High Performance Content-Based Matching Using GPUs

Alessandro Margara
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy
margara@elet.polimi.it

Gianpaolo Cugola
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy
cugola@elet.polimi.it

ABSTRACT

Matching incoming event notifications against received subscriptions is a fundamental part of every publish-subscribe infrastructure. In the case of content-based systems this is a fairly complex and time consuming task, whose performance impacts that of the entire system. In the past, several algorithms have been proposed for efficient content-based event matching. While they differ in most aspects, they have in common the fact of being conceived to run on conventional, sequential hardware. On the other hand, modern Graphical Processing Units (GPUs) offer off-the-shelf, highly parallel hardware, at a reasonable cost. Unfortunately, GPUs introduce a totally new model of computation, which requires algorithms to be fully re-designed. In this paper, we describe a new content-based matching algorithm designed to run efficiently on CUDA, a widespread architecture for general purpose programming on GPUs. A detailed comparison with SFF, the matching algorithm of Siena, known for its efficiency, demonstrates how the use of GPUs can bring impressive speedups in content-based matching. At the same time, this analysis demonstrates the peculiar aspects of CUDA programming that mostly impact performance.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems, Performance evaluation (efficiency and effectiveness)*

General Terms

Algorithms, Measurement, Performance

1. INTRODUCTION

Distributed event-based applications [24] are becoming more and more popular. They usually leverage a *publish-subscribe* infrastructure, which enables distributed components to *subscribe* to the *event notifications* (or "events", for simplicity) they are interested to receive, and to *publish* those they want to spread around.

The core functionality realized by a publish-subscribe infrastructure is *matching* (sometimes also referred as "forwarding"), i.e., the action of filtering each incoming event notification e against the received subscriptions to decide the components interested in e. This is a non trivial activity, especially for a *content-based* publish-subscribe infrastructure, in which subscriptions filter event notifications based on their content [13]. In such systems, the matching component may easily become the bottleneck of the publish-subscribe infrastructure. On the other hand, in several event-based applications, the performance of the publish-subscribe infrastructure may be a key factor. As an example, in financial applications for high-frequency trading [17], a faster processing of incoming event notifications may produce a significant advantage over competitors. Similarly, in intrusion detection systems [22], the ability to timely process the huge number of event notifications that results from observing the operation of a large network is fundamental to detect possible attacks, reacting to them before they could compromise the network.

This aspect has been clearly identified in the past and several algorithms have been proposed for efficient content-based event matching [14, 10, 2, 5]. While these algorithms differ in most aspects, they have one fundamental thing in common: they were designed to run on conventional, sequential hardware. This is reasonable, since standard computers are equipped with sequential CPUs, or, at most, with multicore CPUs, which allow a limited number of processes to be executed in parallel. On the other hand, the importance of graphics in most application domains pushed industry into producing ad-hoc Graphical Processing Units (GPUs) to relieve the main CPU from the (complex) calculations required for graphics.

What is important here is that this hardware is strongly parallel and may operate in a way (largely) independent from the main CPU. A modern but medium level GPU, like those equipping most computers today, allows hundreds of operations to be performed in parallel, leaving the CPU free to execute other jobs. Moreover, several vendors have recently started offering toolkits to leverage the power of GPUs for general purpose programming. Unfortunately, they introduce a totally new model of computation, which requires algorithms to be fully re-designed. In particular, modern GPUs offer hundreds of processing cores, but they can be used simultaneously only to perform data parallel

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS'11, July 11–15, 2011, New York, New York, USA.
Copyright 2011 ACM 978-1-4503-0423-8/11/07 ...$10.00.

computations. Moreover, GPUs usually have no direct access to the main memory and they do not offer hardware managed caches; two aspects that make memory management a critical factor to be carefully considered.

In this paper we present *CCM*[1] – *CUDA Content-based Matcher* – a new content-based matching algorithm designed to run efficiently on GPUs that implement the CUDA architecture [11], and we evaluate its performance, comparing it against SFF [10], the matching component of Siena [7], usually considered among the most efficient solution for content-based matching. Our study demonstrates how the use of GPUs can bring impressive speedups in content-based matching, leaving the main CPU free to focus on the remaining aspects of publish-subscribe, like managing the networking connections with clients and (de)serializing of data. By carefully analyzing how CCM performs under different workloads, we also identify the peculiar aspects of GPU programming that mostly impact performance.

The remainder of the paper is organized as follow: Section 2 offers an overview of the CUDA architecture and programming model, focusing on the aspects more relevant for our purpose. Section 3 introduces the data model and the terminology we use in the rest of the paper. Section 4 describes CCM and how we implemented it on CUDA. The performance of CCM in comparison with SFF is discussed in Section 5, while Section 6 presents related work. Finally, Section 7 provides some conclusive remarks and describes future work.

2. GPU PROGRAMMING WITH CUDA

CUDA is a general purpose parallel computing architecture introduced by Nvidia in November 2006. It offers a new parallel programming model and instruction set for general purpose programming on GPUs.

Since parallel programming is a complex task, industry is currently devoting much effort in trying to simplify it. The most promising result in this area is OpenCL [29], a library that supports heterogeneous platforms, including several multicore CPUs and GPUs. However, the idea of abstracting very different architectures under the same API has a negative impact on performance: different architectures are better at different tasks and a common API cannot hide that. For this reason, although OpenCL supports the CUDA architecture, we decided to implement our algorithms in CUDA C [28], a dialect of C explicitly devoted to program GPUs that implement the CUDA architecture.

In this section we introduce the main concepts of the CUDA programming model and CUDA C, and we briefly present some aspects of the hardware implementation that play a primary role when it comes to optimize algorithms for high performance on the CUDA architecture.

2.1 Programming Model

The CUDA programming model is intended to help developers in writing software that leverages the increasing number of processor cores offered by modern GPUs. At its foundation are three key abstractions:

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

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

Barrier synchronization. Thread blocks are required to execute independently, while threads within a block can cooperate by sharing data through shared memory and by synchronizing their execution to coordinate memory access. In particular, developers may specify synchronization points that act as barriers at which all threads in the block must wait before any is allowed to proceed [27, 3].

The CUDA programming model assumes that CUDA threads execute on a physically separate *device* (the GPU), which operates as a coprocessor to a *host* (the CPU) running a C/C++ program. To start a new computation on a CUDA device, the programmer has to define and call a function, called *kernel*, which defines a single flow of execution. When a kernel k is called, the programmer specifies the number of threads per block and the number of blocks that must execute it. Inside the kernel it is possible to access two special variables provided by the CUDA runtime: the *threadId* and the *blockId*, which together allow to uniquely identify each thread among those executing the kernel. Conditional statements involving these variables are the only way for a programmer to differentiate the execution flows of different threads.

The CUDA programming model assumes that both the host and the device maintain their own separate memory spaces. Therefore, before invoking a kernel, it is necessary to explicitly allocate memory on the device and to copy there the information needed to execute the kernel. Similarly, when a kernel execution completes, it is necessary to copy results back to the host memory and to deallocate the device memory.

2.2 Hardware Implementation

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

Each SM creates, manages, schedules, and executes threads in groups of parallel threads called *warps*. Individual threads composing a warp start together but they have their own instruction pointer and local state and are therefore free to branch and execute independently. When a SM is given one or more blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution.

All threads in a warp execute one common instruction at a time. This introduces an issue that must be carefully taken into account when designing a kernel: full efficiency

[1] CCM is currently available for download from http://cudamatcher.sf.net

is realized only when all the threads in a warp agree on their execution path. If threads in the same warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

Inside a single SM, instructions are pipelined but, differently from modern CPU cores, they are executed in order, without branch prediction or speculative execution. To maximize the utilization of its computational units, each SM is able to maintain the execution context of several warps on-chip, so that switching from one execution context to another has no cost. At each instruction issue time, a warp scheduler selects a warp that has threads ready to execute (not waiting on a synchronization barrier or for data from the global memory) and issues the next instruction to them.

To give a more precise idea of the capabilities of a modern GPU supporting CUDA, we provide some details of the Nvidia GTX 460 [18] card we used for our tests. The GTX 460 includes 7 SMs, which can handle up to 48 warps of 32 threads each (for a maximum of 1536 threads). Each block may access a maximum amount of 48KB of shared memory implemented directly on-chip within each SM. Furthermore, the GTX 460 offers 1GB of GDDR5 memory as global memory. This information must be carefully taken into account when programming a kernel: shared memory must be used as much as possible, since it may hide the higher latency introduced by access to global memory. However, shared memory has a limited size, which significantly impacts the design of algorithms by constraining the amount of data that can be shared by the threads in each block.

3. EVENTS AND PREDICATES

Before going into the details of our matching algorithm we introduce the models of events and subscriptions we assume in this paper. To be as general as possible we assume a very common model among event-based systems [10].

For our purposes, an *event notification*, or simply *event*, is represented as a set of *attributes*, i.e., $\langle name, value\rangle$ pairs. Values are typed: in our algorithm we consider both numerical values and strings. As an example, $e_1 = [\langle area, \text{“}area1\text{”}\rangle, \langle temp, 25\rangle, \langle wind, 15\rangle]$ is an event that an environmental monitoring component could publish to notify the rest of the system about the current temperature and wind speed in the area it monitors.

The interests of components are modeled through *predicates*, each being a disjunction of *filters*, which, in turn, are conjunctions of elementary *constraints* on the values of single attributes. As an example, $f_1 = (area = \text{“}area1\text{”} \wedge temp > 30)$ is a filter composed of two constraints, while $p_1 = [(area = \text{“}area1\text{”} \wedge temp > 30) \vee (area = \text{“}area2\text{”} \wedge wind > 20)]$ is a predicate composed of two filters.

A filter f *matches* an event notification e if all constraints in f are satisfied by the attributes of e. Similarly, a predicate matches e if at least one of its filters matches e. In the examples above, predicate p_1 matches event e_1.

Given these definitions, the problem of content-based matching can be stated in the most general terms as follow: given a set of *interfaces*, each one exposing a predicate, and an event e, find the set of interfaces e should be delivered to, i.e., the set of interfaces that expose a predicate matching e.

In a centralized publish-subscribe infrastructure, it is the *dispatcher* which implements this function, by collecting predicates that express the interests of subscribers (each connected to a different “interface”) and forwarding incoming event notifications on the basis of such interests. In a distributed publish-subscribe infrastructure the dispatcher is composed of several *brokers*, each one implementing the content-based matching function above to forward events to its neighbors (other brokers or the subscribers directly connected to it).

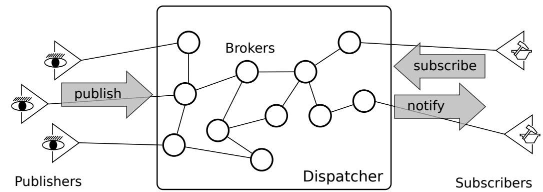


Figure 1: A typical publish-subscribe infrastructure

4. CUDA CONTENT-BASED MATCHER

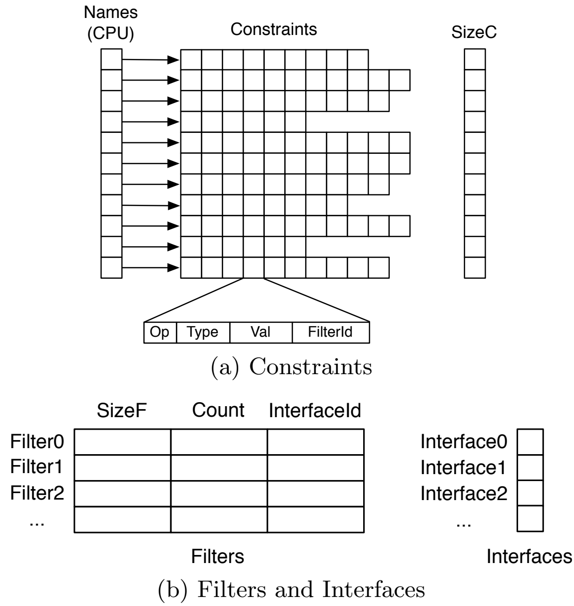
The *CUDA Content-based Matcher (CCM)* algorithm we designed is composed of two phases: a *constraint selection* phase and a *constraint evaluation and counting* phase. When an event enters the engine, the first phase is used to select, for each attribute a of the event, all the constraints having the same name as a. These constraints are evaluated in the second phase, using the value of a. In particular, we keep a counter of satisfied constraints for each filter stored by the system: when a constraint c is satisfied, we increase the counter associated to the filter c belongs to. A filter matches an event when all its constraints are satisfied, i.e., when the associated counter is equal to the number of its constraints. If a filter matches an event so does the predicate it belongs to. Accordingly, the event can be forwarded to the interface exposing it.

As already mentioned in Section 2, using a GPU that implements the CUDA architecture may provide significant computational speedups, but requires the developer to carefully design the algorithm and the data structures to maximize parallelism, exploit programmable caches, and minimize the amount of information that has to be transferred from the main memory to the GPU memory and back. The rest of this section focuses on these aspects, while describing the design and implementation of CCM in details.

4.1 Data Structures

Figure 2 shows the data structures we create and use during processing. Almost all of them are permanently stored into the GPU memory, to minimize the need for CPU-to-GPU communication during event processing.

In Figure 2a we see the data structures containing information about constraints. The GPU stores table `Constraints`, which groups existing constraints into multiple rows, one for each name. Each element of such table stores information about a single constraint: in particular its operator (`Op`), its type (`Type`), its value (`Val`), and an identifier of the filter it belongs to (`FilterId`). Moreover, since different rows may include a different number of constraints, the GPU also stores a vector `SizeC` with the actual size of each row. Finally, table `Constraints` is coupled with map `Names` ($\langle name, rowId\rangle$), stored by the CPU, which associates each attribute name with the corresponding row in `Constraints`.



(a) Constraints

(b) Filters and Interfaces

Figure 2: Data structures

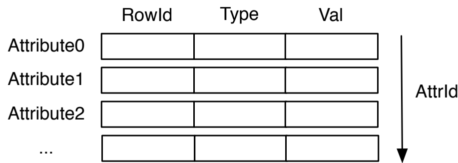


Figure 3: Input data

Figure 2b shows the data structures containing information about filters and interfaces. Each row of table `Filters` represents a different filter and stores its size (`SizeF`), i.e., the number of constraints it is composed of, the number of currently satisfied constraints (`Count`), and the interface it belongs to (`InterfaceId`), i.e., the interface whose associated predicate includes this filter. As we will discuss in more details later, `Count` is set to zero before processing an event, and it is updated by different threads in parallel during processing.

Finally, `Interfaces` is a vector of bytes, with each position representing a different interface. As for `Count`, it is set to zero before processing an event e and it is updated during processing. A value of one in position x means that the event under processing must be forwarded through interface x. Accordingly, the content of vector `Interfaces` represents the result of the matching process, which has to be copied from the GPU memory to the CPU memory when the event under consideration has been processed.

4.2 Parallel Evaluation of Constraints

When a new event e enters the engine, the CPU uses the information included in e together with map `Names` to build table `Input` in Figure 3. It stores, for each attribute a in e, the id of the row in `Constraints` associated to the name of a (`RowId`), the type of a (`Type`), and its value (`Val`). This information is subsequently transferred to the GPU memory to match e against relevant constraints. Notice that, in principle, we could avoid this phase, transferring the whole content of e to the GPU and letting it find the relevant constraints (those having the same names of e's attributes), using parallel threads. However, some preliminary experiments we performed show that the computational effort required to select constraints on the CPU using a STL map is negligible (less than 1% of the overall processing time); on the other hand, this selection can potentially filter out a huge number of constraints, thus making the subsequent evaluation on the GPU much faster.

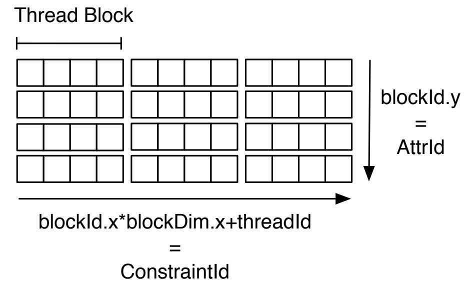


Figure 4: Organization of blocks and threads

After building table `Input` and transferring it to the GPU memory, the CPU launches a new kernel, with thousands of threads working in parallel, each one evaluating a single attribute a of e against a single constraint among those relevant for a in table `Constraints`.

As we mentioned in Section 2, at kernel launch time the developer must specify the number of blocks executing the kernel and the number of threads composing each block. Both numbers can be in one, two, or three dimensions. Figure 4 shows our organization of threads and blocks. It shows an example in which each block is composed of only 4 threads, but in real cases 256 or 512 threads per block are common choices. We organize all threads inside a block over a single dimension (x axis), whereas blocks are organized in two dimensions. The y axis is mapped to event attributes, i.e., to rows of table `Input` in Figure 3. The x axis, instead, is mapped to set of constraints. Indeed, since the number of constraints with the same name of a given attribute may exceed the maximum number of threads per block, we allocate multiple blocks along the x axis.

All threads part of the same kernel execute the same function, whose pseudo-code is presented in Algorithm 1. The only difference between threads are the values of the blockId and threadId variables, initialized by the CUDA runtime.

Using these values, each thread determines its x and y coordinates in the bi-dimensional space presented in Figure 4. More specifically, the value of y is directly given by the y value of blockId, while the value of x is computed as blockId.x·blockDim.x+threadId, where blockDim.x is the x size of each block.

At this point each thread reads the data it requires from tables `Input` and `Constraints`. Since all threads in the same block share the same attribute, i.e., the same element in table `Input`, we copy such element from the global memory to the block shared memory once for all. More specifically, the command at line 3 defines a variable shInput in shared memory, which the first thread of each block (the one having threadId equal to 0) sets to the appropriate value taken from table `Input`. All other threads wait until the copy is

Algorithm 1 Constraint Evaluation Kernel

```
1: x = blockId.x·blockDim.x+threadId
2: y = blockId.y
3: __shared__ shInput
4: if threadId==0 then
5:    shInput = input[y]
6: end if
7: __syncthreads()
8: rowId = shInput.RowId
9: if x ≥ SizeC[rowId] then
10:    return
11: end if
12: constraint = Constraints[x][rowId]
13: type = shInput.Type
14: val = shInput.Val
15: if ! sat(constraint, val, type) then
16:    return
17: end if
18: filterId = constraint.FilterId
19: count = atomicInc(Filters[filterId].Count)
20: if count+1==Filters[filterId].SizeF then
21:    interfaceId = Filters[filterId].InterfaceId
22:    Interfaces[interfaceId] = 1
23: end if
```

finished by invoking the __syncthreads() command in line 7. Our experiments show that this optimization w.r.t. the straightforward approach of letting each thread access table `Input` (which is stored into the slower global memory) separately, increases performance by 2-3%.

Once this initial phase has completed, each thread uses the `RowId` information copied into the shInput structure to determine the row of table `Constraints` it has to process (each thread will process a different element of such row). Notice that different rows may have different lengths: we instantiate the number of blocks (and consequently the number of threads) to cover the longest among them. Accordingly, in most cases we have too many threads. We check this possibility at line 9, immediately stopping unrequired threads. This is a common practice in GPU programming, e.g., see [32]. We will analyze its implication on performance in Section 5.

In line 12 each thread reads the constraint it has to process from table `Constraints` in global memory to the thread's local memory (i.e., hardware registers, if they are large enough), thus making future accesses faster. Also notice that our organization of memory allows threads having contiguous identifiers to access contiguous regions of the global memory. This is particularly important when designing an algorithm for CUDA, since it allows the hardware to combine different read/write operations into a reduced number of memory-wide accesses, thus increasing performance.

In lines 13 and 14 each thread reads the type and value of the attribute it has to process and uses them to evaluate the constraint it is responsible for (in line 15[2]). If the constraint is not satisfied the thread immediately returns, otherwise it extracts the identifier of the filter the constraint belongs to and updates the value of field `Count` in table `Filters`. Notice that different threads may try to update the same counter concurrently. To avoid clashes, we exploit a special atomicInc operation offered by CUDA, which atomically reads the value of a 32bit integer from the global memory, increases it, and returns the old value.

In line 20 each thread checks whether the filter is satisfied, i.e., if the current number of satisfied constraints (old count plus one) equals the number of constraints in the filter. Notice that at most one thread for each filter can positively evaluate this condition. If the filter is satisfied, the thread extracts the identifier of the interface the filter belongs to and sets the corresponding position in vector `Interfaces` to 1. It is possible that multiple threads access the same position of `Interfaces` concurrently; however, since they are all writing the same value, no conflict arises.

As we mentioned in Section 2, CUDA provides better performance when threads belonging to the same warp inside a block follow the same execution path. After table `Input` is read in line 5 and all unrequired threads have been stopped in line 10, Algorithm 1 has two conditional branches where the execution path of different threads may potentially diverge. The first one, in line 15, evaluates a single attribute against a constraint, while the second one, in line 20, checks whether all the constraints in a filter have been satisfied before setting the relevant interface to 1. The threads that follow the positive branch in line 20 are those that process the last matching constraint of a filter. Unfortunately, we cannot control the warps these threads belong to, since this depends from the content of the event under evaluation and from the scheduling of threads. Accordingly, there is nothing we can do to force threads on the same warp to follow the same branch.

On the contrary, we can increase the probability of following the same execution path within function sat in line 15 by grouping constraints in table `Constraints` according to their type, operator, and value. This way we increase the chance that threads in the same warp, having contiguous identifiers, process constraints with the same type and operator, thus following the same execution path into sat. Our experiments, however, show that such type of grouping of constraints provides only a very marginal performance improvement and only under specific conditions. On the other hand, it makes creation of data structures (i.e., table `Constraints` that needs be ordered) much slower. We will come back to this issue in Section 5.

4.3 Reducing Memory Transfers

To correctly process an event, we first need to reset field `Count` associated with each filter and vector `Interfaces`. In a preliminary implementation, we let the CPU perform these operations through the cudaMemset command, which allows to set all bytes of a memory region on the GPU to a common value (0 in our case). Although optimized, the cudaMemset command introduces a relevant overhead, since it involves a communication between the CPU and the GPU over the (slow) PCI-E bus.

On the other hand, since the CUDA architecture executes blocks in non deterministic order and does not allow inter-blocks communication, we could not straightforwardly reset data structures within our kernel. Indeed, assuming a subset of threads is used to reset data immediately after launch, we cannot know in advance whether they are executed immediately, or if other threads (in other blocks) get executed before.

[2] We omit for simplicity the pseudo code of the sat function that checks whether a constraint is satisfied by an attribute.

To overcome this problem, we decided to duplicate data structures `Count` and `Interfaces`. When processing an event we use one copy of these data structures and we reset the other using different threads in parallel, while we reverse the role of the two copies at the next event. Notice that in doing so we take care of writing adjacent memory blocks from contiguous threads: this minimizes the overhead introduced by this reset operation. Through this trick we avoid using the cudaMemset operation and do everything within the single kernel that also executes the matching algorithm. This way the interactions between CPU and GPU memory are reduced to the minimum: indeed, we only need to copy the event content to the GPU (filling table `Input`) to start processing and to copy vector `Interfaces` when computation is finished. In Section 5 we will analyze the cost of these operations in details, measuring the benefits of this approach.

5. EVALUATION

Our evaluation had three main goals: first of all, we wanted to compare CCM with a state of the art algorithm running on the CPU, to understand the real benefits of CUDA. Second, since the performance of a matching algorithm are influenced by a large number of parameters, we wanted to explore the parameter space as broadly as possible, to isolate the aspects that make the use of the GPU more profitable. Third, we wanted to profile our code to better understand the aspects of CUDA that mostly impact performance.

To fulfill the first goal, we decided to compare CCM with SFF [10] version 1.9.4, the matching algorithm used inside the Siena event notification middleware [7]. Similarly to CCM, SFF is a counting algorithm, which runs over the attributes of the event under consideration, counting the constraints they satisfy until one or more filters have been entirely matched. When a filter f is matched, the algorithm marks the related interface, purges all the constraints and filters exposed by that interface, and continues until all interfaces are marked or all attributes have been processed. The set of marked interfaces represents the output of SFF. To maximize performance under a sequential hardware like a traditional CPU, SFF builds a complex, strongly indexed data structure, which puts together the predicates (decomposed into their constituent constraints) received by subscribers. A smart use of hashing functions and pruning techniques, allows SFF to obtain state-of-the-art performance under a very broad range of workloads.

To compare CCM against SFF we defined a default scenario, whose parameters are listed in Table 1, and used it as a starting point to build a number of different experiments, by changing such parameters one by one and measuring how this impacts the performance of the two algorithms. In our tests we focus on the time to process a single event (i.e., latency). In particular, we let each algorithm process 1000 events, one by one, and we calculate the average processing time. To avoid any bias, we repeat each test 10 times, using different seeds to randomly generate subscriptions and events, and we plot the average value measured[3].

Tests were executed on a AMD Phenom II machine, with 6 cores running at 2.8GHz, and 8GB of DDR3 Ram. The GPU was a Nvidia GTX 460 with 1GB of GDDR5 Ram. We used the CUDA runtime 3.1 for 64 bit Linux. It is worth mentioning that nowadays the GTX 460 is a low level graphic card with a limited cost (less than 160$ in March 2011). Nvidia currently offers much better graphic cards, and also cards (TESLA GPUs) explicitly conceived for high performance computing, with a higher number of cores, and up to 6GB of memory, having higher bandwidth and speed.

[3]The 95% confidence interval of this average is always below 1% of the measured value, so we omit it from the plots.

Parameter	Value
Number of events	1000
Attributes per event, min-max	3-5
Number of interfaces	10
Constraints per filter, min-max	3-5
Filters per interface, min-max	22500-27500
Number of names	100
Distribution of names	Uniform
Numerical/string constraints (%)	50/50
Numerical operators	=(25%), ! =(25%), >(25%), <(25%)
String operators	=(25%), ! =(25%), $subString$(25%), $prefix$(25%),
Number of values	100

Table 1: Parameters in the default scenario

	CCM	SFF
Processing time in the default scenario	0.144ms	1.035ms
Subscriptions deployment time in the default scenario	527.5ms	992.28ms
Data structure size in the default scenario	33.9MB GPU Ram	42.4MB CPU Ram
Processing time with a Zipf distribution for names	2.06ms	14.68ms

Table 2: Analysis of CCM

Default Scenario. Table 2 (first row) shows the processing times measured by the two algorithms in the default scenario. This is a relatively easy-to-manage scenario. It includes one million constraints on the average, which is not a big number for large scale applications. Under this load, SFF exhibits a processing time that is slightly above 1ms. On the other hand, our CCM algorithm requires less than 0.15ms to process an event (with a potential throughput of more than 6000 events/s), providing a speedup of 7.19x.

Deployment of Subscriptions. Besides measuring processing time, we were also interested in studying the time required to create the data structures used during event evaluation. Indeed, they need to be generated at run-time, when new subscriptions are deployed on the engine. Even if it is common to assume that the number of publishing largely exceeds the number of subscribing/unsubscribing in any event-based application, this time has also to be considered when measuring the performance of a matching algorithm.

Table 2 (second row) shows the average time required by SFF and CCM to create their data structures. CCM requires 527.5ms: this time includes allocating memory on the GPU, building the data structures (part on the CPU and part on the GPU), and copying them from the CPU to the GPU memory. On the contrary, SFF has only to build the data structures in main memory: however, to attain its perfor-

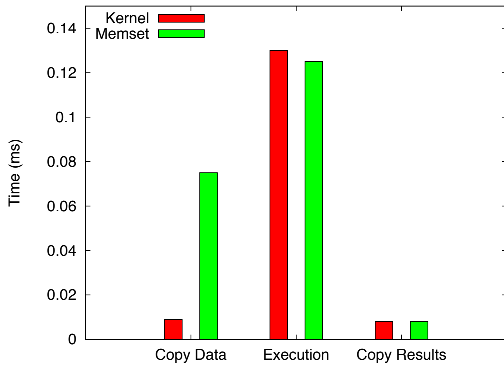


Figure 5: An Analysis of CCM Processing Times

mance under sequential hardware, SFF needs more complex structures, which need more time to be built (around 1s). This is true in all scenarios, where the advantage of CCM w.r.t. SFF was at least as large as the one we reported here (i.e., twice as fast). For space reasons we do not report the exact times we measured in the other scenarios, while we prefer to focus on the event matching time.

The complexity of the data structures used by SFF also reflects on their size. As shown in Table 2 (third row), CCM requires less memory to store data structure: the default scenario occupies 33.9MB of the GPU memory and the maximum occupancy we measured in our tests was below 200MB. Since GPUs nowadays have at least 1GB of Ram, contrary to what happens in many cases, GPU memory occupancy is not a problem for CCM. It is worth mentioning, however, that beside the data structures allocated for processing, the use of the CUDA toolkit introduces a fixed overhead of about 140MB of main (CPU) memory.

An Analysis of CCM Processing Times. Figure 5 analyzes the cost of the different operations performed by CCM during the matching process in the default scenario. In particular, it splits processing time into three parts: the time required to copy data from the CPU to the GPU memory (which also includes the time needed to build table `Input` matching STL map `Names` on the CPU), the time required to execute the kernel, and the time used to copy results back to the main memory. We compare two different versions of our algorithm: as described in Section 4.3, `Memset` resets field `Count` and vector `Interfaces` invoking the cudaMemset command twice, before kernel starts; on the contrary, `Kernel` resets data structures directly inside the kernel code (this is the version we will use in the remainder of this section).

First of all, we observe that in both cases the cost of executing the kernel dominates the others. If we compare the two versions of CCM, we notice how the kernel's execution time is higher when data structures are reset inside the kernel but by a very low margin (moving from 0.125ms to 0.13ms). On the other hand, the cudaMemset operation is quite inefficient and by not invoking it we strongly reduce the time to execute the first step of the algorithm (from 0.075ms to 0.009ms), thus making the `Kernel` version about 30% faster. In the `Kernel` version, which we will use in the remainder of this section, the cost for moving data back and forth over the (slow) PCI-E bus represents about 8% of the overall matching time, meaning that we are exploiting the computational power of the GPU while introducing a relatively small overhead.

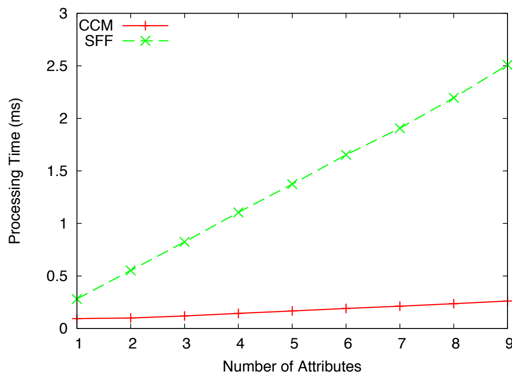
Distribution of Names. As we said in Section 4, our algorithm creates and launches a number of threads that depends from the size of the longest row in `Constraints` among those selected by the names appearing in the incoming event. As we observed in Section 4, in presence of rows with very different sizes, the number of unrequired threads may be relevant. To investigate the impact of this aspect, we changed the distribution of names for both constraints and attributes, moving from the uniform distribution adopted in the default scenario to a Zipf distribution.

Table 2 (fourth row) shows the results we obtained. First of all we notice how both algorithms significantly increase their matching time w.r.t. the default scenario. Indeed, both algorithms use names to reduce the number of constraints to process, and this pre-filtering becomes less effective with a Zipf distribution. What is most important for us, is that the speedup of CCM against SFF remains unchanged. This means that launching a higher number of unrequired threads does not influence the advantage of CCM over SFF.

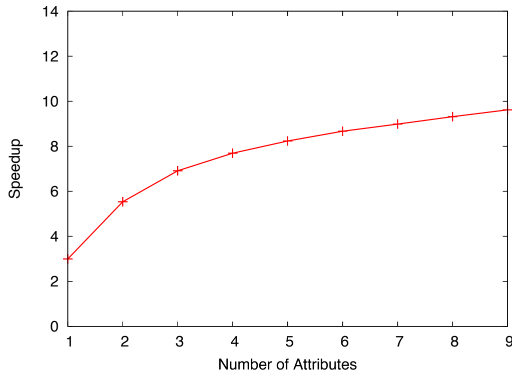
As we mentioned in Section 4, during our experiments we investigated the benefits of ordering constraints according to their type, operator, and value, thus increasing the probability that threads of the same warp follow a common execution path inside the function sat. In the general case, this approach does not significantly decrease the processing time of events, while it has a very negative impact on the cost of deploying subscriptions. Accordingly, we decided not using it. However, when using a Zipf distribution, a large number of constraints, with their types, operators, and values, share a very limited set of names. In this particular case ordering constraints introduces a visible speedup moving the overall processing time from 2.06ms to 1.92ms. We can conclude that the idea of ordering constraints is worth in all those cases when the number of constraints with the same name grows above a few hundred thousands (in the Zipf case we have 500.000 constraints with the most common name, on average)

Number of Attributes. Figure 6 shows how performance changes with the size of events (i.e., the average number of attributes). In particular, Figure 6a shows the processing time of the two algorithms. Both of them exhibit higher matching times with a higher number of attributes. Indeed, increasing the number of attributes in the incoming events also increases the work that need to be performed, since each of them has to be compared with the stored constraints. However, SFF performs all these evaluations sequentially, while CCM launches several blocks of threads to perform them in parallel. This explains why the processing time increases faster in SFF, making the advantage of CCM larger with a high number of attributes. This is evident if we consider the speedup, plotted in Figure 6b: with 9 attributes per event, CCM is ten times faster than SFF.

Number of Constraints per Filter. Figure 7 shows how performance changes with the average number of constraints in each filter. Increasing such number, while keeping a fixed number of filters, increases the overall number of constraints deployed in the engine, and thus the complexity of the matching process. This is demonstrated by Figure 7a. As in the previous case, the possibility to process constraints in parallel advantages our algorithm: both CCM and SFF

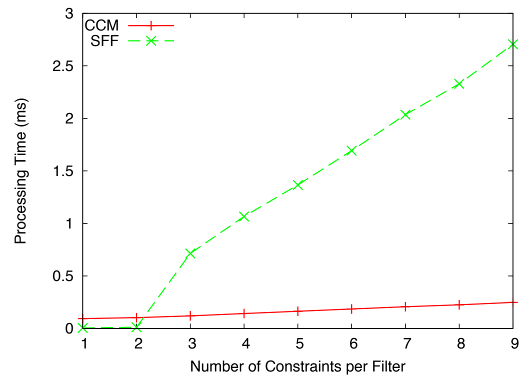


(a) Event Processing Time

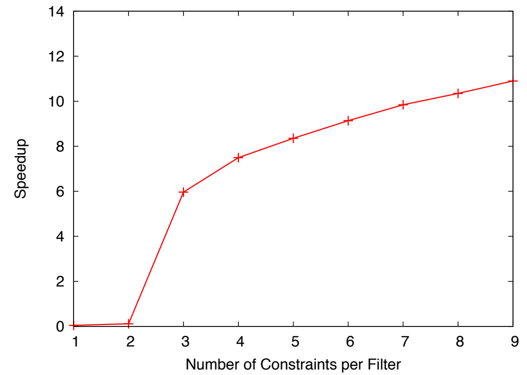


(b) Speedup

Figure 6: Number of Attributes



(a) Event Processing Time



(b) Speedup

Figure 7: Number of Constraints per Filter

show a linear trend in processing time, but CCM times grow much slower than those of SFF. The speedup (see Figure 7b) overcomes 10x with 9 constraints per filter.

Notice that with one or two constraints per filter SFF performs better than CCM. Indeed, a lower number of constraints per filter results in a greater chance to match filters. With one or two constraints per filter we are in a very special (and unrealistic) case in which the chance to find a matching filter for a given interface is very high, such that at the end all events are relevant for all interfaces. The pruning techniques of SFF work at their best in this case, while CCM always process all constraints, albeit in parallel.

Number of Filters per Interface. Figure 8 shows how performance changes with the number of filters defined for each interface. As in the scenario above, increasing such number also increases the overall number of constraints, and thus the complexity of matching. Accordingly both algorithms show higher processing times (see Figure 8a) with CCM suffering less when the scenario becomes more complex. The speedup overcomes 13x with 250000 filters per interface, i.e., 10 millions of constraints (Figure 8b). Notice how the growth of the speedup curve decreases with the number of filters: indeed, a large number of filters increases the number of matching interfaces, allowing SFF to frequently exploit the pruning optimizations described above.

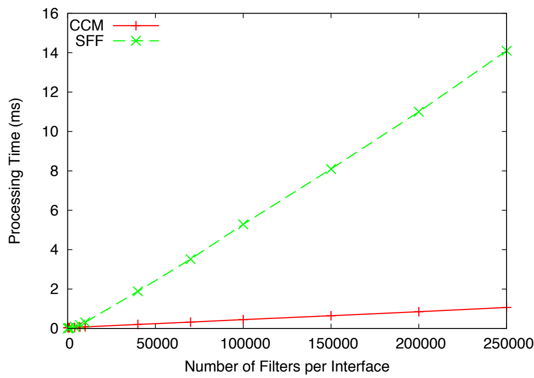
Also observe how a small number of filters favors SFF, which with less than 1000 filters performs better than CCM. Indeed, the latter pays a fixed fee to copy data from/to the GPU memory and to launch the kernel. In a situation in which the computational complexity is very low, this fee accounts for a large fraction of the total cost. On the other hand, under such circumstances the matching is very fast, with both algorithm registering an average processing time below 0.05ms, which makes the advantages of SFF less relevant in practice.

Number of Interfaces. Another important aspect that significantly influences the behavior of a content-based matching algorithm is the number of interfaces. In Figure 9 we analyze its impact on SFF and CCM, moving from 10 to 100 interfaces. Notice that 100 interfaces can be a common value in real applications, in which several clients are served by a common event dispatcher that performs the matching process for all of them.

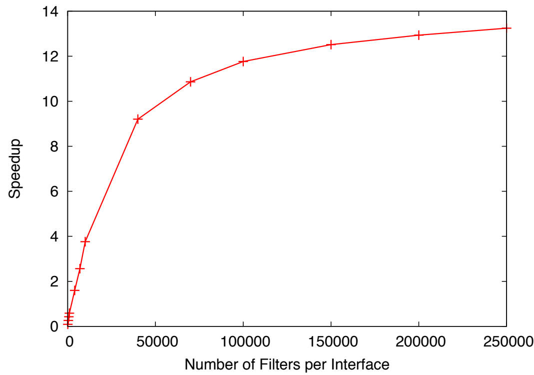
As in the previous experiments, increasing the number of interfaces also increases the number of constraints, and thus the complexity of matching. Accordingly, both algorithms show higher processing times as the number of interfaces grows (Figure 9a). Also in this case, the speedup of CCM increases with the complexity of processing (Figure 9b), moving from 7x to more than 13x.

Number of Names. Both SFF and CCM use the attribute names in the incoming event to perform a preliminary selection of the constraints to evaluate. Accordingly, the total number of possible names used inside constraints and attributes represent a key performance indicator.

Figure 10a shows how the processing time of the two algorithms changes with the number of names for attributes and

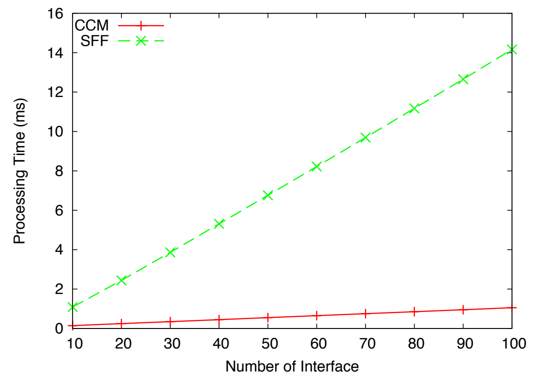


(a) Event Processing Time

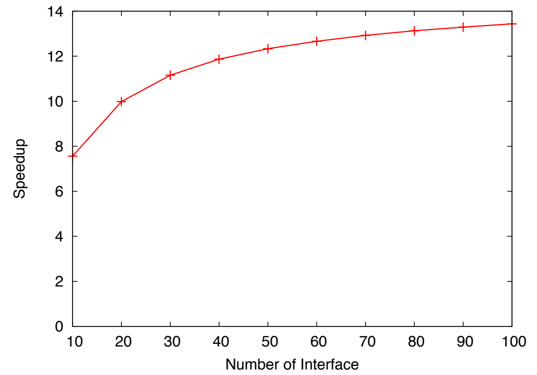


(b) Speedup

Figure 8: Number of Filters per Interface



(a) Event Processing Time



(b) Speedup

Figure 9: Number of Interfaces

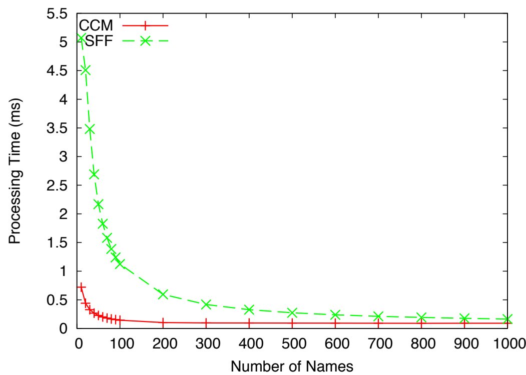
constraints. Notice that increasing the number of names allows the "constraint selection" phase (performed on the CPU even by CCM – see Section 4), to discard a higher number of constraints. Accordingly, the cost of the "constraint evaluation and counting" phase (the more expensive in terms of computation and also the one that CCM performs on the GPU) decreases. This is demonstrated by Figure 10a where both algorithms perform better when the number of names increases, especially when moving from 10 to 100 names. After this threshold times tend to stabilize. This fact can be explained by observing that over a certain number of names the "constraint evaluation and counting" phase becomes so simple that the processing time does not decrease much.

If we look at the speedup (Figure 10b) we observe two different regions. When moving from 10 to 30 names the speedup of CCM increases. Indeed, with few names the probability for a filter to match an event is very high and this allows SFF to leverage its pruning techniques. This benefit vanishes when moving from 10 to 30 names. After 30 names the speedup decreases. Indeed, as we already noticed, increasing the number of names drops the complexity of the "constraint evaluation and counting" phase, which CCM performs in parallel on the GPU. Moreover, this phase has a minimal fixed cost for CCM (the cost of copying data from/to the GPU memory and launching the kernel).

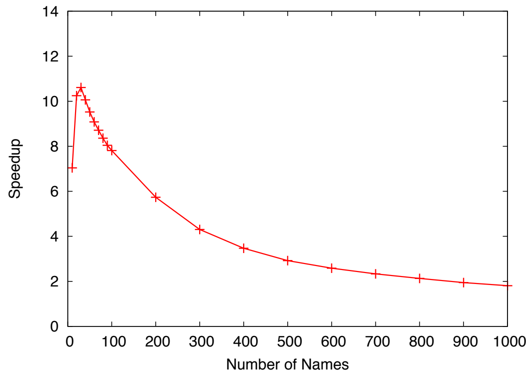
Finally, we observe that even with a high number of names (1000) CCM outperforms SFF, with a 2x speedup on our reference hardware. This result is obtained with a uniform distribution of names. A Zipf distribution, considered more representative of several real-case scenarios [15], would strongly favor CCM, as shown in Table 2 (fourth row). Indeed, in such scenario, most constraints would refer to a few attribute names, those also present in most events. In other terms, the results obtained with a Zipf distribution of a large number of names are similar to those obtained with a uniform distribution of much less names.

Type of Constraints. Finally, Figure 11 shows how performance changes when changing the type of constraints. In particular, we measured the processing time when changing the percentage of constraints involving numerical values (the remaining involve strings).

At a first look, the behavior registered in Figure 11a is counter intuitive, since apparently matching numerical values is more expensive than matching strings. On the other hand, we notice that the chances a numerical constraint matches an event are higher than those of a string constraint. On the average, about 1% of interfaces are selected when all constraints are on strings, and about 7% when all constraints are numerical. This means that a larger number of operations have to be performed in the latter case, increasing counters and checking if all constraint of a filter have been matched; while the low chance of matching strings means that a few comparisons on characters are enough to discover that the constraint does not matches. This has a very limited impact on CCM, where all such operations are performed in parallel, while it decreases the performance of SFF. Accordingly, as shown in Figure 11b, the speedup of

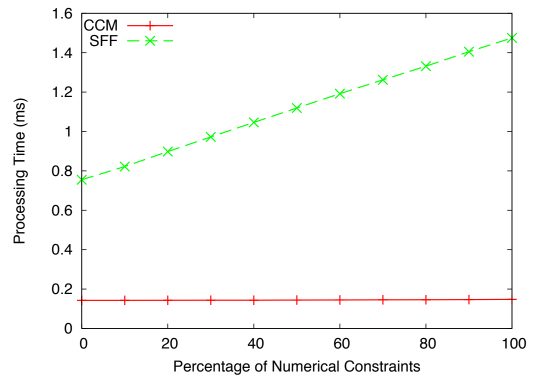


(a) Event Processing Time

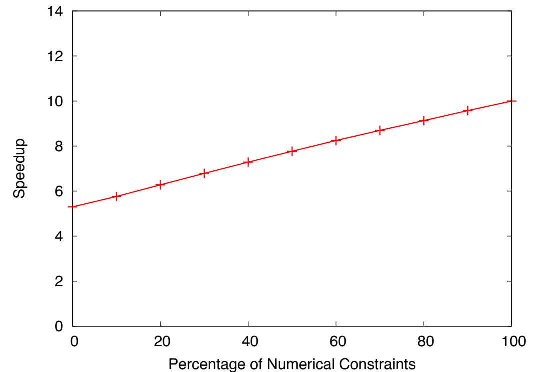


(b) Speedup

Figure 10: Number of Different Names



(a) Event Processing Time



(b) Speedup

Figure 11: Type of Constraints

CCM increases linearly with the number of numerical constraints deployed.

Final Considerations. Our experience and the results presented so far allow us to draw some general conclusions about CUDA programming in general and content-based matching on CUDA in particular.

First of all we may observe how programming CUDA is (relatively) easy, while attaining good performance is (very) hard. Memory accesses and transfers tend to dominate over processing (especially in our case) and must be carefully managed (see the case of the initial cudaMemset operation), while having thousands of threads, even if they are created to be immediately destroyed, has a minimal impact.

Also, we observed the presence of a fixed cost to pay to launch the kernel, which makes (relatively) simple problems not worth being demanded to the GPU (see the case of simple filters with 1-2 constraints). Fortunately, at least for our problem, it is possible to determine if we are in one such cases or not before starting processing. In practical terms, we leverage this characteristics by implementing a translator from Siena events and subscriptions to those managed by our matching engine. This way the two engines, SFF and CCM, can be both integrated into Siena (and any similar publish-subscribe middleware) to postpone at run-time the decision if activating CCM from the very beginning or letting SFF solve the matching problem until enough subscriptions are collected that justify using CCM.

Focusing on the specific problem we addressed, we notice how using a GPU may provide impressive speedup w.r.t. using a CPU, with the additional, fundamental advantage, of leaving the CPU free to focus on those jobs (like I/O) that do not fit GPU programming. Moreover, this speedup grows with the scale of the problem to solve.

A final consideration we can do is related with the availability of multicore CPUs. To the best of our knowledge, all existing matching algorithms are sequential, and consequently they do not leverage multicore hardware. On the other hand, an event-based infrastructure collects events from multiple sources and could, in principle, match them in parallel, using multiple cores. This promises to increase throughput in traditional systems (those using CPUs to perform matching) but the same promise holds for systems using GPUs. In fact, the new version of CUDA, currently under beta test, fully supports multiple kernels launched by different CPU threads in parallel. Moreover, it is possible to add more GPUs operating in parallel, to further improve performance.

6. RELATED WORK

The last years have seen the development of a large number of content-based publish-subscribe systems [26, 4, 13, 25, 12] first exploiting a centralized dispatcher, then moving to distributed solutions for improved scalability. Despite their differences, they all share the problem of matching event notifications against subscriptions.

Two main categories of matching algorithms can be found in the literature: *counting* algorithms [14, 10] and *tree-based* algorithms [2, 5]. A counting algorithm maintains a counter

for each filter to record the number of constraints satisfied by the current event. Both SFF and CCM belong to this category. A tree-based algorithm organizes subscriptions into a rooted search tree. Inner nodes represent an evaluation test; nodes at the same level evaluates constraints with the same name; leaves represent the received predicates. Given an event, the search tree is traversed from the root down. At every node, the value of an attribute is tested, and the satisfied branches are followed until the fully satisfied predicates (and corresponding interfaces) are reached at the leaves.

To the best of our knowledge, no existing work has demonstrated the superiority of one of the two approaches. Moreover, SFF, which we used as a term of comparison in Section 5, is usually cited among the most efficient matching algorithms.

Despite the efforts described above, content-based matching is still considered to be a complex and time consuming task [9]. To overcome this limitation, researchers have explored two directions: on the one hand they proposed to distribute matching among multiple brokers, exploiting covering relationships between subscriptions to reduce the amount of work performed at each node [8]. On the other hand, they moved to probabilistic matching algorithms, trying to increase the performance of the matching process, while possibly introducing evaluation errors in the form of false positives [6, 20]. The issue of distributing the event dispatcher is orthogonal w.r.t. the matching algorithm. Indeed, brokers have to perform the same kind of matching we analyzed here. Accordingly, CCM can be profitably used in distributed scenarios, contributing to further improve performance. At the same time, some of the ideas behind CCM can be leveraged to port probabilistic algorithms inside GPUs. Indeed, probabilistic matching usually involves encoding events and subscriptions as Bloom filters of pre-defined length, thus reducing the matching process to a comparison of bit vectors: a strongly data parallel process, which perfectly fits CUDA. We plan to explore this topic in the future.

Recently, a few works have addressed the problem of parallelizing the matching process, using ad-hoc (FPGA) hardware [33], or multi-core CPUs [16]. While the first approach has limited applicability, since it requires ad-hoc hardware, we plan to compare, and possibly combine, the use of multicore CPUs with GPUs in the future.

The adoption of GPUs for general purpose programming is relatively recent and was first enabled in late 2006 when Nvidia released the first version of CUDA [11]. Since then, commodity graphics hardware has become a cost-effective parallel platform to solve many general problems, including problems coming from linear algebra [21], image processing, computer vision, signal processing [31], and graphs algorithms [19]. For an extensive survey on the application of GPU for general purpose computation the reader may refer to [30]. To the best of our knowledge, our work is the first one that explores the possibility of using GPUs to implement a content-based matching algorithm.

7. CONCLUSIONS

In this paper we presented a new content-based matching algorithm designed to run on GPUs implementing the CUDA architecture. We compared it with SFF, the matching algorithm of Siena, well known for its efficiency. Results demonstrate the benefits of GPUs in a wide spectrum of scenarios, showing speedups from 7x to 13x in most of them, especially the most challenging ones. This reflects the difference in processing power of the two platforms, as acknowledged by Intel itself [23]. Moreover, delegating to the GPU all the effort required for the matching process potentially brings other advantages to the whole system, by leaving the main CPU free to perform other tasks.

In our analysis we observed how the main overhead in using GPUs to perform content-based matching has to do with the need of transferring data between the main and GPU memory through the (relatively) slow PCI-E bus. Our algorithm reduces these memory transfers to the minimum making it convenient also in small scenarios.

Notice that our algorithm was presented here focusing on the case of content-based publish-subscribe systems, but the problem of matching is more general than this. Indeed, as observed by others [10, 13], several applications can directly benefit from a content-based matching service. They include intrusion detection systems and firewalls, which need to classify packets as they flow on the network; intentional naming systems [1], which realize a form of content-based routing; distributed data sharing systems, which need to forward queries to the appropriate servers; and service discovery systems, which need to match service descriptions against service queries.

Finally, in this paper we focused on exact matching, i.e., on an algorithm that does not produce false positives, since this makes our solution amenable to be straightforwardly integrated in most systems that require content-based matching. On the other hand, we plan to explore probabilistic, inexact matching as well, which we believe could receive great advantages if implemented on a GPU, using techniques similar to those we adopted in CCM.

Acknowledgment

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

8. REFERENCES

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. of the 17th Symp. on Operating Syst. Principles (SOSP99)*, pages 186–201. ACM Press, Dec 1999.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proc. of the 8th Symp. on Principles of distributed computing*, PODC '99, pages 53–61, New York, NY, USA, 1999. ACM.

[3] T. S. Axelrod. Effects of synchronization barriers on multiprocessor performance. *Parallel Comput.*, 3:129–140, May 1986.

[4] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical report, DIS, Università di Roma "La Sapienza", 2005.

[5] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proc. of the 23rd Intl. Conf. on Software Engineering*, ICSE

'01, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

[6] A. Carzaniga and C. P. Hall. Content-based communication: a research agenda. In *SEM '06: Proceedings of the 6th Intl. Workshop on Software engineering and middleware*, Portland, Oregon, USA, Nov. 2006. Invited Paper.

[7] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. of the 9th Symp. on Principles of Distributed Computing*, pages 219–227, Portland, Oregon, July 2000.

[8] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, Mar. 2004.

[9] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, Arizona, Oct. 2001. Springer-Verlag.

[10] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proc. of ACM SIGCOMM 2003*, pages 163–174, Karlsruhe, Germany, Aug. 2003.

[11] CUDA. http://www.nvidia.com/object/cuda_home_new.html, 2011. Visited Jan. 2011.

[12] G. Cugola and G. Picco. REDS: A Reconfigurable Dispatching System. In *Proc. of the 6th Int. Workshop on Softw. Eng. and Middleware. (SEM06)*, pages 9—16, Portland, nov 2006. ACM Press.

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, June 2003.

[14] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of the 2001 SIGMOD Intl. Conf. on Management of data*, SIGMOD '01, pages 115–126, New York, NY, USA, 2001. ACM.

[15] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29:251–262, August 1999.

[16] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *Proc. of the 3rd Intl. Conf. on Distributed Event-Based Systems*, DEBS '09, pages 8:1–8:4, New York, NY, USA, 2009. ACM.

[17] L. Fiege, G. Mühl, and A. P. Buchmann. An architectural framework for electronic commerce applications. In *GI Jahrestagung (2)*, pages 928–938, 2001.

[18] Nvidia GeForce GTX 460. http://www.nvidia.com/object/product-geforce-gtx-460-us.html, 2011. Visited Jan 2011.

[19] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proc. of the 14th Intl. Conf. on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *Proc. of the 2nd Intl. Conf. on Distributed Event-Based Systems*, DEBS '08, pages 71–81, New York, NY, USA, 2008. ACM.

[21] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[22] C. KrÃijgel, T. Toth, and C. Kerer. Decentralized event correlation for intrusion detection. In K. Kim, editor, *Information Security and Cryptology, ICISC*, volume 2288, pages 59–95. Springer Berlin / Heidelberg, 2002.

[23] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proc. of the 37th Intl. Symp. on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

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

[25] G. Mühl, L. Fiege, F. Gartner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proc. of the 10th Intl. Symp. on Modeling, Analysis, and Simulation of Comput. and Telecom. Syst. (MASCOTS02)*, 2002.

[26] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems.* Springer, 2006.

[27] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6:40–53, March 2008.

[28] *Nvidia CUDA C Programming Guide*, 2010.

[29] OpenCL. http://www.khronos.org/opencl, 2011.

[30] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A Survey of General–Purpose Computations on Graphics Hardware. *Computer Graphics*, Volume 26, 2007.

[31] J. D. Owens, S. Sengupta, and D. Horn. Assessment of graphic processing units (gpus) for department of defense (dod) digital signal processing (dsp) applications. Technical report, Department of Electrical and Computer Engineering, University of California,, 2005.

[32] S. Schneidert, H. Andrade, B. Gedik, K.-L. Wu, and D. S. Nikolopoulos. Evaluation of streaming aggregation on parallel hardware architectures. In *Proc. of the 4th Intl. Conf. on Distributed Event-Based Systems*, DEBS '10, pages 248–257, New York, NY, USA, 2010. ACM.

[33] K. H. Tsoi, I. Papagiannis, M. Migliavacca, W. Luk, and P. Pietzuch. Accelerating publish/subscribe matching on reconfigurable supercomputing platforms. In *Many-Core and Reconfigurable Supercomputing Conference (MRSC)*, Rome, Italy, 03/2010 2010.