High-Performance Location-Aware Publish-Subscribe on GPUs

Gianpaolo Cugola, Alessandro Margara

High-Performance Location-Aware Publish-Subscribe on GPUs

Gianpaolo Cugola and Alessandro Margara

Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32, Milan, Italy
{cugola,margara}@elet.polimi.it

Abstract. Adding location-awareness to publish-subscribe middleware infrastructures would open-up new opportunities to use this technology in the hot area of mobile applications. On the other hand, this requires to radically change the way published events are matched against received subscriptions. In this paper we examine this issue in detail and we present CLCB, a new algorithm using CUDA GPUs for massively parallel, high-performance, location-aware publish-subscribe matching and its implementation into a matching component that allows to easily build a full-fledged middleware system. A comparison with the state-of-the-art in this area shows the impressive increment in performance that GPUs may enable, even in this domain. At the same time, our performance analysis allows to identify those peculiar aspects of GPU programming that mostly impact the performance of this kind of algorithm.

Keywords: Publish-Subscribe Middleware; Location-Awareness; Content-Based Matching; Parallel Hardware; CUDA GPUs

1 Introduction

The diffusion of mobile devices, like notebooks, tablets, and smartphones, which characterized the last few years, has enabled *mobile computing* scenarios based on *location-aware services*. In several cases these services involve some form of *event-based interaction* [22] among the different parties, being them the final users or the components of the mobile applications they use. Examples of services that combine these two models of interaction are location-aware advertising (that reach potential clients based on their location and interests), location-aware social networking (that want to let co-located people to "socialize", i.e., communicate and coordinate), traffic information services (where information reaches interested users based on their location), emergency services (that spread some emergency-related information only to the people present in a specific area where the emergency situation occurs), and so on.

From a software engineering standpoint we notice that this model of interaction can be efficiently supported by a *location-aware publish-subscribe* middleware layer, which lets distributed components *subscribe* to the *notification of events* (often simply "events") happening in a given *location* (usually in the neighborhood of the subscriber) and *publish* the events they want to notify to

others. In particular, a *content-based* infrastructure [14] is the most suited for the kind of services we mentioned, as it provides the level of expressiveness to allow subscribers to express their interests based on the whole content of the event notifications published.

The key element of every content-based publish-subscribe middleware infrastructure is the *matching component* in charge of filtering incoming events against received subscriptions to decide the interested recipients. If we focus on this component and on the algorithm it implements, we notice that none of those proposed so far [1, 14] fits the mobile scenarios we address. Indeed, in order to maximize performance all the matching algorithms assume: (i) that subscriptions are fairly stable and (ii) that they differ from each other (i.e., they include constraints on different attributes[1]), and they leverage these assumptions to index existing subscriptions in complex data structures that minimize the number of comparisons required to match incoming events. Unfortunately, both these assumptions are violated by location-aware publish-subscribe: the location constraint is present in every subscription and subscriptions change frequently since the area of users' interests moves with them.

To overcome these limitations we developed *CLCB - Cuda Location-aware Content-Based matcher*, a new matching algorithm that leverages the processing power of *CUDA Graphical Processing Units (GPUs)* to provide high-performance location-aware content-based matching. In designing CLCB we started from the consideration that modern GPUs in general, and those that implement the CUDA architecture in particular, offer a huge computing power suited for different types of processing, once the right algorithm has been designed. Indeed, GPU programming is a complex task that requires programmers to take into account the peculiarities of the hardware platform, from the memory layout and the way memory is accessed, to the fact that GPU cores can be used simultaneously only to perform data-parallel computations.

In the remainder of the paper we show how CLCB addresses these issues, exploiting all the processing power of CUDA GPUs to minimize both the time to perform location-aware content-based matching of incoming events, and the time to update subscriptions when users move and the area of their interests changes. A comparison against various state-of-the-art systems shows the advantages GPUs may bring to this domain. In particular, next section introduces the location-aware publish-subscribe model we consider, while Section 3 offers an overview of the CUDA programming model. The CLCB algorithm is described in Section 4, while Section 5 evaluates its performance. Finally, Section 6 presents related work and Section 7 provides concluding remarks.

2 The Interaction Model in Details

As the name suggests, location-aware publish-subscribe middleware infrastructures enable a model of interaction among components that extends the tradi-

[1] See Section 2 for the specific nomenclature we use to refer to the format of events and subscriptions.

tional publish-subscribe model by introducing a concept of "location". In particular, we assume a data model that is very common among content-based publish-subscribe middleware infrastructures [9], where event notifications are represented as a set of *attributes*, i.e., $\langle name, value \rangle$ pairs, while subscriptions are a disjunction of *filters*, which, in turn, are conjunctions of elementary *constraints* on the values of single attributes, i.e., $\langle name, operator, value \rangle$ triples.

As far as location is concerned, we assume that each event happens in a specific *location*, while filters have an *area of relevance*. Notice that we associate the area of relevance to filters and not subscriptions on purpose. Indeed, this choice allows to easily model the (common) situation of a user that wants to subscribe to events X happening in an area A_X *or* to events Y happening in a different area A_Y.

For simplicity we assume that locations are expressed using Cartesian coordinates and that the area of relevance of each filter is a circle, which we represent using three floats: two for the center of the area and one for its radius[2].

Given these definitions, the problem of location-aware content-based matching we want to solve can be stated as follows: given an event e happening at a location $loc(e)$ and a set of subscriptions $S = \{s_1, ..., s_n\}$, each composed of a set of filters $s_i = \{f_{i_1}, ..., f_{i_m}\}$ with their area of relevance $area(f_{i_1}), ..., area(f_{i_m})$, find those subscriptions s_j such that:

$$\exists k : loc(e) \in area(f_{j_k}) \wedge matches(e, f_{j_k})$$

where $matches(e, f_{j_k})$ iff every constraint in f_{j_k} is satisfied by an attribute in e.

Moreover, the peculiarity of the scenarios we consider is that the area of relevance of filters changes frequently as it reflects the actual location of the subscribers, which are supposed to move at run-time.

3 Parallel Programming with CUDA

Attaining good performance with parallel programming is a complex task. A naïve paralleling of a sequential algorithm is usually not sufficient to efficiently exploit the presence of multiple processing elements, and a complete re-design of the algorithm may be necessary, taking into account the peculiarity of the underlying architecture and its programming model.

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

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

[2] This choice does not impact our algorithm and can be easily changed to represent both the location of events and the area of relevance of filters differently, including 3-dimensional areas.

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

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

Kernels. They are special functions that define a single flow of execution for multiple threads. When calling a kernel k, the programmer specifies the number of threads per block and the number of blocks that must execute it. Inside the kernel it is possible to access two 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 can be used to differentiate the execution flows of different threads.

Architectural Issues

There are details about the hardware architecture that a programmer cannot ignore while designing an algorithm for CUDA. First of all, the CUDA architecture is built around a scalable array of multi-threaded *Streaming Multiprocessors* (*SMs*). When a CUDA program on the host CPU invokes a kernel k, the blocks executing k are enumerated and distributed to the available SMs. All threads belonging to the same block execute on the same SM, thus exploiting fast SRAM to implement the shared memory. Multiple blocks may execute concurrently on the same SM as well. As blocks terminate new blocks are launched on freed SMs.

Each SM creates, manages, schedules, and executes threads in groups of parallel threads called *warps*. Individual threads composing a warp start together but they have their own instruction pointer and local state and are therefore free to branch and execute independently. On the other hand, full efficiency is realized only when all threads in a warp agree on their execution path, since CUDA parallels them executing one common instruction at a time. If threads in the same warp diverge via a data-dependent conditional branch, the warp

executes each path serially, disabling threads that are not on that path. Only when all paths complete the threads converge back to the same execution flow.

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

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

4 The CLCB Algorithm

In this section we first explain why existing solutions for content-based matching and spatial searching cannot fully satisfy the requirements of a location-aware publish-subscribe middleware, then we present our CLCB algorithm in details.

4.1 Why a New Algorithm?

To support the model of interaction described in Section 2 a middleware has to perform a location and content-based filtering of incoming events against existing subscriptions, which are two complex and time consuming tasks. In principle, this can be done in three ways::

1. by encoding the location of events as part of their content (i.e., as an ad-hoc attribute) and the area of relevance of filters inside the filters themselves (i.e., as an ad-hoc constraint), using a traditional content-based matching algorithm to filter incoming events against existing subscriptions;
2. by separating the location from the content matching problem, to solve the former through an algorithm explicitly designed for spatial searching and the latter through a traditional content-based matching algorithm;
3. by combining the location and content matching steps in a single, ad-hoc algorithm.

The first approach has two limitations: (*i*) in the mobile scenario we target the area of relevance of filters changes frequently and this would require a frequent update of the location constraints, while traditional content-based matching algorithms organize subscriptions into complex data structures that make updates relatively expensive; (*ii*) the presence of a similar constraint (the one about location) on every filter reduces the efficiency of existing algorithms,

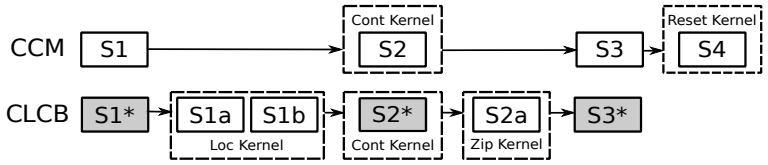


Fig. 1. The CCM and CLCB algorithms compared

which leverage the differences among filters to reduce the number of comparisons to perform. In Section 5 we will measure the actual impact of these limitations.

The second approach is the one we take as a benchmark in Section 5, showing that it is outperformed by our CLCB algorithm, which, in turn, follows the third approach. The next two sections describe how it works.

4.2 CLCB: an Overview

To perform the content-based matching part of its job, CLCB exploits a modified version of CCM, our CUDA-based matching algorithm [24]. CCM stores received subscriptions into the GPU memory, organizing the constraints that compose them into groups, based on the name of the attribute to which they apply. To process an incoming event e, CCM moves e into the GPU memory and evaluates all the constraints that apply to e. For each satisfied constraint it increments a counter[3] associated to the corresponding filter. When the counter for filter f equals the total number of constraints in f then f is satisfied and so is the subscription to which f belongs. When this happens, CCM marks the element that corresponds to the satisfied subscription into an ad-hoc bit vector that represents the result of processing. Such vector is kept in the GPU memory and copied back to the CPU memory when the processing of e finishes. To maximize the utilization of the GPU's computing elements, CCM processes all the constraints in parallel, using a different CUDA thread for each of them, and it increases the counters of filters through atomic operations.

In summary, for each incoming event e CCM performs the following steps (see top of Figure 4.2):

S1 copies e and all the data structures required for processing from the CPU to the GPU memory;
S2 uses the GPU to evaluate all the constraints that apply to the attributes of e in parallel, counting the satisfied constraints and setting the bit vector of matched subscriptions;
S3 copies the bit vector of matched subscriptions to the CPU memory.
S4 resets the bit vector of matched subscriptions and the counters of satisfied constraints associated to each filter to 0, ready for processing a new event.

A naïve approach to add location-awareness to CCM would be to add an additional step, before or after the content-based matching, where checking the

[3] CCM belongs to the vast category of "counting" algorithms. See Section 6.

location of event e against the area of relevance of stored filters. Unfortunately, this would not attain the best performance. Instead, we followed a different approach (see bottom of Figure 4.2), which combines location and content-based matching in a single, integrated process. In particular, we added two intermediate steps between S1 and S2. For each filter f:

S1a performs an initial content-based pre-filtering, by encoding the names of attributes in e as a Bloom filter [5] that is compared with the pre-calculated Bloom filter that encodes the names of constraints in f. This allows to efficiently[4] compare the two sets of names, discarding f if it includes constraints on attributes not present in e;

S1b checks the area of relevance of f against the location of e.

Both these steps are executed into a single CUDA kernel (named `Loc` in Figure 4.2), using a different CUDA thread to process each filter in parallel.

The presence of the two steps above allowed us to optimize the content-based matching algorithm of CCM (i.e., step S2) to immediately skip those filters whose area of relevance does not match the location of e. We also modified CCM by observing that it was designed for scenarios where a large number of filters is included into a relatively small number of subscriptions. For this reasons it encodes its results (i.e., the set of matched subscriptions) as a bit vector. On the contrary, we expect most location-aware services to have a large number of subscriptions and to select only a small portion of them while processing an event. In this scenario, encoding results as a large and sparse bit vector becomes inefficient. Accordingly, we added the following step just before S3:

S2a converts (using a CUDA kernel named `Zip` in Figure 4.2) the bit vector generated by CCM into an array of matched subscription identifiers.

This is the result that is copied back to the CPU memory at the new step S3*.

Finally, we were able to move most of the processing formerly in S4 into the `Loc` kernel (which implements steps S1a and S1b), reducing the total number of kernels to launch.

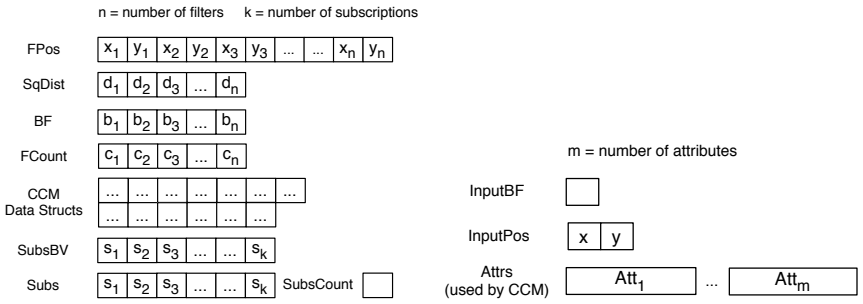
4.3 CLCB in Detail

Data structures. Figure 4.3 shows the data structures used in CLCB. In particular, Figure 2(a) shows the data structures stored on the GPU memory persistently (across event processing)[5].

Vector `FPos` stores the center (x and y coordinates) of the area of relevance of each filter as two 32 bit floats (more precisely we use a `float2` CUDA type). Nearby is vector `SqDist`, which stores the square of the radius of the corresponding area of relevance. These two data structures are separated from the others to simplify location updates.

[4] Comparing two Bloom filters for set inclusion requires a single bit-wise *and* plus a comparison.

[5] We focus on the CLCB specific data structures, leaving aside those used by CCM. The interested reader may find a precise description of these structures in [24]



(a) Stored on the GPU (b) Generated from the input event

Fig. 2. Data structures of CLCB

`BF` is a vector of Bloom filters (as 32 bit integers), which encodes, for each filter f, the set of names of the constraints in f (see step S1a above). Notice that 32 bit may seem small for a Bloom filter, but: (*i*) they are enough to guarantee 10% of false positives with up to 6 constraints per filter (independently from the number of names in the workload); (*ii*) we use them to quickly identify those filters that have no chance to be matched, i.e., we may tolerate some false positives if this increases performance, and moving to 64 bit or more would greatly reduce performance as the bandwidth toward GPU memory is limited.

Vector `FCount` stores, for each filter f, the number of constraints currently satisfied in f (including the implicit constraint on the area of relevance). As such, it is used both during the location-based filtering step S1b and during the content-based filtering step S2.

`SubsBV` is the bit vector of matched subscriptions generated after the location-based and content based filtering steps S1a, S1b, and S2 take place.

Finally, vector `Subs` represents the final result of the CLCB computation (generated by step S2a). It stores the (32 bit) identifiers of the subscriptions satisfied by the event. `SubsCount` contains the number of matched subscriptions, i.e., the number of elements actually stored in `Subs`.

Notice that the internal organization of data into `FPos`, `SqDist`, `BF`, and `FCount` allows to store the relevant information regarding filters into contiguous memory regions. This allows to implement steps S1.a and S1.b (i.e., the entire `Loc` kernel) in such a way that threads with contiguous ids access contiguous memory regions: a key choice to allow the CUDA runtime to optimize memory accesses by grouping them into a reduced number of memory-wide operations.

The data structures that encode the relevant information about the event e under processing are shown in Figure 2(b). They are built by the CPU and transferred to the GPU memory for processing. In particular, `InputBF` is the Bloom filter that encodes (as a 32 bit integer) the names of e's attributes; `InputPos` is a `float2` element that represents the coordinates of the location of e. Finally, vector `Attrs` stores the attributes of e, which are used during the content-based filtering step S2.

Algorithm 1 The `Loc` and `Zip` kernels in details

```
1: function Loc
2:     id = blockId.x · blockDim.x + threadId
3:     if id ≥ n then
4:         return
5:     end if
6:     if id==0 then
7:         SubsCount = 0
8:     end if
9:     if ! includes(InputBF, BF[id]) then
10:        FCount[id] = 0
11:        return
12:    end if
13:    Pos = FPos[id]
14:    sqDistX = (InputPos.x − Pos.x) · (InputPos.x − Pos.x)
15:    sqDistY = (InputPos.y − Pos.y) · (InputPos.y − Pos.y)
16:    if sqDistX + sqDistY > SqDist[id] then
17:        FCount[id] = 0
18:    else
19:        FCount[id] = 1
20:    end if
21: end function
22:
23: function Zip
24:    id = blockId.x · blockDim.x + threadId
25:    if id ≥ k then
26:        return
27:    end if
28:    if SubsBV[id] == 1 then
29:        SubsBV[id] = 0
30:        position = atomicAdd(SubsCount, 1)
31:        Subs[position] = id
32:    end if
33: end function
```

Processing Kernels. Algorithm 1 shows the `Loc` and `Zip` kernels, while the details of the `Cont` kernel can be found in [24]. The `Loc` kernel performs steps S1a and S1b. In implementing it, we tried to stop the execution of threads as soon as possible. This increases the chances that all the threads in a warp (the minimum allocation unit for CUDA) terminate, thus freeing resources for other threads. Moreover, this also reduces the number of memory accesses performed by each thread, which often represents the main bottleneck in CUDA. In particular, each thread of the `Loc` kernel first computes its id and uses it to decide which filter to consider, from 0 to $n - 1$. Since each block consists of a fixed number of threads (usually 256) and threads are allocated in blocks, it is often impossible to allocate the exact number of threads required (n in our case). Accordingly we check, at Line 3, if the current thread is required, discarding useless threads immediately. Line 7 is performed by a single thread (the one with id = 0), which resets the counter of matched subscriptions `SubsCount`. This is a necessary step to be ready to process the new event and embedding it into this kernel reduces the number of operations that the host program issues to the GPU.

Lines 9–12 encode step S1a above. Each thread performs a preliminary content-based evaluation of the filter f for which it is responsible, by comparing the Bloom filter that encodes the set of constraint names in f (i.e., `BF[id]`) with the Bloom filter that encodes the set of attribute names in e (i.e., `InputBF`).

This operation only requires to read a 32 bit element (`InputBF`) shared among threads (automatically cached by modern NVIDIA GPUs), while another 32 bit element for each thread must be read from the main memory (`BF[id]`). As already mentioned, the layout of the `BF` data structure and the way it is accessed by threads allows the CUDA runtime to combine the latter reads into a reduced number of memory-wide operations.

If the content-based comparison above succeeds, each thread compares the location of the input event and the area of relevance of the filter it is responsible for. This is done at Lines 13–16. If the comparison succeed the thread sets the counter of satisfied constraints of the current filter (`FCount[id]`) to 1. In any other case this value is reset to 0 (Lines 10 and 17). Again, this is a necessary step to be ready to process the new event and embedding it into the `Loc` kernel allows to eliminate kernel `Reset`, which was originally part of CCM (see Figure 4.2). We also notice that the introduction of kernel `Loc` allows to modify the CCM algorithm (kernel `Cont`) so that each thread there immediately checks the value of the counter for the filter it is responsible for. If it is 0 than the thread can immediately terminate as it is sure that either the Bloom filter based content check or the location-based matching did not succeed.

After the `Loc` and `Cont` kernel runs, we execute the `Zip` kernel, whose pseudocode is shown in Algorithm 1. It executes one thread for each subscription. At the beginning (Line 24) every thread computes its id and immediately terminates if it exceeds the total number of subscriptions. Then, every thread checks one element of the `SubsBV` bit vector in parallel. If the element is set to 0, the thread can safely terminate. Otherwise, it resets the element to 0 to be ready for the next event and appends the identifier of the corresponding subscription to vector `Subs`. To do so, it atomically increases `SubsCount` using the `atomicAdd` function provided by the CUDA runtime. This function returns the old value of `SubsCount`, which the thread uses to access the `Subs` vector.

Reducing memory accesses. In the `Loc` kernel, each thread accesses a different element of vector `FCount`, setting it to 1 if the filter matches the event and to 0 in the other cases (i.e., to be ready for the next steps). In most application scenarios, we expect that only a (small) fraction of `FCount` needs to be set to 1, since only a small fraction of the filters is geographically close to the location of the event under processing. Accordingly, we could reduce memory accesses by reducing the number of times we have to reset the `FCount` elements to 0. To obtain this result we notice that each element of `FCount` must be a 32 bit integer for architectural reason: the `Cont` kernel needs to increase it using an `atomicAdd` operation, which is defined only for 32 bit integers. However, we expect filters to include only a small number of constraints, much less than 256, so a single byte would be enough for our purposes. Moving from these premises, we optimized the `Loc` and `Cont` kernels grouping runs by four. At run $r = 1, ..., 4$ we set to 1 the r^{th} byte of `FCount[id]` if necessary, while we reset the whole 4 bytes only at the first run. This way we reset the `FCount` vector only once every four runs, which results in an average improvement in processing time of about 20%.

Reducing latency. Both the operations of launching a kernel and issuing a memcopy between the CPU and the GPU memory in CUDA are asynchronous and initiated by the host CPU. A straightforward implementation could force the CPU to wait for an operation involving the GPU to finish before issuing the following one. This approach, however, pays the communication latency introduced by the PCI-Ex bus for every command sent to the GPU. To solve this issue, CLCB makes use of a *CUDA Stream*, which is a FIFO queue where the CPU can put operations to be executed sequentially on the GPU. This way we may explicitly synchronize the CPU and the GPU only once for each event processed, to make sure that the GPU has finished its processing and all the results have been copied into the main memory before the CPU accesses them. This approach enables the hardware to issue all the instructions it finds on the Stream immediately, paying the communication latency only once.

5 Evaluation

This section evaluates CLCB, comparing it with existing approaches for content-based matching and spatial indexing. We evaluate the time required to match an incoming event and the time required to update the area of relevance of a filter. Moreover, we show how the relatively small amount of memory provided by existing GPUs does not constitute a limitation for our algorithm.

Experiment setup. To study the performance of CLCB we started from a default scenario (see Table 1) that represents a typical metropolitan situation, with 250k subscribers, each installing 10 filters. The area of relevance of filters is fixed at 0.01% of the entire area under analysis: this is equivalent to consider a circle with 76m radius in a city like Milan. Each filter contains 3 to 5 constraints and is satisfied by 0.5% of incoming events, on the average. We consider subscribers (hence the area of relevance of the filters they sent) to be uniformly distributed. Since in some scenarios this could be a non-realistic assumption, we also considered the case where subscribers are concentrated in certain areas (actually, we will show that this further increases the advantages of CLCB w.r.t. existing approaches). To compute the time required to match each event, we submitted 1000 events having 3 to 5 attributes each, and calculated the average processing time. Similarly, to compute the update time we changed the area of relevance of 1000 filters, calculating the average update time.

All tests have been run on a 64bit Linux PC with an AMD Phenom II x6 CPU running at 2.8GHz and 8GB of RAM. We used GCC 4.6 and the CUDA Toolkit 4.1. The GPU is a Nvidia GTX 460 with 1GB of RAM. We repeated each test several times with different seeds to generate subscriptions and events. For each measure, we report the average value we measured, omitting, for readability, the 95% percentile, which is always below 1% of the measured value.

Limitation of content-based matching algorithms. As already mentioned in Section 4.1, it is theoretically possible to provide a location-based service using a traditional content-based matching algorithm and encoding the area of relevance of each subscription as a special constraint. Here, we show the

Table 1. Parameters for the default scenario

Number of events	1000
Attributes per event, min-max	3-5
Number of subscriptions	250000
Content constr. per filt., min-max	3-5
Filters per subscription	10
Number of distinct names	100
Area covered by each filter	0.01%
Spatial distribution	Uniform
Selectivity of content	0.5%

Table 2. Processing and update times of content-based matching systems

	SFF	BETree
Proc. Time w/o Location	13.78 ms	1.48 ms
Proc. Time w/ Location	118.69 ms	84.09 ms
Update Time	10151 ms	n.a.

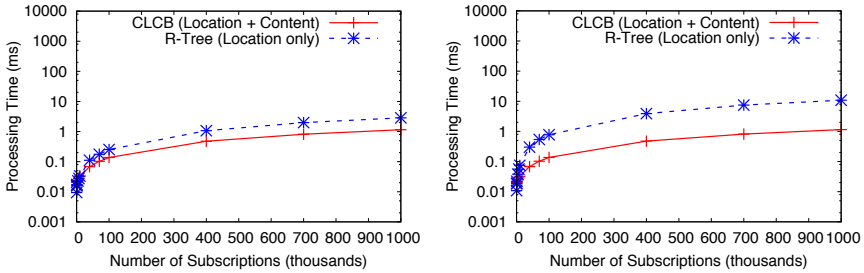
limitations of this approach by analyzing the performance of two state of the art algorithms: SFF [9] v. 1.9.5 and BETree [27]. They are among the fastest implementations of the two main classes of matching algorithms, i.e., counting algorithms and tree-based algorithms, respectively (see Section 6).

The results we collected using the parameters of our default scenario are shown in Table 2. We first consider the average time required to process a single event: in the first line we consider only content-based matching (there is no area of relevance associated to filters). In this scenario SFF requires 13.78ms to process a single events, while BETree requires 1.48ms. In the second line of Table 2, we also consider the area of interest. We observe a significant increase in processing time: SFF becoming 8.6 times slower and BETree becomes 56.8 times slower. This happens because the complexity of both the (classes of) algorithms is influenced by the number of constraints on each event attribute. For this reason, adding the location constraint (that applies to the same attribute of every event) to each filter represents a challenging scenario for these algorithms. By comparison, in the default scenario, CLCB requires 0.306ms to perform both content-based and location-based matching, providing a speedup of 389× over SFF and 275× over BETree.

The third line of Table 2 shows the average time required to update the area of relevance of a single filter. SFF builds some complex indexing structures to speedup the processing: since it was designed to work under the assumption of rare changes in the interests of subscribers, it does not provide primitives to update filters, but completely re-creates all indexes after each update. For this reason, updating the area of relevance requires more than 10s with SFF. As for BETree, we could not made this test directly, as we only had access to an executable binary that implements the BETree algorithm starting from a file that holds all the subscriptions. On the other hand, given the way BETree operates, we expect results similar to those of SFF.

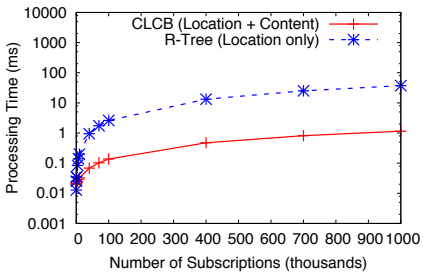
Limitation of location-based filtering algorithms. Several data structures have been proposed in the literature to store and retrieve localized information. They are generally known as spatial indexing structures and the most widely adopted is R-Tree [20]. In the following we compare CLCB against R-Tree. In particular, we used the R*-Tree variant[6], known for its efficiency [3]. Since the

[6] We adopted the open source C++ `libspatialindex` library 1.7.0 available at http://libspatialindex.github.com

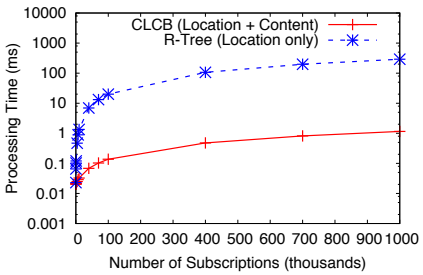


(a) Each filter covers 0.001% of area

(b) Each filter covers 0.01% of area



(c) Each filter covers 0.1% of area



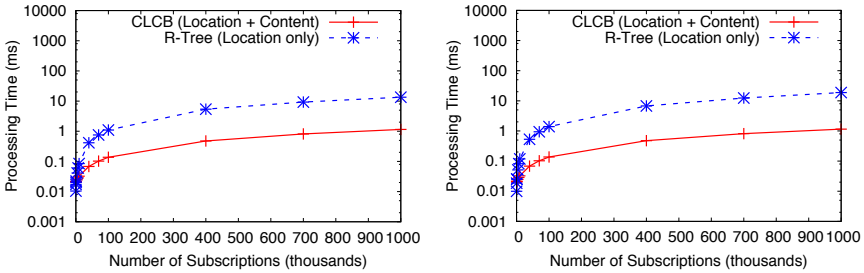
(d) Each filter covers 1% of area

Fig. 3. Matching times of CLCB and R-Tree compared (uniform spatial distribution)

performance of R-Tree is influenced by several parameters, we conducted some preliminary tests to determine the most convenient ones for our scenarios.

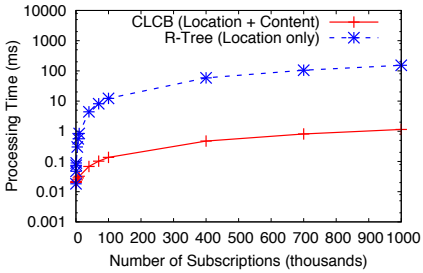
Here we compare the event matching and location update times of R-Tree and CLCB while varying (*i*) the percentage of area covered by each filter and (*ii*) the geographical distribution of filters and events. All the other parameters are defined as in our default scenario. As for the area of relevance, we consider 4 different coverage percentages: 0.001%, 0.01%, 0.1%, and 1%. In our metropolitan scenario, considering the city of Milan, this means using areas of relevance with a radius of 24m, 76m, 240m, and 759m, respectively. Considering an entire country like Italy these numbers become: 1km, 3km, 10km, and 31km, while in a small city like Pisa they become: 7m, 22m, 71m, and 225m, respectively.

Figure 3 shows the average time required by R-Tree and CLCB to process a single event when using a uniform geographical distribution for events and filters. Notice that CLCB executes both location-based and content-based matching, while R-Tree only provides location-based filtering. First, we observe that the advantage of CLCB increases with the number of subscriptions. With very small problems, R-Tree is more efficient, since CLCB pays the (almost fixed) overhead for launching the kernels and moving input data and results between the CPU and the GPU memory. However, also in these cases, the performance of CLCB and R-Tree are comparable; moreover, CLCB starts to provide better results with about 400 subscriptions (i.e., 4000 filters). Second, the area of relevance of each filter does not impact on the performance of CLCB; on the contrary, it has a great influence on R-Tree, whose performance degrades when the areas of

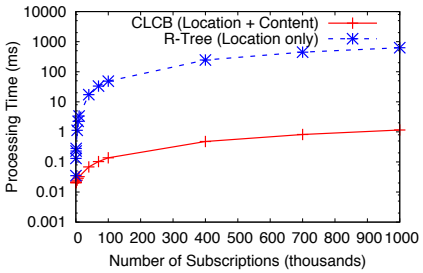


(a) Each filter covers 0.001% of area

(b) Each filter covers 0.01% of area



(c) Each filter covers 0.1% of area



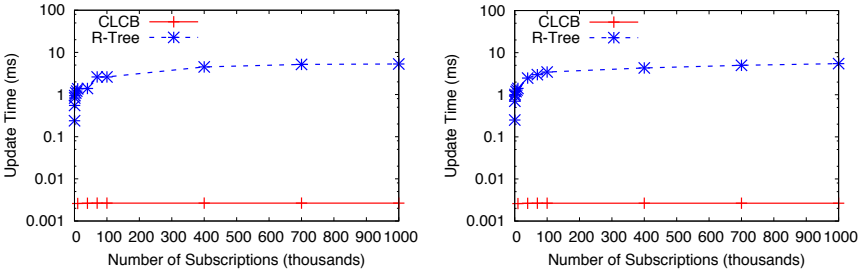
(d) Each filter covers 1% of area

Fig. 4. Matching times of CLCB and R-Tree compared (zipf spatial distribution)

filters overlap. With 1 million subscriptions (10 millions filters), CLCB provides a speedup of 2.47× when each filter covers 0.001% of the area, and 251.7× speedup when each filter covers 1% of the area.

In most application scenarios, we expect events and filters to exhibit an uneven geographical distribution, with higher density of population concentrated around a few areas of interest. We analyze how this aspect impacts on performance in Figure 4, where we use a Zipf power law to generate the location of events and the center of the area of interest of filters. This change does not significantly impact the performance of CLCB. On the contrary, it has a great impact on the matching time of R-Tree, which increases significantly w.r.t. Figure 3. Even in the less expensive scenario in which each filter covers 0.001% of the area (Figure 4 a), it exhibits a matching time of more than 10ms with 700k subscriptions or more. With 1 million subscriptions, CLCB provides a speedup of 11.7×, 16.3×, 132.8×, and 544× with filters covering 0.001%, 0.01%, 0.1%, and 1% of the area, respectively.

Figure 5, shows the average time required to move the area of relevance of a single filter. Since our tests showed that this time is only marginally influenced by the average size of the area of relevance, both for R-Tree and CLCB, Figure 5 shows the results obtained in our default scenario (i.e., when each filter covers 0.01% of the area). Notice also that, according to some preliminary tests we made, the update time is independent from the specific changes we consider, being changes that move the area of relevance a few meters away, or changes that move it far away; and, again, this is true both for CLCB and R-Tree. This



(a) Uniform distribution (b) Zipf distribution

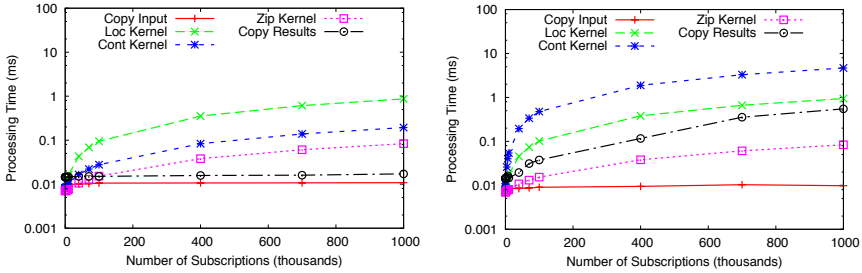
Fig. 5. Update times of CLCB and R-Tree compared

means that the results we collect do not depend from the specific pattern of mobility [6] followed by clients. Accordingly we considered 1000 random changes and measured the average time to update the area of interest of filters.

Given these premises, we observe that R-Tree organizes filters into a balanced tree. Moving the area of relevance of a filter requires removing the old area of relevance and adding a new one: in some cases this operation may also require a recursive re-balancing of the tree. The case of CLCB is much simpler since each update only requires the copy of 2 float values (32 bit each) from the CPU to the GPU memory: this takes a constant time of 2.65 microseconds. On the contrary, the update time for R-Tree increases with the number of filters installed. With 1 million subscriptions, the update time is about 5.26ms: in this scenario CLCB provides a speedup of 1985× when considering a uniform distribution for the areas of relevance. By comparing Figure 5 a and b, we observe that, differently from the matching time, the update time is only marginally influenced by the geographical distribution of filters (being uniformly distributed or aggregated around certain areas).

Analysis of CLCB. To better understand the performance of CLCB, we measured the time spent in the five steps described in Section 4 (see Figure 4.2): (*i*) Copy Input, where the CPU generates the input data structures and copies them to the GPU (step S1 in figure); (*ii*) execution of the `Loc` kernel performing location-based filtering; (*iii*) execution of the `Cont` kernel performing content-based filtering; (*iv*) execution of the `Zip` kernel that stores matching subscriptions into a compact array; (*v*) Copy results, where the results are copied back from the GPU to the CPU memory (step S3 in figure).

Figure 6(a) shows the results we measured when changing the number of subscriptions, while the remaining parameters are defined as in our default scenario. First of all, we observe that the first and last steps (copy of input from the CPU to the GPU and copy of results from the GPU to the CPU) require a small amount of time (less than 0.1ms), which does not depend from the number of filters deployed in the system. Moreover, they are dominated by the cost of kernels execution, which increases with the complexity of the problem. This is a significant result: often, when porting algorithms to the GPU, the overhead required for moving information back and forth from the CPU to the GPU mem-



(a) Filter selectivity = 0.5% (b) Filter selectivity = 100%

Fig. 6. Distribution of times in CLCB

ory overcomes the benefits of parallel execution. This is not the case for CLCB, which minimizes the amount of information transferred from the CPU to the GPU (only the content of the event under analysis) and from the GPU to the CPU (only the list of matched subscriptions).

To stress the system even more, we considered a second scenario, where the constraints of filters were chosen to select every event. The only form of filtering remaining depends on the location of events and the area of relevance of filters. Albeit unrealistic, this is an interesting scenario for the extreme challenges it brings to CLCB: (*i*) the content-based pre-filtering of events does not provide any advantage; (*ii*) a larger amount of subscriptions is selected and need to be transferred back to the CPU, at the end of computation; (*iii*) the `Cont` kernel becomes more expensive, since all constraints are satisfied and no thread can be stopped. Figure 6(b) shows the results we measured in this scenario. Interestingly, the execution time of the `Loc` kernel is only marginally influenced and the same happens to the `Zip` kernel; moreover, despite the time to copy results back to the CPU memory grows, it remains below 0.6ms even in the larger scenario with 1 million of subscriptions. In practice, the overall running time is dominated by the `Cont` kernel, which is the one registering the largest growth. This is an inevitable consequence of the extreme scenario. As we already verified, every content-based matching algorithm suffers when filters contain the same names of the event under processing.

Memory consumption. Memory often represents a serious bottleneck to the scalability of GPU-based algorithms. Indeed, GPUs often host a limited amount of memory w.r.t. CPUs (up to 4GB, at most). Moreover, to increase performance, information needs to be flattened out and stored into contiguous regions, often increasing memory consumption.

Figure 7 shows the GPU memory demand of CLCB when increasing the number of filters deployed. Our reference hardware (which is a cheap card only providing 1GB of RAM) could support more than 1.3 millions subscriptions (13 millions of filters). Considering one subscription (10 filters) per user this means supporting 1.3 millions users (the entire population of a city like Milan) with a single entry-level graphic card. We can reasonably assert that memory consumption does not represent an issue for CLCB.

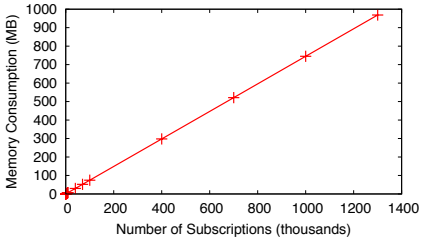


Fig. 7. Memory consumption of CLCB

As a final note, we foresee the possibility to combine CLCB with higher level partitioning algorithms to exploit multiple machines, each one covering a different geographical region.

6 Related Work

This section describes related work in the fields of publish-subscribe middleware and matching algorithms, models and algorithms for location-aware publish-subscribe, and spatial indexing structures.

Publish-subscribe middleware. The last decade saw the development of a large number of publish-subscribe middleware [26, 2, 14, 25, 12] first exploiting a centralized dispatcher, then moving to distributed solutions for improved scalability. A key aspect of every publish-subscribe middleware is the matching algorithm it uses to evaluate incoming events against installed subscriptions.

Two main categories of matching algorithms can be found in the literature: *counting* algorithms [16, 9, 24] and *tree-based* algorithms [1, 7, 27]. In our evaluation, we considered one algorithm for each class: SFF and BETree. SFF maintains a counter for each filter to record the number of constraints satisfied by the current event. On the contrary, tree-based algorithms, like BETree, organize subscriptions into a rooted search tree. Inner nodes represent an evaluation test; leaves represent the received subscriptions. Given an event, the search tree is traversed from the root to the leaves. At every node, the value of a single attribute is tested, and the satisfied branches are followed until the fully satisfied subscriptions are reached at the leaves. To the best of our knowledge, no existing work has demonstrated the superiority of one of the two approaches in every scenario. However, in [27] BE-Tree has been compared against many state-of-the-art matching algorithms, showing best performance in a wide range of scenarios.

Despite the efforts described above, content-based matching is still considered a time consuming task. To overcome this limitation, researchers have proposed to distribute matching among multiple brokers, exploiting covering relationships between subscriptions to reduce the amount of work performed at each node [8]. The use of a distributed dispatching infrastructure is orthogonal w.r.t. CLCB, which can be used in distributed scenarios, contributing to further improve performance. In this field, it becomes critical to efficiently propagate updates to subscriptions through the network of processing brokers. To accomplish this

goal, the idea of parametric subscriptions [21] has been proposed. Again, CLCB could play a role here, as it allows to efficiently install updates at each broker.

The idea of parallel matching has been recently addressed in a few works. In [17], the authors exploit multi-core CPUs both to speedup the processing of a single event and to parallelize the processing of different events. Unfortunately, the code is not available for a comparison. Other works investigated how to parallel matching using ad-hoc (FPGA) hardware [28]. To the best of our knowledge, CCM [24] is the first matching algorithm to be implemented on GPUs, and CLCB is the first to explore GPUs for location-based publish-subscribe. Along the same line, in [13] we explored the possibility to use GPUs to support Complex Event Processing.

Location-aware publish-subscribe. Location-aware publish-subscribe has been introduced as a key programming paradigm for building mobile applications [10, 15]. Existing proposals mainly focused on techniques for supporting physical and logical mobility of clients in distributed publish-subscribe infrastructures [18], with little or no emphasis on the matching algorithm.

A more general model is represented by context-aware publish-subscribe, in which a generic context (not only location) is associated with each subscription [11, 19]. We plan to study how to extend CLCB to efficiently support the expressiveness provided by the context-aware publish-subscribe model.

Spatial indexing data structures. Spatial indexing data structures organize and store information items that have an associated location on a bi-dimensional or multidimensional space. They provide spatial access methods for extracting stored elements through spatial queries (e.g., to extract all elements contained, in a given area, that overlap a given area, etc.). The most known and widely adopted spatial indexing structure is the R-Tree [20], a variant of a B^+ tree in which each inner node stores the minimum bounding rectangle including all the areas defined in its children. The performance of an R-Tree strongly depends on the heuristics used to decide how to keep the tree balanced: the heuristics used by R*-Tree [3] (our reference) are often cited among the most effective.

A few works have been proposed that aim at parallelizing spatial indexing methods. None of them can be directly applied to the problem we target in this paper. In [29] the authors focus on parallelizing spatial join for location-aware databases. Similar results are presented in [4], where the authors discuss how several data mining techniques can be efficiently implemented on GPUs. In [23], a GPU-based implementation of R-Tree is presented: differently from our approach, parallelism is not exploited to increase the performance of a single query, but to run different queries in parallel. Finally, the work in [30] proposes a technique to speedup processing of large R-Tree structures by storing recently visited nodes on the GPU memory and re-use them for future queries. Different from our approach, the authors focus on structures that do not fit in main memory; only a portion of the computation is performed on the GPU and several interactions between the CPU and the GPU may take place while navigating the tree. A maximum speedup of $5\times$ is achieved with this technique, which is significantly below the results provided by CLCB. More generally, none of these

works is publicly available for comparison, while, to the best of our knowledge, CLCB is the first solution that combines both location-based and content-based filtering into one solution.

7 Conclusions

In this paper, we presented CLCB, a location-aware content-based matching algorithm for CUDA GPUs. CLCB is designed to enable both high-performance processing of events and low-latency update of the area of relevance associated with subscriptions. As such, it may easily be adopted as the core component of a location-aware event-based middleware infrastructure capable of supporting large scale scenarios. A comparison of CLCB with existing content-based matching algorithms and with algorithms for spatial access, shows relevant speedups both in terms of processing and update time.

Acknowledgment

We would like to thank Prof. Hans-Arno Jacobsen and Dr. Mohammad Sadoghi for giving us access to their BETree prototype and for helping us in using it during our tests. This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

1. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: PODC 1999. pp. 53–61. ACM, New York, NY, USA (1999)
2. Baldoni, R., Virgillito, A.: Distributed event routing in publish/subscribe communication systems: a survey. Tech. rep., DIS, Università di Roma "La Sapienza" (2005)
3. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The r*-tree: an efficient and robust access method for points and rectangles. In: SIGMOD 1990. pp. 322–331. ACM, New York, NY, USA (1990)
4. Bohm, C., Noll, R., Plant, C., Wackersreuther, B., Zherdin, A.: Data mining using graphics processing units. In: Hameurlain, A., Kng, J., Wagner, R. (eds.) Transactions on Large-Scale Data- and Knowledge-Centered Systems I, Lecture Notes in Computer Science, vol. 5740, pp. 63–90. Springer Berlin / Heidelberg (2009)
5. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics 1(4), 485–509 (2004)
6. Camp, T., Boleng, J., Davies, V.: A survey of mobility models for ad hoc network research. Wireless Communications and Mobile Computing 2(5), 483–502 (2002)
7. Campailla, A., Chaki, S., Clarke, E., Jha, S., Veith, H.: Efficient filtering in publish-subscribe systems using binary decision diagrams. In: ICSE 2001. pp. 443–452. IEEE Computer Society, Washington, DC, USA (2001)
8. Carzaniga, A., Rutherford, M.J., Wolf, A.L.: A routing scheme for content-based networking. In: INFOCOM 2004. Hong Kong, China (Mar 2004)
9. Carzaniga, A., Wolf, A.L.: Forwarding in a content-based network. In: SIGCOMM 2003. pp. 163–174. Karlsruhe, Germany (Aug 2003)

10. Cugola, G., de Cote, J.: On introducing location awareness in publish-subscribe middleware. In: 25th IEEE ICDCS Workshops. pp. 377 – 382 (june 2005)
11. Cugola, G., Margara, A., Migliavacca, M.: Context-aware publish-subscribe: Model, implementation, and evaluation. In: ISCC 2009. pp. 875 –881 (july 2009)
12. Cugola, G., Picco, G.: REDS: A Reconfigurable Dispatching System. In: SEM 2006. pp. 9—16. ACM Press, Portland (nov 2006)
13. Cugola, G., Margara, A.: Low latency complex event processing on parallel hardware. Journal of Parallel and Distributed Computing 72(2), 205 – 218 (2012)
14. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. 35, 114–131 (June 2003)
15. Eugster, P., Garbinato, B., Holzer, A.: Location-based publish/subscribe. In: NCA 2005. pp. 279 –282 (july 2005)
16. Fabret, F., Jacobsen, H.A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D.: Filtering algorithms and implementation for very fast publish/subscribe systems. In: SIGMOD 2001. pp. 115–126. ACM, New York, NY, USA (2001)
17. Farroukh, A., Ferzli, E., Tajuddin, N., Jacobsen, H.A.: Parallel event processing for content-based publish/subscribe systems. In: DEBS 2009. pp. 8:1–8:4. ACM, New York, NY, USA (2009)
18. Fiege, L., Gartner, F., Kasten, O., Zeidler, A.: Supporting mobility in content-based publish/subscribe middleware. In: Endler, M., Schmidt, D. (eds.) Middleware 2003, Lecture Notes in Computer Science, vol. 2672, pp. 998–998. Springer Berlin / Heidelberg (2003)
19. Frey, D., Roman, G.C.: Context-aware publish subscribe in mobile ad hoc networks. In: Murphy, A., Vitek, J. (eds.) Coordination Models and Languages, Lecture Notes in Computer Science, vol. 4467, pp. 37–55. Springer Berlin / Heidelberg (2007)
20. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: SIGMOD 1984. pp. 47–57. ACM, New York, NY, USA (1984)
21. Jayaram, K., Jayalath, C., Eugster, P.: Parametric subscriptions for content-based publish/subscribe networks. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010, Lecture Notes in Computer Science, vol. 6452, pp. 128–147. Springer Berlin / Heidelberg (2010)
22. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Boston, MA, USA (2001)
23. Luo, L., Wong, M., Leong, L.: Parallel implementation of r-trees on the gpu. In: ASP-DAC 2012. pp. 353 –358 (30 2012-feb 2 2012)
24. Margara, A., Cugola, G.: High performance content-based matching using gpus. In: DEBS 2011 (2011)
25. Mühl, G., Fiege, L., Gartner, F., Buchmann, A.: Evaluating advanced routing algorithms for content-based publish/subscribe systems. In: MASCOTS 2002 (2002)
26. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer (2006)
27. Sadoghi, M., Jacobsen, H.A.: Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In: SIGMOD 2011. pp. 637–648. ACM, New York, NY, USA (2011)
28. Tsoi, K.H., Papagiannis, I., Migliavacca, M., Luk, W., Pietzuch, P.: Accelerating publish/subscribe matching on reconfigurable supercomputing platforms. In: MRSC 2010. Rome, Italy (Mar 2010)
29. Yampaka, T., Chongstitvatana, P.: Spatial join with r-tree on graphics processing units. In: IC2IT 2012 (2012)
30. Yu, B., Kim, H., Choi, W., Kwon, D.: Parallel range query processing on r-tree with graphics processing unit. In: DASC 2011. pp. 1235 –1242 (dec 2011)