column will hold on average 33333 events. Starting from this base scenario many parameters have been changed to track the response of the engine in different situations.

4.4 Parameters settings

The new GPU engine provides different parameters that can be tweaked in a single header file to compile it for various workloads. This is necessary because of the static allocations needed by the GPU engine to work efficiently; they describe the maximum capacity in different fields, like the maximum number of rules that can be handled or the maximum number of parameters that a state in a rule can have. If oversizing most of these parameters has the only consequence of wasting some memory, 2 of them affect seriously the performance; they are the maximum number of states for a rule and the maximum number of attributes that an event can have.

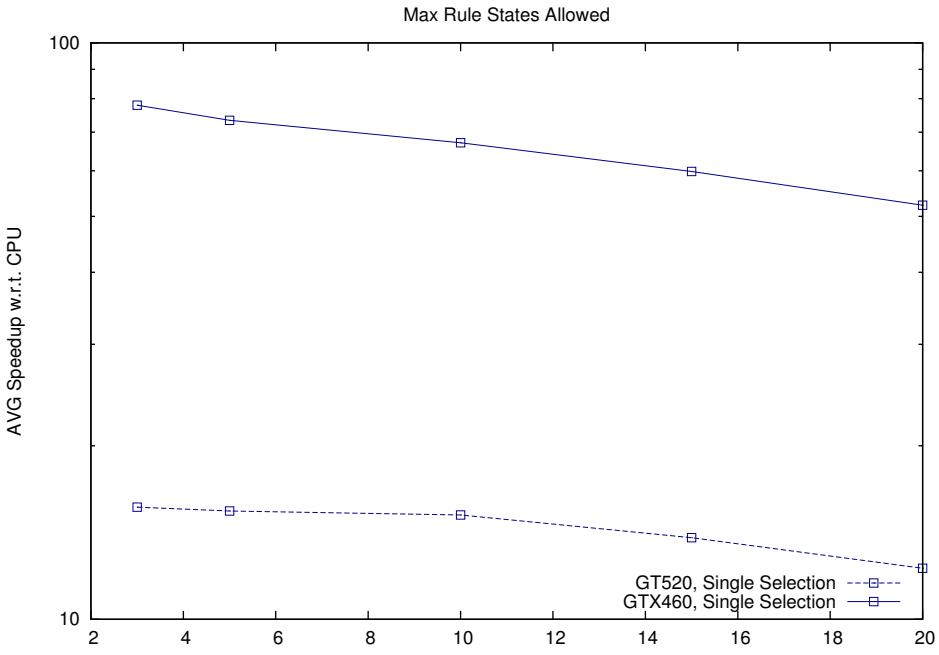


Figure 4.1: Impact of an oversized max rule fields parameter

Graph 4.1 shows the impact of different max rule fields parameters on exactly the same workload: the speedup decreases with larger values. Note

that the very same parameter is defined and used also by the CPU engine, but thanks to its memory structures based largely on pointers, its performance is not affected by the parameter. On the GPU, on the other hand, more unused slots in partial sequence containers mean more memory fragmentation, thus worse performance.

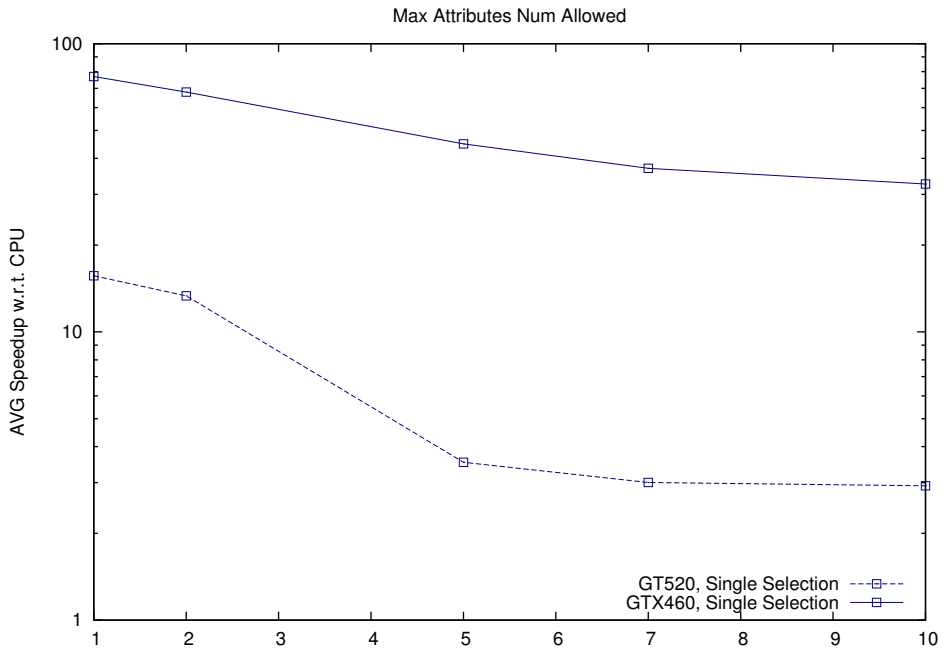


Figure 4.2: Impact of an oversized max attributes

Exactly the same happens for the max attr parameter; this time the unused memory slots are in each event space allocated in the GPU memory and the performance are affected even more. This kind of parameter is not even defined in the CPU implementation, where attributes are in a dynamically allocated array in an event packet.

In the following tests these parameters have been set to the optimal value for the specific workload. In reality it could be impossible to know exactly a priori the characteristics of the workload.

4.5 Testing methodology

All the following tests, comparing the performance of GTRex with those of the standard CPU engine, were done locally, with the two engines acti-

vated in the same middleware instance. Also the events notifications were generated within the same program, with an evaluation component built ad hoc for these simulations.

Feeding the engine directly with packets eliminated the impact of the communication layer in the measures. Moreover the complex events notifications generated triggered exactly the same execution paths for both the engines, and ad hoc tests confirmed that it didn't alter the results in any way.

The initialization phase was left out from the time recordings: only the time to process the single packet and signal the creation of the complex event was registered from outside the engines in order to conduct an analysis as much consistent as possible.

4.6 Base Scenario

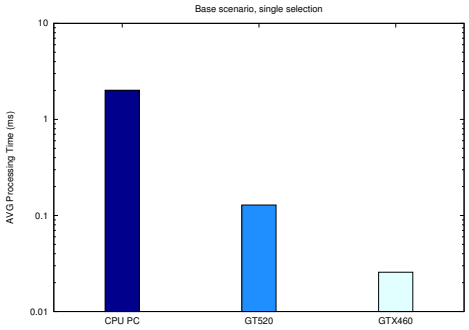


Figure 4.3: Single Selection.

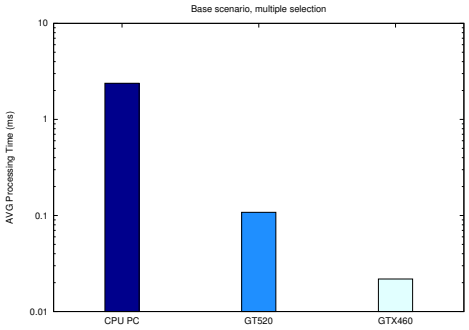


Figure 4.4: Multiple Selection.

Figure 4.5: Base Scenario

In the base scenario the advantage of the GPU is consistent and significant. Indeed the scenario is very suited for the GPU computation: it has many events per column, that occupy enough the GPU in terms of workload, it has a single rule, that limit the CPU sequential management code, and it is hard to find valid complete sequences, because the chance that all the parameters are met is quite low (only 1800 to 2500 complex events have been generated in these tests). This latest point reduces the advantage that the sequential code has in the computation of single selection constraints, because it can stop as soon as the first valid sequence is found.

Moreover, while the computation time for the single selection policy on the CPU is lower than that of the multiple selection policy, on the GPU the situation is turned upside down. Indeed when a single selection policy is applied the CPU algorithm stops the computation as soon as the first valid element is found; the GPU instead when has to deal with single selection policy must fire two consecutive kernels to complete the computation correctly. On the other hand the CPU when dealing with a multiple selection policy must check all the events in the column in any case, just like the GPU does.

The speedup is incredible: the GT520, in spite of its irrelevant price, already gives an average speedup of 15X in the single selection case and 21X in the multiple selection. The GTX460, on the other hand, reaches respectively an average speedup of 77 and 110 times.

4.7 Length of sequences

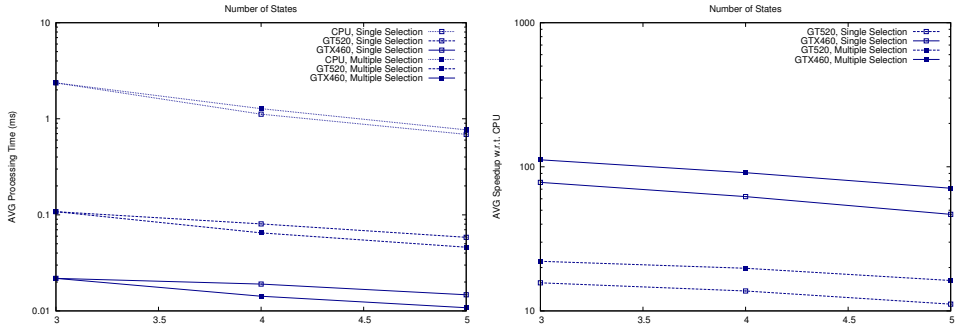


Figure 4.6: Varying Sequence Length Scenario

The main aspect that influences the performance when varying the sequence length while keeping anything else constant is the number of events per column. Indeed more states bring to less events per column, that go from above 30k when there are 3 states to about 20k with 5 states. As a consequence the rule becomes simpler to compute and the average computation time lowers both on the CPU and on the GPU. The speedup of the GPU over the CPU is lowered, since the amount of work parallelizable decreases, but still consistent even in the 5 states test, because 20k events per column are still enough to exploit the SMs of the GPUs. Again the multiple selection policy is faster than single selection on the GPU, while the opposite happens on the CPU.

4.8 Number of Values

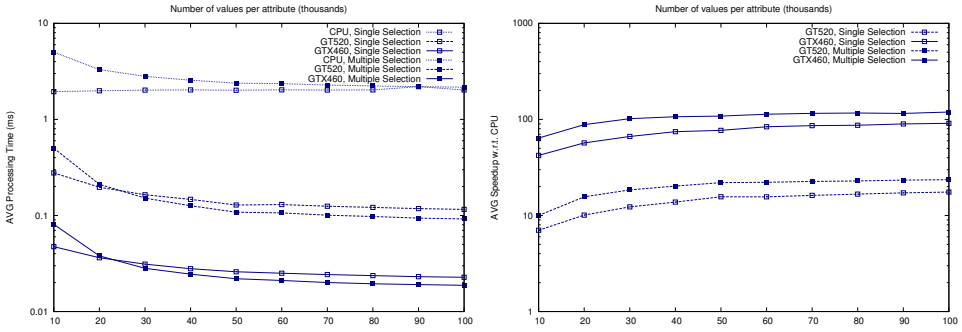


Figure 4.7: Varying Number of Values

With number of values the amount of possible values for the attribute att is indicated. A lower number increases the possiblity that the parameters of events in the same partial sequence match and thus increases the number of complex events created as well as of partial computation brought on by the engines before being discarded. A lower number of values impacts on the performance of the GPU engine more than on the CPU engine; indeed more complex events created mean more atomic access to the shared variables in the CUDA kernels, an so more serializations and performance drops. Also moving full partial sequences from the device memory to the host one has a cost in terms of time, and it raises with the number of events created. Finally having less possible values helps the CPU engine in the single selection scenario, where the first valid event in a column found is enough to stop the computation on that column. In the end the speedup becoming lower with less possible values was expected, but, unless the rule becomes very simple and a single selection policy only is useed, the speedup remains consistent.

4.9 Size of windows

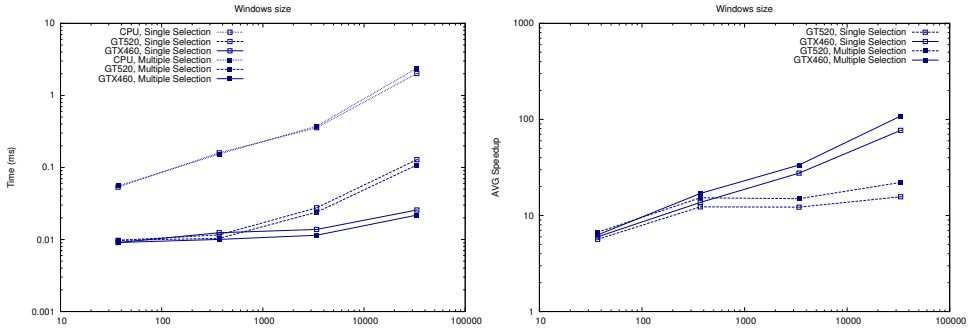


Figure 4.8: Varying Size of Windows

Reducing the size of windows has the same effect as increasing the number of states: there are less events in the columns to be computed in parallel, thus a smaller window means less avdantage for the parallel hardware. Note that the relation between windows size and number of events per column in this example is not 1:1. It is very interesting to see how below a certain windows size the performance of the GT520 and of the GTX460 become comparable: it happens because below a threshold of about 370 events per columns the GTX460 has more computational units than needed, that remain unused. This is totally expected, remembering that the GT520 has 1 SM with 48 cuda cores; each cuda core executes 32 threads concurrently, so that the limit of threads running in parallel on the GT520 is 1536, more or less the point at which the performance of the two GPUs become comparable. At the same time the GTX460 is capable of executing up to 10752 threads concurrently, and this is reflected in the average computation time that starts growing only in the test with about 30k events per column.

4.10 Number of Rules

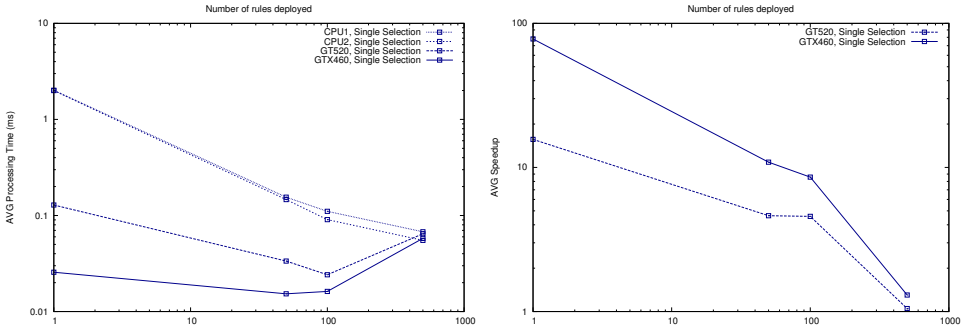


Figure 4.9: Varying Number of Rules

While the time needed by the CPU implementation decreases when more rules are installed, that of the GPU one increases, causing also a rapid decrease of the average speedup to 1. The cause is mainly found in the number of events stored in the columns when a computation is launched. As example, if with only one rule deployed each column will store about 33K events, with 50 there will already be around 660. Considering a block size of 256 in the CUDA configuration, this results in only 3 blocks launched per kernel, far less than an ideal situation and not enough to exploit the computational power of the GPUs used. Moreover the quantity of work to find the right columns for events and for work distribution that must be done sequentially on the CPU increases with the number of rules, helping the convergence of the time taken by the two implementations for the average computation. With 500 rules this job becomes the relevant one, and the performance of the engines converge. Note also that 500 rules mean only about 66 events per column, far too few to test the performance of the engines and to create any complex event.

4.11 Negations

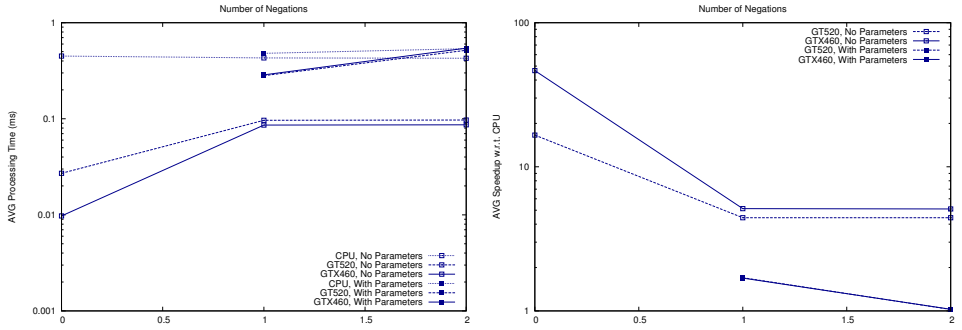


Figure 4.10: Varying Number of Negations

Support for negations is one of the new features brought by GTRex; as said in Cap. 3, two different algorithm have been developed for this new feature, that is indeed not an easily parallelizable operation. It is not possible to know the exact size of the execution in advance, and lots of computational units are wasted in the kernels, since few threads can keep the whole warp busy. For these tests the fastest version has been used; the other options should be maintained because it could be a good base for future developments exploiting the dynamic parellelism of the Kepler GPUs (refer to the conclusions for further info). Indeed these pitfalls are reflected in the graphs, that show a clear drop in the speedup when 1 or 2 negations are added. The CPU time is almost identical in the different tests, while that of the GPUs raises notably when the first negation is added. Adding parameters to negations impacts again on the sequential code executed by each thread on the GPU, raising once more consistently the average computation time; at the sime time it doesn't affect the CPU performance a lot. Another effect to consider is that adding parameters to negations increases the number of complex events generated, with the additional effect of damaging the GPU performance because of the data copied from the device memory.

4.12 Basic Single Selection Rules

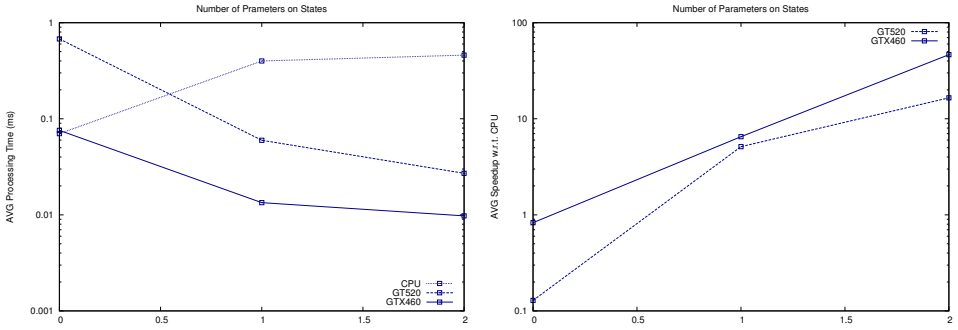


Figure 4.11: Varying Number of Parameters between states

These tests are very interesting because they show the most common situation in which the CPU implementation still outperforms GTRex, even on a single rule. In this case a very simple rule, without any parameter between constraints, is deployed. The number of states is still 3 and there are still many events per column, enough to fully exploit the computational power of the GPUs. The selection policy tested is only single because using a multiple one in these conditions would bring to an explosion of events created, and because the interesting aspect is only present in the single selection case.

What happens is that without parameters the CPU engine is much faster than the GPU one: in fact finding a compatible event for partial sequences is very easy. This, combined with the single selection policy, makes the CPU job absolutely trivial. Indeed the first element tested will always be valid and the computation for that state will immediately stop, with a valid result. The GPU, on the other hand, will still analyze all the events in the columns without knowing anything on their values; the GT520 indeed needs much more time than the CPU and the GTX460, that can still analyze all the events of the columns in a time comparable to that taken by the CPU. Increasing the number of events would worsen even more the performance of the GPUs, even for the GTX460, while would not affect the CPU performance. The effect is highly hidden already with 1 parameter, that makes the CPU average time much higher, while the time needed by the GPU is lowered. This last effect is caused by the number of complex events created (that increases a lot with less parameters) and to be copied on the host.

4.13 Memory Manager performance

The tests above analyze the response of the engine to different kinds of workload. One important feature of the new engine, on the other hand, is the memory manager that allows memory pagination to save the limited GPU memory to install more rules in the GPU engine, and also to swap memory pages to increment even more the efficiency, allowing the allocation of virtual memory pages on the GPU. The size of a single page can be set in terms of events number at compile time with a specific preprocessor flag. Also the usage of memory pagination can be tweaked as explained in SECTION 3.6.

4.13.1 Page Size

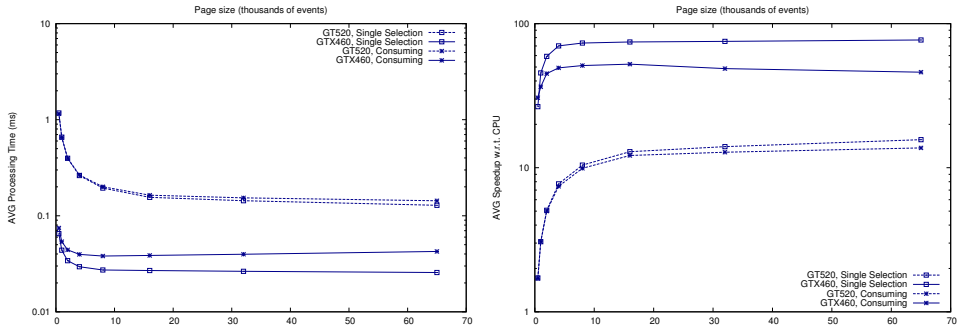


Figure 4.12: Varying Page Size

A lower page size allows the engine to better use the device memory, avoiding the allocation of memory areas that won't be used for the computation. On the other hand a lower size means that more pages will be needed for the same computation, thus also a bigger parameter space and more possible fragmentation slowing down global memory reads. As the graph shows this is not a problem until the size of the page becomes very small, but when the size becomes lower than 8192 performance decreases rapidly. It is also interesting how the usage of a consuming flag interacts with the pagination. Overall the consuming clause worsens the speedup of the GPU, since it introduces more memory copies between the host and the device as well as sequential work to be done by the CPU to find the events to be deleted. Indeed when an event is consumed it must be deleted from the columns. If this is a trivial task on the CPU thanks to the structures based

on pointers, when dealing with the GPU memory the whole page must be copied again after the deletion of an event. Note that it could be possible to simply flag the event as deleted, but this would cause memory fragmentation and wasted threads. In the end smaller pages should help when a consuming clause is used, because the size of memory blocks that must be copied again when a page is invalidated is reduced. This reflects marginally on the overall performance, as it can be seen from graph 4.12, where the difference in the average computation between the consuming and the not consuming rules lowers with smaller pages.

4.13.2 Swapping

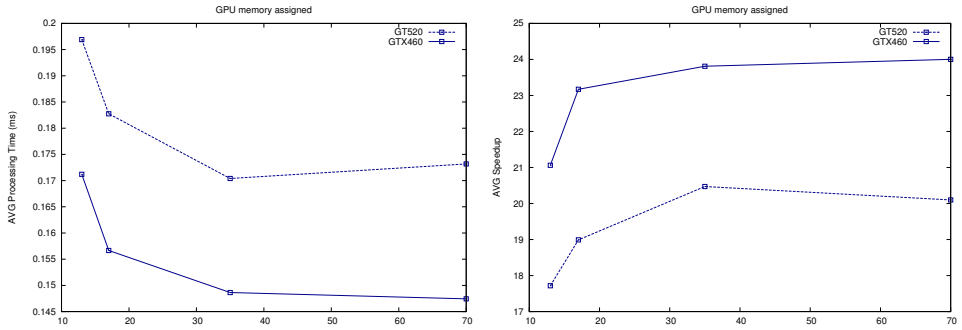


Figure 4.13: Varying Page Swapping Usage

Testing exactly the impact of page swapping on the overall performance was not trivial: there's no direct way to control the swapping of the pages, and it must be intentionally caused combining the effects of the available GPU memory, the host memory multiplier, the number of rules, the size of the pages and the distribution of events.

Luckily enough the distribution of events is, as usual, equal: each event entering the engine has the same probability to be stored in any column created. Setting the pages to be small in relation to the forecast of events that each column will store then is essential to enhance the usage of different pages and thus swapping. The last parameter to be set is GPU memory: keeping all the other settings fixed, lowering the GPU memory until the computation cannot finish is the first step. At that point, increasing the hsot over device memory multiplier allows the computation to finish correctly and the swapping feature to be used.

Thus in the graph the GPU memory used to complete the computation is reported: at each step a greater multiplier was needed, so that in the 70mb test no swapping was used, with 35mb a page could be shared by two different rules, with 17mb by three and with only 13mb four different rules could access the same page.

The results are interesting: page swapping allows to greatly reduce memory consumption without affecting too much the performance.

In the last test memory usage was just about 18% of that of the first one, and the average computation time raised by only 13% on the GT520 to 16% on the GTX460.

Conclusions

Future work There are various enhancements that can be applied to the code that could potentially optimize the performance of GTRex.

In first place the GPUs used for the development and the performance analysis of the project were all based on the Fermi architecture, that implies computation capabilities 2.1.

One of the very interesting features of the new Kepler GPUs, bringing CUDA 3.5 capabilities, is dynamic parallelism. It can be exploited to bypass the limitations of both the approaches proposed for the negations' computation; indeed with this new feature new kernels can be called from the device code, so that a new parallel function that checks all the negations can be launched only if the event meets parameter and timestamp requirements in the previous steps of the computation.

Dynamic parallelism could also reduce the synchronization requirements actually present during the computation of complex events in single selection cases.

Another interesting aspect that can be studied is related to the page swapping management. As seen before, the actual code that chooses a new page to be allocated for a requesting rule is very basic. It chooses simply the first slot available that is not conflicting with another slot already possessed by the rule and not being shared by more rules than the minimum possible. This choice works well with the scripted workloads tested during the project development. On the other hand in different, maybe more realistic, situations there may be better implementations. As example it could be possible to implement an LRU (least recently used) selection policy, that could lead to less memory swapping in particular cases where terminators for some rules come much less frequently than others.

In the end, note that all the GPU activity done in the project relies on the first GPU listed by the NVIDIA driver. This is ok for most configurations where the computer has only 1 GPU. On a PC with many GPUs, though, it may be useful to have the possibility to assign different rules to different GPUs, so that even a greater parallelization and even greater performance could be achieved. On the other hand this should be a trivial modification to the project code, and could be done without too many efforts. It could be enough do instantiate more than one GPUengine (thus also more MemoryManagers) in the execution and tell each one which GPU it should use. Of course then the rules must be assigned proportionally to all the available engine according to the capabilities of the related GPU.

State Of Art One of the most known and used CEP middleware used in the world is Esper [1], an open source java enterprise level product, widely used and mature, adopting a flexible language with a rich syntax and focusing on efficiency and performance. It uses an automata representation of sequences of events similar to that of the AIP algorithm

As shown by Cugola and Margara in their paper [5] the first version of TRex, exploiting the AIP algorithm, was already faster than Esper in all the test cases; in some scenarios the difference was very considerable.

In addition GTRex demonstrated to be very powerful: it brings incredible performance improvements in many workloads without loosing the features and the flexibility of the standard CPU implementation of the TRex engine, with the exception of some compile time variables that must be set for the sizes of different components of the engine.

In conclusion, even if a direct comparison between GTRex and Esper was not conducted, the engine developed in this thesis should outperform substantially Esper in most, if not all, the possible usage scenarios.

Resume and Final Considerations In this thesis I have implemented a new CUDA powered engine that can be easily embedded in the existing TRex2 middleware for Complex Event Processing.

My implementation started from a test project developed by Cugola and Margara, that had a very limited feature set and wasn't compatible with the current version of the middleware.

I added support for various types of aggregates, negations, consuming clause, rules with both single and multiple selection operators as well as the same flexibility in the rule definitions characterizing the TRex standard engine and adapted the data structures and the interface to be compatible with the other components of the engine.

Moreover I designed and developed from scratch the new memory manager that brings pagination and virtualization, allowing at the same time GPUs to handle a big number of rules even using a small amount of memory and to optimize memory transfers with performance improvements.

Finally I optimized the whole engine with the goal of overlapping as much as possible the execution of different jobs on the host CPU and on the GPU.

The new engine demonstrated to be very efficient in terms of performance and even quite flexible, keeping a considerable advantage over the classic CPU engine in a wide variety of use cases.

On the other hand it has still some limitations, that can be reduced with future developments.

In my opinion pairing the current CPU based engine with GTRex would be a great improvement for the TRex middleware: with an appropriate rule dispatcher both the engines could work in pair and in parallel with the rules that best fit their characteristics. As example rules with many single selection operators and an high probability of finding new complex events could be routed to the CPU engine, while in case of multiple selection and large windows of events they could be processed by GTRex.

In this way the performance of the middleware can be substantially enhanced with an affordable expense, since even a GPU costing less than 100 Euros can bring massive improvements in certain situations.

Bibliography

[1] Esper website.

[2] *nVidia CUDA Programming Guide - www.nvidia.com/object/cuda_develop.html.*

[3] www.khronos.org/opencl.

[4] www.nvidia.com/object/cuda_home.html.

[5] Alessandro Margara Gianpaolo Cugola. Complex event processing with t-rex.

[6] Alessandro Margara Gianpaolo Cugola. Low latency complex event processing on parallel hardware.

[7] Alessandro Margara Gianpaolo Cugola. Tesla: A formally defined event specification language.