The VLDB Journal (2016) 25:719–740
DOI 10.1007/s00778-015-0409-y

SPECIAL ISSUE PAPER

GPU-accelerated string matching for database applications

Evangelia A. Sitaridi[1] · **Kenneth A. Ross**[1]

Received: 29 January 2015 / Revised: 12 September 2015 / Accepted: 22 October 2015 / Published online: 12 November 2015

Abstract Implementations of relational operators on GPU processors have resulted in order of magnitude speedups compared to their multicore CPU counterparts. Here we focus on the efficient implementation of string matching operators common in SQL queries. Due to different architectural features the optimal algorithm for CPUs might be suboptimal for GPUs. GPUs achieve high memory bandwidth by running thousands of threads, so it is not feasible to keep the working set of all threads in the cache in a naive implementation. In GPUs the unit of execution is a group of threads and in the presence of loops and branches, threads in a group have to follow the same execution path; if some threads diverge, then different paths are serialized. We study the cache memory efficiency of single- and multi-pattern string matching algorithms for conventional and pivoted string layouts in the GPU memory. We evaluate the memory efficiency in terms of memory access pattern and achieved memory bandwidth for different parallelization methods. To reduce thread divergence, we split string matching into multiple steps. We evaluate the different matching algorithms in terms of average- and worst-case performance and compare them against state-of-the-art CPU and GPU libraries. Our experimental evaluation shows that thread and memory efficiency affect performance significantly and that our proposed methods outperform previous CPU and GPU algorithms in terms of raw performance and power efficiency. The Knuth–Morris–Pratt algorithm is a good choice for GPUs because its regular memory access pattern makes it amenable to several GPU optimizations.

Keywords Text queries · String matching · GPU Processing · Thread divergence · Cache efficiency

✉ Evangelia A. Sitaridi
eva@cs.columbia.edu

Kenneth A. Ross
kar@cs.columbia.edu

[1] Department of Computer Science, Columbia University, 1214 Amsterdam Ave, New York, NY 10027, USA

1 Introduction

Relational queries often contain string matching conditions within the LIKE predicate. For example, the following query appears as a subquery of Q16 of TPC-H [56]:

```
select s_suppkey from supplier
where s_comment like '%Customer%Complaints%'
```

The "%" character is a wildcard that can match an arbitrary number (including 0) of characters. In this way, the SQL LIKE predicate allows for a limited form of regular expression matching. When evaluating such queries, the system does not need to find all occurrences of a pattern in a string, or even the position of the match(es) within the string. It is therefore possible to apply optimizations for string matching that might not be possible in a more general context, such as terminating string matching early as soon as a match is found.

For database processing, it is common to preprocess data to support efficient search. Indexes can speed up search dramatically for selective queries. For string data, the state-of-the-art index structure is the suffix tree [59]. Unfortunately, even very efficient implementations of suffix trees (or suffix arrays [18]) are an order of magnitude larger than the string data being indexed [18,28,55]. It is unclear whether allocating the extra space is a good use of resources. What is more, a suffix tree lookup does not provide a complete solution for SQL LIKE conditions. If there are many substrings within a

LIKE clause, separated by "%" symbols, one would need to search the index once for each of the substrings, intersect the matching string identifiers, and then verify that the substrings occur in the correct order, without overlap.

String matching is a well-studied problem. Two classic algorithms are the Knuth–Morris–Pratt (KMP) algorithm [29] and the Boyer–Moore (BM) algorithm [8]. To search for a sequence of strings as specified in a LIKE clause with "%" symbols, one can search for each string in turn using any string matching algorithm, starting each search where the previous search left off. On CPUs, empirical studies have shown that BM is generally superior to KMP [12] because it facilitates larger jumps through the target string. Parallelizing these algorithms on CPUs is straightforward: Each available thread can be used to independently search a different string in the database.

GPUs are becoming popular for data processing tasks because of their high memory bandwidth and abundant parallelism. The implementation of various data management operators on GPUs has resulted in significant speedups [4,16,27]. We would like to leverage these same properties of GPUs to accelerate string matching.

Modern GPUs have a moderate amount of onboard RAM. For example, the Nvidia K40 has 12 GB of RAM, with a potential access bandwidth of 288 GB/s. For the purposes of string matching, we propose that the GPU RAM be used to store the string columns for a database system, while the CPU RAM is used to store the remainder of the database. Multiple GPU cards can be combined to store and process (in parallel) larger string collections. String matching can then run at GPU speeds without needing large data transfers between the CPU and GPU. Only string identifiers would be communicated between the CPU and GPU.

1.1 GPU parallelization challenges

Unlike for CPUs, the parallelization of string matching algorithms for GPUs is challenging for several reasons. The challenges are primarily due to the GPU architecture, and apply to any string matching algorithm. Our discussion focuses on the NVIDIA CUDA framework, but similar principles apply to other kinds of GPUs.

1.1.1 Thread divergence

GPUs implement the "single instruction multiple threads (SIMT)" architecture. Threads are organized into SIMT units called *warps*, and the warp size in CUDA is 32 threads. Threads in the same warp start executing at the same program address but have private register state and program counters, so they are free to execute on independent data. Only one common instruction can be executed at a time for a warp. To maximize the number of active threads in a warp, the same execution path must be followed by all threads. When threads in a warp diverge in their execution paths, different paths are serialized by the hardware. For example, a branch followed by a subset of the threads in the warp will cause the remaining threads to be idle, resulting in resource underutilization.

1.1.2 Cache and memory pressure

GPUs achieve a very high degree of parallelism by having many processing elements, each of which can have many warps in flight at any point in time. On the Nvidia K40, there are 15 processing elements called "stream multiprocessors" (SMs), each capable of running 64 concurrent warps. When running at full capacity, there may be $15 \times 64 \times 32 = 30{,}720$ threads in flight. The L2 cache, which is shared by all processing elements has 12,288 128-byte cache lines. With many more threads than cache lines, any algorithm that tries to assign threads independent work is liable to thrash in the L2 cache if those threads each access even a single cache line.

A second kind of cache pressure arises from the limited bandwidth of the L2 cache, which is 1024 bytes per cycle. The cache access granularity is 32 bytes. Each SM can dispatch two independent instructions per thread for each of four concurrent warps. If independent L2 accesses were to occur in every instruction of every thread of every concurrent warp, the bandwidth needed would be $15 \times 4 \times 2 \times 32 \times 32 = 122{,}880$ bytes per cycle. For a workload whose threads all access independent data, the GPU would only be able to sustain one L2 access every 120 instructions, on average.

Global memory is the most plentiful but also the slowest type of memory. GPUs coalesce the global memory accesses of the threads in a warp into as few transactions as possible. If all threads access the same cache line, then there will only be one memory transaction, 32 times fewer than if all threads accessed different cache lines. Shared memory and L1 cache are on-chip fast memories but have limited capacity. Their combined size is 64 KB per SM, and they are two orders of magnitude faster than the global memory. We could use shared memory to achieve coalescing: All threads in an SM would load contiguous strings into the shared memory from global memory. Given 64 warps (2048 threads) in flight in each SM and up to 48 KB of shared memory, all threads could read in a coalesced fashion strings of 24 characters. However, for longer strings, the parallelism would be reduced, resulting in worse performance.

1.2 Our contribution

In this paper, we make the following contributions: First, we study the L2 cache behavior of different string matching algorithms on GPUs. Second, we address and suggest techniques reducing thread divergence of GPU string matching. Third, we suggest alternative string layouts in the GPU mem-

ory reducing cache footprint and evaluate these layouts for different string matching algorithms. Finally, we formulate the problem of memory divergence when threads issue uncoordinated memory requests and suggest how to rewrite KMP to coordinate memory requests.

The main outcome of our study is that memory layout, cache performance, and thread coordination are critical aspects of string matching performance. We demonstrate improvements over prior CPU and GPU implementations. Unlike for CPUs, KMP is superior to BM using the best layout and split configurations for GPUs. The fact that KMP has a steady and predictable memory access behavior makes it easy to coordinate threads to issue favorable memory access patterns. For more general conditions, such as those including disjunctions, the Aho–Corasick algorithm [1] achieves good performance, and is also able to step through the input in a coordinated fashion similar to KMP.

The remainder of the paper is organized as follows: Sect. 2 presents an algorithmic background and gives an overview of related work. Section 3 describes the queries we execute. Section 4 describes the approaches we use to address GPU performance, including segmentation, splitting, and pivoting. We describe optimization and implementation details in Sect. 5, and in Sect. 6 we analyze the performance of different string algorithms and memory layouts for the suggested parallelism methods. Finally, in Sect. 7, we conclude and discuss some future directions of our work.

2 Background and related work

2.1 String matching methods

A database system should be able to give robust and stable performance without performance "surprises" for particular inputs. On GPUs the worst case is amplified due to the SIMT architecture. If one string in a warp exhibits worst-case behavior, then all threads in the warp will suffer the latency consequences. For this reason, we exclude algorithms (such as Boyer–Moore–Horspool [22] and Quick-Search [52]) whose worse-case behavior is $O(mn)$ where m is the pattern size and n the string size. We briefly describe several well-known string-searching algorithms and state their time complexity.

Boyer–Moore (BM) [8] BM preprocesses the pattern and creates two shift tables: BM bad-character and BM good-suffix tables. The good-suffix table stores the shift value for each character in the pattern. The bad-character table stores a shift value for each character in the alphabet, with the shift value being based on the occurrence of the character in the pattern. BM compares the pattern with the string from right to left. The worst-case performance is $O(n+m)$ steps [2]. BM can skip over large parts of the input string if the last character of the pattern does not match the input string.

Knuth–Morris–Pratt (KMP) [29] KMP preprocesses the pattern storing the necessary information in the partial match table. The partial match table, which we call $next$, stores how far we have to backtrack if a comparison of the current position in the pattern with the input fails. Unlike BM, KMP compares the pattern to the input string from left to right, and in case of a failure it shifts the pattern rather than the input based on the partial match table. KMP has worst-case $O(n+m)$ time complexity. In Sect. 5.2 we present and evaluate three alternative KMP implementations.

Aho–Corasick (AC) [1] AC is a generalization of KMP for multiple patterns. The complexity is linear in the length of the patterns and length of the input string. It uses a deterministic finite automaton (DFA) for the matching process. An equivalent non-deterministic finite state automaton (NFA) can be designed, which is typically more space efficient, but results in slower matching performance. In the context of database query processing, we advocate the use of a DFA for string matching.

Multi-pattern matching has been accelerated on Cell Processors for Intrusion Detection Systems [25,49]. Cell processors similarly to GPUs require memory coalescing and in order to use the SIMD instructions different streams processed in parallel are interleaved in SIMD registers.

RE2 is an optimized high-performance regular expression matching library developed by Google [47]. RE2 uses a DFA to detect whether a pattern appears in an input string. Boost is a set of libraries implementing, among others, the KMP and BM algorithms [7]. The SSE 4.2 instruction set provides the CMPISTRI instruction which implements a substring search for patterns fitting in an SSE register [24].

We focus on exact string matching, but there is also a large body of work focusing on approximate string matching, allowing mismatches [41]. Some popular algorithms that are used extensively in computational biology are Smith–Waterman and Needleman–Wunsch [42,51]. Approximate pattern matching can also be implemented using hidden Markov models [6].

2.2 String matching on GPUs

Significant speedups were obtained for the GPU implementation of multi-pattern matching algorithms Wu–Manber [45] and Aho–Corasick [34,64]. Also a multi-pattern version of BM was implemented but had inferior performance to AC [64]. Strings were stored contiguously, and to reduce the global memory latency, input is first loaded in the shared memory to achieve memory coalescing. NFA matching has been implemented to reduce the space required for the automaton [11]. String pivoting in 4-byte units has been suggested for NFA-based string matching to achieve coalescing

[65]. Off-loading string matching onto GPUs has been used to accelerate a digital forensics tool [37] and in intrusion detection system computation [19,26,58].

A simplified version of KMP in older GPU architectures did not result in performance speedups under normal load conditions [26]. KMP has been implemented on the GPU using shared memory to store the pattern and the partial match table [5]. It faced similar issues with thread divergence and it had suboptimal performance compared to PFAC for multiple pattern searches. Loop unrolling in the code of KMP resulted in a minor performance improvement. KMP has been implemented on multi-GPUs [33]. However, there were no exact performance numbers reported, just the speedup compared to a CPU implementation. Various single-pattern matching algorithms have been evaluated, taking advantage of the various GPU memories [58].

Implementations of AC on GPUs have used device memory [64] and texture memory to store the DFA [34,58]. There are multiple ways to parallelize string matching on a set of input strings by varying the mapping of threads to strings. For example, in the PFAC library a thread is allocated for each input byte, and each thread starts matching from the given byte offset [34,35]. If no match is found, the threads terminate searching without backtracking the state automaton. PFAC also uses state number reordering to check more efficiently whether an accepting state has been reached, which we also use in our implementation of AC. PFAC has good performance for intrusion detection systems, but its performance varies by more than order of magnitude for adversarial inputs [35]. (Our AC implementation uses shared memory to store the transition table. This is a realistic choice for database queries where there is a reasonable number of patterns in a single query.) PFAC uses texture memory to store the transition table, but it caches the initial row in the shared memory.

The GPU implementations of Needlman–Wunsch and Smith–Waterman algorithms have resulted in high-performance improvements over CPU implementations [17,32,36]. Hidden Markov models are computationally intensive so evaluating them on GPUs resulted in significant performance speedups exploiting the high parallelism and memory bandwidth [31].

2.3 String matching on GPU databases

Many GPU databases have been implemented following different co-processing approaches for the GPU and CPU processors [9]. Typically, GPU databases first apply dictionary compression on strings and process the compressed representations in the GPU memory [9]. However, not all string predicates, such as wildcards, can be answered using just the compressed representation of the strings. String processing has been identified either as an unsuitable application for GPU acceleration [46] or it has been identified as a still open problem [44]. A data structure called tablet has been suggested, that handles variable-length data as strings for a GPU database [3]. Redfox is a run time framework for database queries that runs all TPC-H queries [60]. Strings are stored in different tables with each string table storing different length strings. Significant speedups were observed on queries containing string matching predicates, but the matching algorithm was not specified. Each thread performed matching on independent strings resulting in branch and memory divergence. We consider alternative ways to map GPU threads to input strings and also alternative GPU device string memory layouts. MapD is a Big Analytics platform taking into advantage the high parallelism and fast memory of GPU processors and uses indexes to search tweet contents [39]. Other state-of-the art GPU database systems are Parsteam [23] and SQream [43], but there is not available documentation on their string matching approach.

String processing has also been identified as an application that can be accelerated using vector processors [66]. GPU processors have different threads of the same warp executing in each step rather than different data lanes. We believe that our approach can also be adapted for SIMD processors.

2.4 Thread divergence on GPUs

Previous work on GPU database processing has resulted in significant speedups [4,16,27]. A subset of SQLite commands has been implemented on GPUs. Thread divergence affected performance when the operators of a query plan were fused in a single CUDA kernel as opposed to different serially executed kernels. Compiling queries to multiple kernels and writing intermediate results in the global memory can reduce divergence [50].

Software-based solutions can eliminate thread divergence by thread data remapping [62], which has been extended to remove additional code irregularities, e.g., irregular memory references [63]. A way to perform thread data remapping is by reordering the data. Iteration delaying and branch distribution reduce the performance impact of branch divergence on programs [21]. Loop-splitting reduces register pressure caused by thread divergence [10]. Software-based branch predication has been suggested for AMD GPUs to reduce branch penalties [53]. Another software-based solution reduces thread and memory divergence for applications that can tolerate errors [48].

Hardware extensions have also been suggested for thread divergence elimination for both GPU and SIMD processors: dynamic regrouping of threads into new warps [20], adjusting the SIMD width based on branch or memory latency divergence [38] and identifying reconvergence points for threads [15]. Different warp scheduling policies have been suggested to reduce resource utilization on GPUs [40].

3 Problem description

We assume an analytical database where we store the string columns in the GPU global memory before executing the query workload. We initially consider LIKE predicates of the form

```
LIKE '%string%'
```

which are answerable directly using string search. We then consider predicates of the form

```
LIKE '%string1%string2%...%stringk%'
```

which may be answered by an algorithm that applies a sequence of string matching steps. Omitting the leading or trailing "%" symbol requires the target string to appear at the beginning or end of the string respectively; dealing with such cases is a straightforward generalization of the methods presented here. Finally we will consider disjunctive pattern matching predicates, of the form:

```
LIKE '%(string1|string2|...|stringk)%'
```

for which multi-pattern matching algorithms such as AC are relevant. Strictly speaking this syntax is not valid in SQL, but equivalent functionality can be provided by the REGEXP_LIKE predicate [57].

It is common in databases to adopt dictionary compression to reduce the size of the string columns. However, dictionaries might grow in size for string columns of high cardinality or for longer strings. In addition even if data is compressed, it has to be decompressed to apply the matching predicate. In that case the column will be decompressed and the matching predicate will be applied to the dictionary value fetched from the dictionary.

There are multiple strategies to execute a complete query in a coprocessing environment with CPUs and GPUs. Suppose there are some integer and some string filters in the query, as for example in Q16 of TPC-H. The output of the algorithm is expected to be the row-identifiers of rows matching all of the specified conditions. Figure 1 shows three execution strategies, and the data exchanged between the CPU and the GPU. The query optimizer is responsible for choosing a suitable strategy given selectivity and cost estimates for the various conditions.

Queries that require the matching string itself in the SELECT clause are handled outside the matching process.

4 Our approach

In this section we analyze our suggested string processing optimization techniques and the performance implications of combining different techniques to implement more complex string matching methods.

4.1 Addressing divergence

Size grouping Thread divergence can occur when some threads have reached the end of their string, while others in the same warp have not. The threads that have finished sit idle until all of the remaining threads in the warp complete their task, which may involve scanning a long string segment. For this reason, we make sure that all threads in a warp operate on strings of similar length.

Splitting Even with equal-length strings, thread divergence can occur when some threads have found a match and sit idle, while others in the same warp have not. To address this kind of divergence, we consider the option of *splitting* strings. For example, we could split each string into two equally sized pieces and initially perform string matching on the first half. Those strings for which a match has been found have their

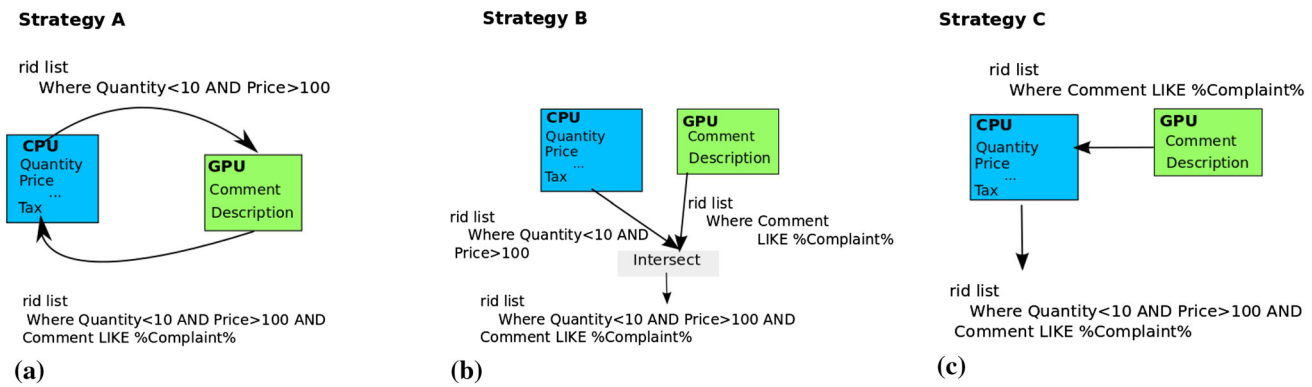

Fig. 1 Strategies for CPU–GPU interaction during query execution containing two integer and one string filter. **a** Sends row-ids satisfying the two integer filters to the GPU; the GPU then searches only the listed strings for matches, and returns the corresponding row-ids. **b** Executes the integer filters on the CPU and in parallel the string matching condition on all rows in the GPU; the results are intersected to compute the final result. **c** This strategy first applies the string filter on the GPU and sends the matching row-ids to the CPU which can use this information to process less data

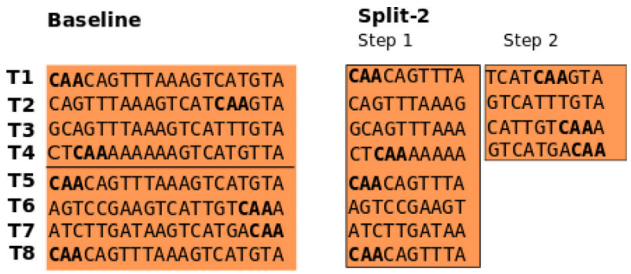

Fig. 2 This diagram shows the execution for the baseline method and a method that splits the strings into two pieces and uses two warps, each having four threads. The search pattern is "CAA." In the baseline method threads T1, T4, T5, and T8 are idle for more than half of the string length. In the split version, only one warp is needed to process the second half of the input strings

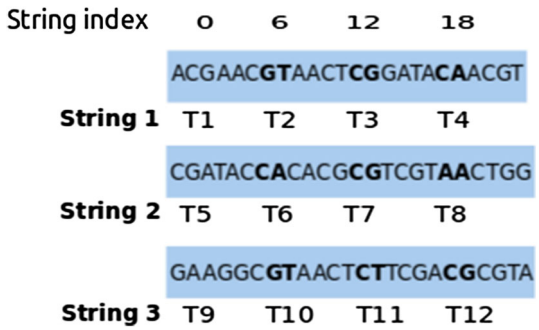

Fig. 3 Execution of Seg 6–4 parallelism method for a pattern of three characters. The gap is of 6 bytes, the input strings are 24 characters, and there are four threads processing the same string. Thread 1 will start matching from offset 0, thread 2 from offset 6 and so on. The *letters in bold* indicate the extra characters that have to be processed because of the boundary overhead

IDs added to the answer set, while strings for which a match has not been found have their IDs added to a pending set. In a second pass, we process the second halves of strings in the pending set in parallel, without ever looking at the second halves of strings that matched in the first pass. In this way, threads that match early do not have to sit idle for the entire string length. Splitting is illustrated in Fig. 2.

We implement the multiple steps of split optimization in a single GPU function. In all steps but the last, the threads store in the shared memory the record-ids of the strings that have not matched the pattern. Different threads might process different parts of a string in distinct steps. We divide the shared memory in different buffers, one for each warp. For selective conditions, the expected performance gain is higher because more strings are filtered in each step, and there is lower shared memory writing cost.

Splitting may introduce a small overhead at split boundaries. Searching each piece of the string (except the last) might need to process up to $m-1$ characters from the following piece, where m is the length of the pattern. When the following piece is later searched, some of those $m-1$ characters may be processed again. For KMP (but not BM), we avoid this boundary overhead by recording the search state for the last character of the previous string piece.

4.2 Addressing cache and memory pressure

The observations of Sect. 1.1.2 suggest that allowing all threads to do independent work (the method of choice for CPUs) will not be sustainable on GPUs. Efficient GPU string matching will need some form of *locality* so that data from each cache line is useful for multiple threads.

Register usage To reduce the number of memory loads from cache, we try to read as large a unit of string data as possible into GPU registers. Using an extended load instruction, it is possible to read up to 16 bytes of data at once into two registers. Depending on the memory access pattern of the string matching algorithm, this approach can reduce the traffic from the L2 cache, although it does not reduce the cache footprint needed by the collection of threads.

Segmentation One way to enhance locality is to allocate multiple threads from a warp to different segments of the same string in such a way that the interval between consecutive threads on a string is less than a cache line. In that way, multiple segments of the string can be examined concurrently, and the cache misses needed to read the string are amortized over multiple threads. For example, if four threads span a cache line, then the cache footprint would be reduced by a factor of 4. Threads processing the same string can coordinate and finish searching when one of them finds a match. For the segmented method, we will use the term "Seg-k-t" to denote a method using t threads per input string with a gap between threads of k bytes. Figure 3 demonstrates the execution of the Seg-6-4 method for string of 24 characters. Typically we choose larger gaps, but we show a small gap for the sake of simplicity.

While locality is improved in such a scheme, there is a boundary overhead for each segment. To detect patterns that span segment boundaries for a pattern of m characters all threads but the last will have to process the following $m-1$ characters. Because segments are processed in parallel, we miss opportunities to skip over the initial characters of each segment.

Pivoting So far, we have implicitly assumed that strings are stored contiguously. However, in database applications preprocessing the data (e.g., by rearranging the string layout) is a viable option to maximize workload performance. We propose pivoting[1] the strings in the GPU global memory to eliminate cache thrashing and to facilitate coalesced loads. We divide the string into equally sized pieces and store each

[1] To the best of our knowledge, string pivoting has been suggested only for fixed 1-byte or 4-byte units [25,49,65], without studying the impact on cache behavior.

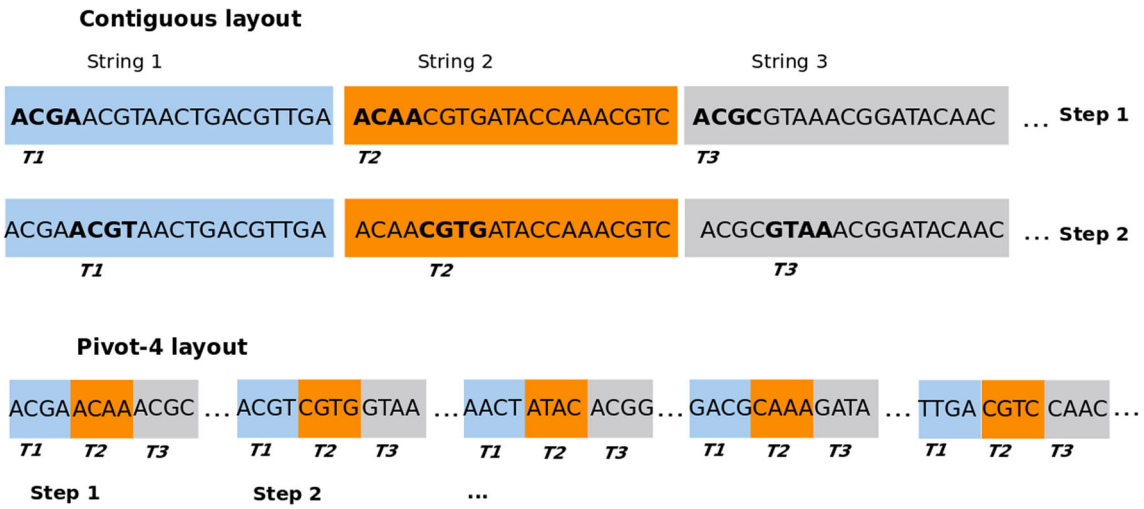

Fig. 4 This diagram shows the contiguous and pivoted layout for 20-character strings and pivoted piece of 4-characters. It also shows how different threads advance over the strings in 4-byte pieces. For the contiguous layout in the first step T1 will process string characters in addresses 0–3, T2 characters 20–23 and T3 characters 40–43. For the pivoted layout in the first step T1 will process characters 0–3, T2 characters 4–7 and T3 characters 8–11 so different memory requests can be combined in one global memory transaction

piece as if it was a different column of a column store. For example, suppose we choose a piece size of 4 bytes for strings 128 bytes long. The first "column" would contain the initial 4 characters of each string. The second "column" would contain the second 4 characters, and so on up to "column" 32. Figure 4 shows the contiguous and the pivoted layout for 20-character strings and 4-character pivoted pieces. In the "Appendix," we show additional examples of the execution of the pivoted and segmentation methods. In case strings have different lengths, we need to pad them to have the same number of "columns." Threads would be assigned one per string, with threads in a warp processing consecutive strings. When they start, warps would read the initial 4 bytes from a contiguous group of strings, which is an ideal memory access pattern because (a) it is coalesced, and (b) it supports good spatial locality and a reduced cache footprint. As string matching progresses, threads may progress at different rates depending on the matching algorithm and pattern. We evaluate the effect on performance for different pivot widths.

In the case of pivoted strings, algorithms for which threads progress at different rates are vulnerable to a phenomenon we call *memory divergence*. If threads in a warp are accessing mostly different offsets within the pivoted layout, we will have lost most of the performance benefits of pivoting. As a result, there is an implicit advantage to methods that proceed in lockstep (or close to it) through the strings, even if it means shorter advances on each step. In fact, we show how one can slightly change the implementation of KMP to guarantee lockstep processing, perhaps at the expense of increased thread divergence. Memory divergence has not previously been studied for string matching applications.

Self-pivoting For long strings, we suggest the following solution combining the best features of pivoting and segmentation: Segment the string into equally sized pieces and store the segments of each string in a pivoted fashion. For example, suppose we segment a 4-KB string into 32 segments of 128 bytes each. With a pivot width of 4 bytes, the first cache line would contain the first four bytes from each segment; the second cache line would contain bytes 5 though 8 for each segment, and so on. Threads in a warp process different segments of a single string in parallel, and achieve locality because the data needed by those threads at any point in time is concentrated in just a few cache lines (one if the threads manage to proceed in lock-step). Threads in a group coordinate and terminate when any of them locates the pattern.

Self-pivoting requires minimal padding, because each string can be divided into segments that differ in size by at most one symbol. Unlike pivoting across strings, self-pivoting retains good coalesced memory access behavior even for filtered string access (Strategy A in Fig. 1).

4.3 Combining optimization techniques

In Table 1, we summarize the different optimization techniques we evaluate in this paper. Our suggested techniques can be combined to implement more complex methods. Table 2 summarizes the techniques used by the matching methods evaluated in our experiments. We always implement size grouping and use registers to reduce the global memory traffic. When using pivoting the size of the used registers is limited to the pivoting width: For example, in Pivot-4 method, we load in 4-byte registers four characters

Table 1 This table summarizes the advantages and drawbacks of each string matching optimization

	Independent	Segmented	Split	Pivot	Split + pivot	Self-pivot (long strings)
+Thread efficiency	Low	High	High	Low	High	High
+L2 efficiency	Low	High	Low	High	High	High
−Boundary overhead	No	Yes	KMP:no/BM:yes	No	No	Yes

One or more optimization techniques can be combined to implement more complex string matching methods

from the input string. We cannot use registers larger than the pivot width because this would require multiple loads from non-contiguous memory locations.

Splitting can be combined with segmentation: In each step, each piece of the string can be processed by multiple threads as in the segmented method. Splitting can also be combined with pivoting, although it does not lead to better performance compared to plain pivoting. Splitting incurs the cost of writing the intermediate results in the shared memory but this additional cost is justified by the reduced string processing cost. However, the benefits of splitting are not as significant for pivoted layouts since the global memory cost of string processing is reduced. For the conventional layout, there is a cache miss in each thread memory access while for the pivoted layout multiple thread accesses can be combined in a few global memory transactions. Also, the benefits of pivoting will be reduced in the subsequent steps because the strings being processed might not be contiguous. Self-pivoting is designed for longer strings by combining the best features of the segmented and pivoted methods.

5 Algorithmic details

As discussed in Sect. 4, and in Table 1, there are several classes of parallelization that we will consider. For the "Seg-k-t" method, we require t to evenly divide the thread warp size. If the string size was not perfectly divisible by the gap size, there would be a load imbalance in the last iteration.

For the split method, we will use the term "Split-k" to denote a method with each string evenly split into k pieces. For the pivoting method, we will use the term "Pivot-k" to denote a method with each string stored in a pivoted fashion, k bytes per pivoted "column." If a configuration is not labeled as "segmented," then it is assumed that one thread is assigned per string. If a configuration is not labeled as "pivoted," it is assumed that strings are stored contiguously.

5.1 L2 cache analysis

Table 3 shows the L2 cache footprint for different parallelization methods.

Segmentation and pivoting can reduce the cache footprint below the L2 capacity. Splitting does not impact the cache footprint, but could still be effective at reducing other forms of latency. Below $k = 4$, the ideal footprint of Pivot-k does not change because a single cache line can accommodate 32 threads' worth of 4-byte data. The numbers for pivoting labeled "ideal" reflect the footprint at the start of matching, and if matching proceeds at precisely the same rate in each thread of a warp. The row labeled "divergent" corresponds to a situation in which threads progress at the sufficiently different pace that all threads are touching different cache lines. In practice, partial thread divergence may lead to intermediate cache footprint sizes. We will examine this issue in more depth in Sects. 5.2 and 5.3. The cache footprint of self-pivoting is the same as regular pivoting.

To validate the analysis in Table 3 and Fig. 5 shows the number of L2 misses for varying segment size as measured

Table 2 This table shows the set of techniques used in the methods we evaluate

Method	Techniques
String matching methods	
Segmentation	Segmentation, size grouping, register usage
Splitting	Splitting, size grouping, register Usage
Pivoting	Pivoting, size grouping, register usage
Self-pivoting	Self-pivoting, size grouping, register usage

Table 3 L2 cache footprint for different methods, on a GPU like the K40 with 15 SMs and a 128-byte cache line

Method	Cache footprint (MB)
Independent	**3.75**
Seg-64	**1.875**
Seg-32	0.938
Seg-16	0.469
Split-k	**3.75**
Pivot-2 (ideal)	0.117
Pivot-4 (ideal)	0.117
Pivot-8 (ideal)	0.234
Pivot-16 (ideal)	0.469
Pivot-32 (ideal)	0.938
Pivot-k (divergent)	**3.75**

Note that the K40 has an L2 capacity of 1.5 MB; bold values in the table exceed this capacity

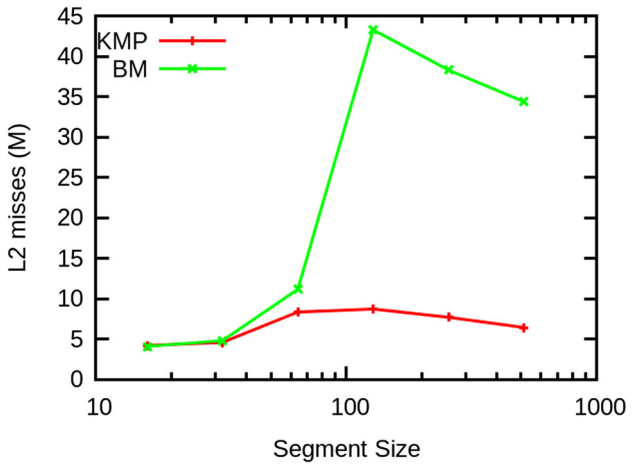

Fig. 5 KMP and BM Seg-*k*-*t* L2 misses for varying segment size on a dataset of 512K strings with $t = 4$. The string length is 1024 bytes, the pattern size is four bytes and the selectivity 0.9

on Nvidia K40. For both algorithms L2 misses increase significantly at a segment size of 64, as predicted in Table 3.

5.2 Optimizing memory divergence

GPUs are sensitive to changes of source code even if the time complexity remains the same. However, alternative implementations of KMP have not been studied. We show below two alternative ways to code the KMP algorithm [14], assuming that the algorithm returns "true" after finding the first match. In both of these code fragments, `s` is the string being searched, `p` is the pattern, `nxt` is the partial match table, `m` is the length of the pattern, and `n` is the length of the string. For simplicity, we omit optimizations such as reading many characters at a time from memory into registers.

```
KMP_basic(char *s, char *p,
    int *nxt, int m, int n){
int i=0; int j=0; while (i<n) {
    if (p[j]==s[i]) { //Divergent branch
       j++;
       i++;
       if (j==m)
           return true;
    } else {
       if (j != 0)   //Divergent branch
          j = nxt[j];
       else
          i++;
    }
}
KMP_step(char *s, char *p,
    int *nxt, int m, int n){
int j=0; for (int i=0;i<n;i++) {
    while (j>=0 && p[j]!=s[i]) //Divergent
    loop
          j=nxt[j];
    j++;
    if (j==m)
          return true;
}
```

In the `KMP_basic` algorithm, if the while loop is executed in parallel by many threads, each thread either advances through the string or advances the offset within the pattern. Strings that advance the offset in the pattern will "fall behind" the other threads that advance though the string. Unlike `KMP_basic`, a thread-parallel execution of the `for` loop in `KMP_step` waits for those threads that need to advance far enough through the pattern to advance to the next character. We also propose a hybrid method `KMP_hybrid` that has an outer loop like `KMP_step`, to ensure memory alignment, with an inner loop like `KMP_basic` to reduce thread divergence. The granularity of the outer loop should correspond to the pivot width in a pivoted layout; the following code segment corresponds to a pivot width of 4. We assume a little endian architecture.

```
KMP_hybrid(char *s, char *p,
int *nxt, int m, int n){
int j=0;
for(int i=0;i<n;i+=4){
    unsigned t=*((unsigned *)s+(i>>2));
    for(int k=0;k<4;){
       if (p[j]==((char)t)) {//Divergent
       branch
              t>>=8; k++; j++;
              if (j==m)
                 return true;
         } else {
             if (j != 0)    //Divergent
       branch
                 j = nxt[j];
            else {
                t>>=8; k++;
}}}}
```

In KMP_Basic and KMP_Hybrid, the input is advanced in two cases: (1) if the comparison of the pattern to the input succeeds or (2) if j equals zero. If the input is not advanced, then the jump table is accessed. Based on that observation we can remove some of the branches in the code to increase the string matching performance. We show below the version of KMP_Hybrid, with the eliminated branches:

```
KMP_hybrid(char *s, char *p,
 int *nxt, int m, int n){
int j=0; for(int i=0;i<n && j<m;i+=4){
    unsigned t=*((unsigned *)s+(i>>2));
    for(int k=0;k<4 && j<m;){
          const int cmp=p[j]==((char)t);
```

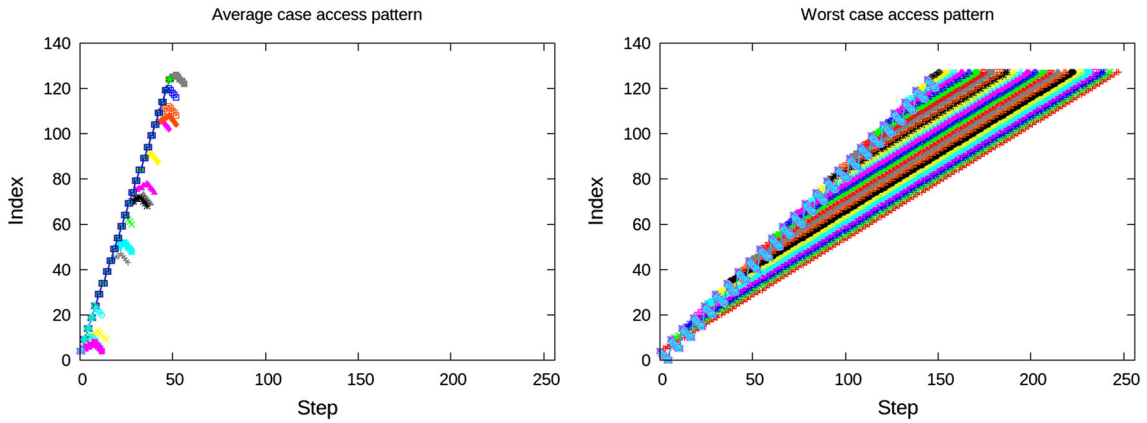

Fig. 6 Memory access pattern for BM on an average-case dataset and an adversarially generated input maximizing memory divergence. There are 32 curves in each subfigure showing the index accessed by each thread of a warp. The input strings are 128 characters/bytes long

```
            const int cmp2=cmp || j==0;
            if(cmp2)  {
                    t>>=8;
            } else {
                    j = nxt[j];
            }
            j+=cmp;
            k+=cmp2;
    }
    return j==m;
}
```

Removing the remaining branches resulted in inferior performance. In Fig. 20 of the Appendix we show an execution example of KMP_Hybrid for a pivoted layout of the input, illustrating the string indices accessed in different iterations of the matching process.

5.3 Worst-case memory access patterns

We are interested in constructing worst-case inputs for the BM and KMP algorithms. We must avoid performance "surprises" on these inputs. The worst case for a single instance of the KMP algorithm is searching for a pattern with m repetitions of a single character in an input string which is a repetition of the following segment: The first $m-1$ characters match, but the mth does not. For the warp level adversarial dataset the input strings are constructed in such way that the first thread in the warp matches the first four characters and fails, the second thread fails after the first eight character comparisons and so on. This will result in memory divergence for Pivot-KMP_basic: Some of threads will fall behind, accessing different column segments. KMP_step will have increased thread divergence because threads will each execute a different number of iterations of the internal while loop.

For BM we construct a similar adversarial set of strings as in KMP with the only difference that the matching fails in the leftmost character rather than the rightmost because it compares the characters from right to left. In Fig. 6 we show the memory access pattern of BM for two sets of input strings. There are 32 color bands in each of the subfigures corresponding to the 32 different threads in a warp. Different threads process 32 different input strings, so the different curves show the string index accessed by each of the threads in the warp. The first set of strings is randomly generated. In the second dataset to evaluate the worst-case memory access pattern of BM, the strings are adversarially generated so that the divergence of BM memory access pattern is maximized. Figure 7 shows the memory access pattern of the KMP implementations. We note that different steps have different cost for the three KMP versions and more steps do not imply slower performance. With a pivoted layout, when concurrent threads access locations separated by more than the pivot width, additional cache misses will result.

6 Experiments

GPU performance was measured on two GPUs: an Nvidia Kepler K40, and an Nvidia Tesla C2070. ECC was turned on. L1 caching is off by default on the K40. We turned off L1 caching on the C2070 because we observed that performance improved for bandwidth-bound algorithms as a result of the reduced bandwidth needed from the L2 cache.[2] Table 4 contains the specifications of each GPU. The default machine is the K40, but we show some results for both machines to prove that our techniques and analysis are not specific to one

[2] When the L1 cache is turned on, 128 byte cache lines are sent from the L2. When L1 caching is off, 32 bytes are accessed at a time from the L2.

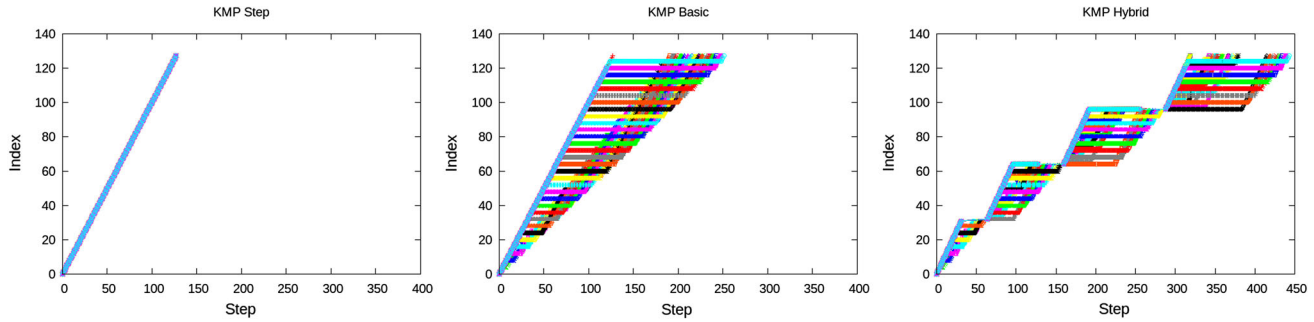

Fig. 7 Memory access pattern for a worst-case input of KMP. The *y*-axis shows the string offset and the *x*-axis the number of each memory access. *Different curves* show the string index accessed by each thread of a warp. For example in `KMP_basic` during step 128, threads access string indexes from 64 to 127. The `KMP_hybrid` chart corresponds to a pivot width of 32. There is a varying number of steps for each 32-byte unit because threads enter the 32-byte units in different initial states. To show that the long-term memory access behavior of KMP_Hybrid stabilizes, we provide the corresponding chart in "Appendix 2" for a longer input string

Table 4 GPUs used in our experiments. Prices were taken on 05/08/2015 from amazon.com

GPU	Tesla C2070	Tesla K40
Memory size	6 GB	12 GB
Frequency (MHz)	1150	745
Bandwidth (GB/s)	144	288
L2 (KB)	768	1536
L2 band. (B/cycle)	384	1024
Cores	448	2880
Shared mem. (B/cycle/bank)	2	4
SMs	14	15
Cost ($)	1250	3100

machine, and to highlight situations where the machine used does matter.

CUDA C code was compiled using nvcc of CUDA toolkit 5.5 using the full optimization level. We also used loop unrolling in the string matching functions to reduce the number of executed instructions. For a more fair comparison against CPUs we use the base clock of the K40 GPU (745 MHz), although we noticed an improvement of 17 % in performance when using the maximum clock frequency (875 MHz).[3] We used a dual-socket Intel E5 2620 CPU to compare against the GPU performance. Each socket has six cores (12 threads) and a power rating of 95 W. We used all 24 threads of the dual-socket machine.

Strings are either random (for average-case performance measurements), constructed as described in Sect. 5.3 (for worst-case measurements) or generated from real-world datasets. String characters in our experiments are one byte so we use the terms character and byte interchangeably. To

[3] The higher frequency is a "boost" frequency that can only be used when there is power and temperature headroom.

Table 5 Workload parameters

	A1	A2	R1	R2	TPC-H
String size	1024	16–16K	500	4.51M	63,1024
Occurrences	0–32	0	N/A	N/A	N/A
Pattern size	1–32	15	10	20/25	N/A
Rows	512K	32K	3.52M	64	512K
Alphabet	26	4	128	4	N/A

achieve a given selectivity, the pattern is inserted into random locations of a suitable number of randomly chosen strings. For some experiments, we also vary the number of times that a pattern appears in a string. Table 5 summarizes the workload parameters for our datasets. Workload A1 reflects a textual search task. A2 reflects a search of an artificial DNA sequence. R1 is a set of wikipedia abstracts as downloaded from DBpedia [13]. For R1 we chose a set of 10-character patterns randomly selected from the input. All patterns correspond to low selectivity (<0.01). R2 searches the genome sequence of Yersinia pestis [61] which we have replicated to achieve full parallelism to simulate a realistic workload searching the genomes of many bacteria (one string per organism) for the given pattern. For the TPC-H workload, we use the data generator provided by the TPC-H benchmark. We ran the subquery of Q16, which we will be referring to as Q16_1:

```
select s_suppkey from supplier
where s_comment like '%Customer%Complaints%'
```

We use scale factor of 53 during data generation for the Supplier column to produce 512K rows. A fixed number of rows is randomly selected from the TPC-H data generator to contain strings matching %Customer%Complaints%. We show the results for both the original s_comment column size and for a wider column size of 1024 characters.

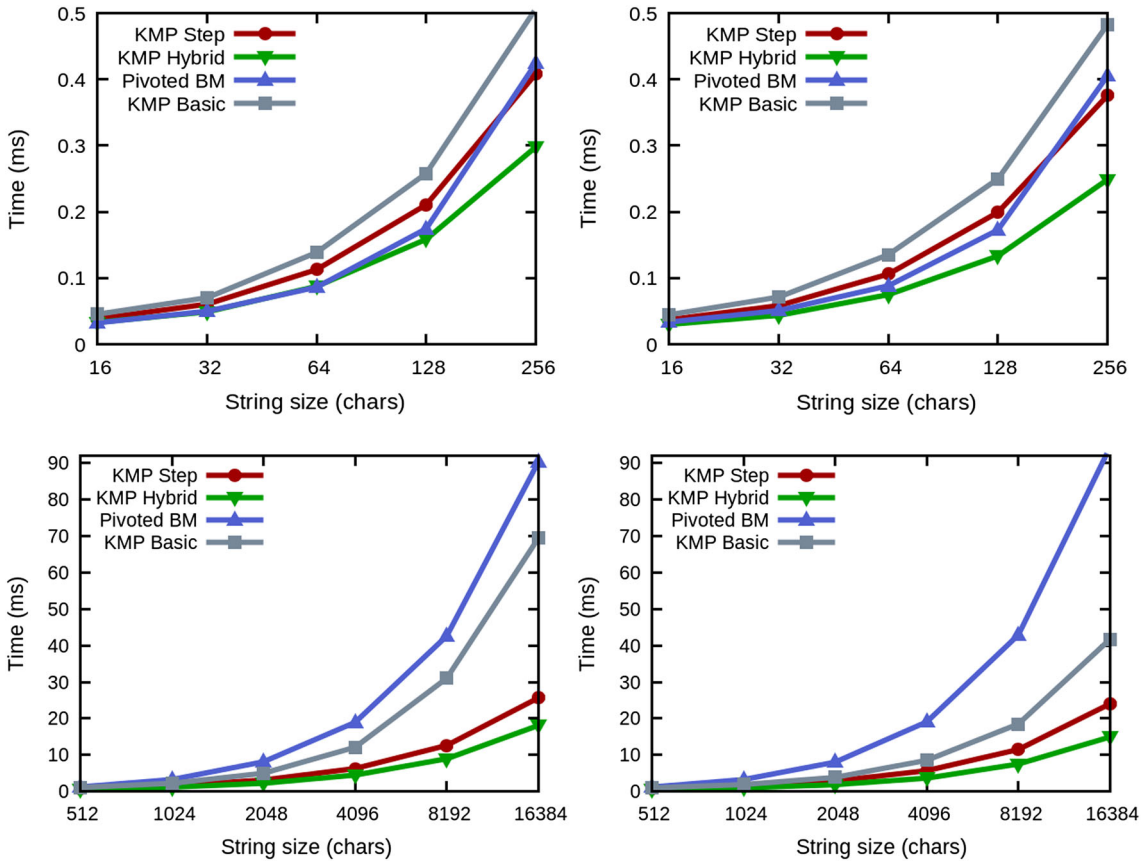

Fig. 8 Performance as a function of string length for Pivot-4 (*left*) and Pivot-8 (*right*) layouts. The *top row* shows the results for shorter strings and the *bottom row* for longer strings. The *x*-axis is logarithmic

Pattern preprocessing is done only once for a pattern so it has no performance impact. We store the jump table in the shared memory in addition to the search pattern. We compare our GPU implementation to the performance of PFAC multi-pattern matching running on the GPU and a multi-threaded CPU implementation using three different CPU implementations for A1.

6.1 String length and pivoting

Figure 8 shows the performance of all three versions of KMP and of BM on workload A2 as a function of the string length, using Pivot-4 and Pivot-8 layouts. The biggest impact of our methods is for longer strings where there is worse memory locality and higher memory divergence.

The top row shows the performance for strings up to 256 characters and the bottom for strings up to 16,384 characters. For strings less than 64 characters, BM has similar performance to KMP-Hybrid. For longer strings the performance of BM and KMP-Basic deteriorates due to memory divergence. Pivot-8 is clearly superior to Pivot-4 for all methods. Figure 9 shows the performance of pivoted KMP and pivoted BM algorithms for varying pivot width on workload A1. We

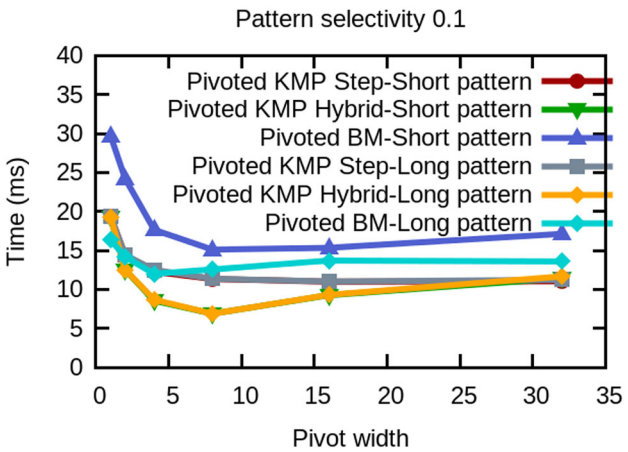

Fig. 9 Time performance on A1 for varying pivoted width for BM and KMP_step and KMP_hybrid

observe that KMP has stable performance regardless of the pivoted width and pattern length, and in general it has superior performance to BM.

We repeated the experiment for workload R1, corresponding to a textual search on a real-world dataset, where the character distribution is non-uniform. In Fig. 10 we show the corresponding results. We observe that the performance dif-

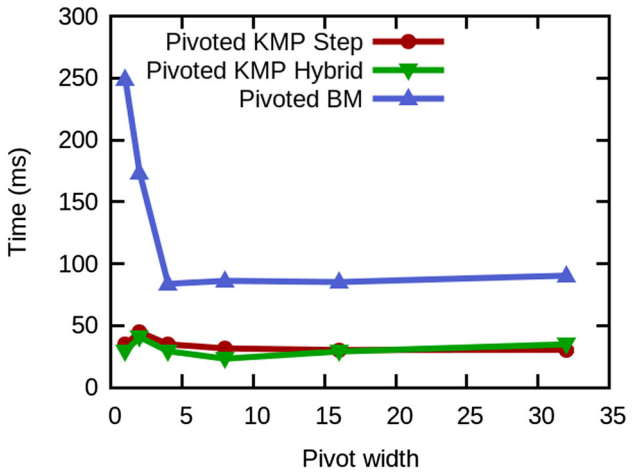

Fig. 10 Time performance on R1 for varying pivoted width for BM, KMP_step, and KMP_hybrid

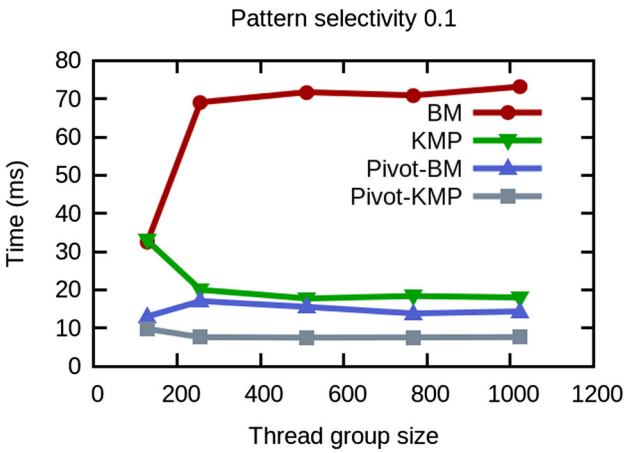

Fig. 11 Performance of KMP and BM for varying group size and input strings of length 1024

ference between KMP methods and BM is higher than the difference for A1. This happens because when some characters occur more frequently than others, as it is the case for real text, this resembles inputs of smaller alphabet size for which BM has less competitive performance.

Taken together, the results of Figs. 9 and 10 suggest that 8 is the best choice for pivoting width. We have repeated the same experiment for shorter strings (64 characters) suggesting the same pivoted width. The results also show that, at least for pivoted layouts, techniques that minimize memory divergence (`KMP_step` and `KMP_hybrid`) are superior. `KMP_hybrid` is better than `KMP_step` in Fig. 8 because thread divergence is reduced. Threads do not have to proceed in lockstep at character granularity, just at 8-character boundaries. From now on, we will use just `KMP_hybrid` for single-pattern matching on pivoted layouts.

We also implemented pivoting itself on the GPU to transform strings from contiguous to pivoted representations. For a pivot width of 8, pivoting ran at 60 GB/s on the K40, which is typically faster than string search.

6.2 Thread group size

Figure 11 shows the performance of (Pivot-8 and unpivoted) KMP and BM for varying group size on A1 for a 4-character pattern. `KMP_basic` is used for unpivoted and `KMP_hybrid` for Pivot-8. The optimal thread group size for unpivoted BM is 128 because it has larger cache footprint when there are more concurrent threads. For the Seg-*k*-*t* BM method, the L2 footprint is reduced so the optimal group size is larger; we use the optimal thread group size for each method.

6.3 Alphabet size

Figure 12 shows the performance of the pivoted and unpivoted methods for varying alphabet size.

We observe that BM methods, both pivoted and unpivoted depend on the alphabet size: For larger alphabets the performance improves, because the skipping distance increases. The performance improvement is more observable for pivoted BM because for large alphabets the memory divergence decreases and threads in a warp may skip entire pivoted pieces. KMP also slightly improves: The memory access cost remains the same because memory accesses are better coordinated (KMP-Hybrid and KMP-Step), but there are fewer partial matches so there are fewer jumps over the pattern table. The overall performance variation is less observable because the pattern and the jump table are stored in the shared memory which is faster. Finally, we observe that pivoted KMP-Hybrid is the faster matching method even for larger alphabets.

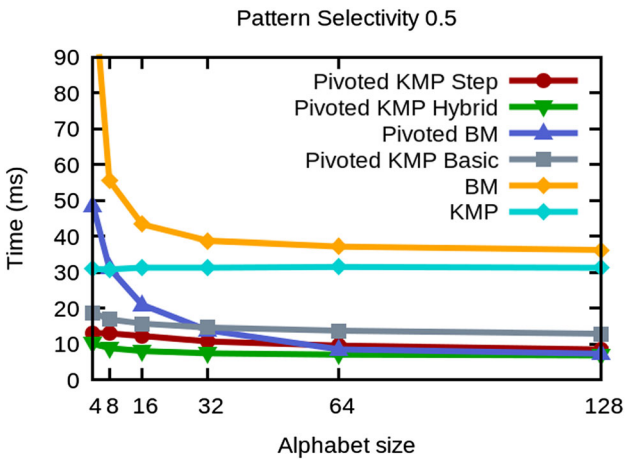

Fig. 12 Time performance of pivoted and unpivoted methods for varying alphabet size and an 8-character pattern

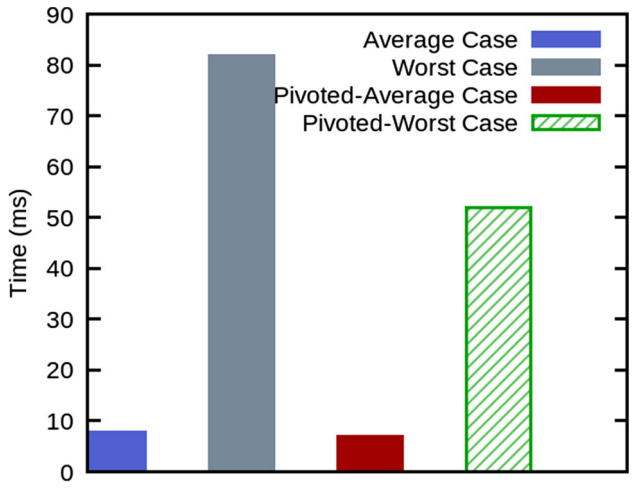

Fig. 13 Performance of BM for average and adversarial input

6.4 Worst-case performance

Figure 13 shows the performance of row-wise and Pivoted-8 BM for an average and worst-case input set of strings on workload A1 for a 32-character pattern.

The performance difference is about an order of magnitude. We also observe pivoting helps the performance of BM. This is because subset of the threads particularly at the beginning of the search process "jump" to the same pivoted column. However depending on the pattern length and the pivoted width the performance improvement might be less significant, as we noted in Fig. 9. The performance difference for all versions KMP between the average and worst case is less than 2x making it a more robust choice.

The fastest strategy is Strategy A because the predicates applied on the CPU are "cheap" to evaluate. For more expensive predicates, we expect Strategy B to be more competitive.

6.5 Co-processing performance

Figure 14 shows the performance of Pivot-KMP, Self-Pivot-KMP and unpivoted KMP for two different access patterns. String matching is applied on a sparse and a dense subset of records rather than the full table. These patterns correspond to queries where one or multiple predicates have already been applied. The list of the record identifiers are communicated from the CPU to GPU. In the pivoted methods the dense rid-list performance is hardly affected because we're still doing just a small number of memory transactions (2 rather than 1, since the selectivity is 0.5). As expected, the unpivoted method's performance is not significantly affected. The performance for the unpivoted method is still lower than the performance of the pivoted methods on sparse record lists.

For the sparse case, threads in a warp will access different pivoted pieces so we are doing 32 transactions rather than 1 and pivoting performance degrades significantly. However, self-pivoting maintains most of the performance benefits even when a sparse subset of the strings is processed from a query.

Table 6 shows the performance of the three CPU–GPU co-processing strategies for the following query:

```
Select count(*) From Orders
Where o_comment not like '%special%packages%'
 AND  o_orderstatus='F' AND o_totalprice>50000
```

Table 6 Time performance of the three alternative CPU–GPU interaction strategies (Fig. 1)

Strategy A	Strategy B	Strategy C
0.85 ms	1.02 ms	1.05 ms

Strategy A is the fastest because the CPU predicates are "cheap" to evaluate. For more expensive CPU conditions other strategies could be the most efficient

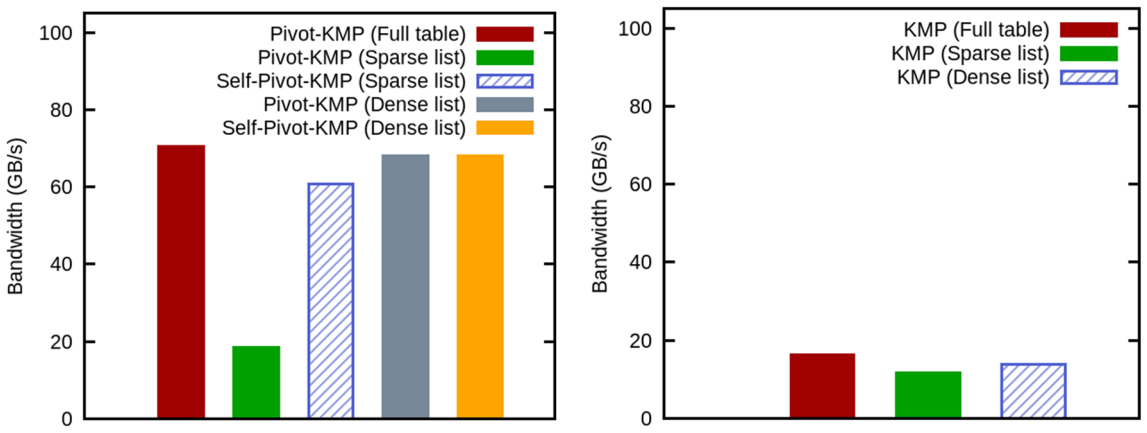

Fig. 14 Bandwidth of string matching for sparse and dense record lists for pivoted (*left*) and unpivoted (*right*) KMP methods. The sparse record list corresponds to 1% selectivity while the dense for 50% selectivity

The dataset is generated by the TPC-H generator. The string column o_comment is stored in the GPU. The other two columns are stored in the CPU RAM.

6.6 Segmentation

Figure 15 shows the performance of Seg-k-t for $t = 16$ for increasing pattern length on workload A1. For BM and patterns shorter than 16 characters the best method is Seg-32 because of the reduced cache misses, but for patterns longer than 16 characters the most efficient method is having threads matching independent strings. For BM the performance is initially increasing because BM can skip larger parts of the input, but it starts decreasing because of the increased effect of boundary overhead. For KMP the optimal parallelism method is independent threads because we already reduce the memory cost by prefetching multiple characters into the registers. We also observe that KMP performance seems to depend less on the pattern length. When $t > 8$, we observed that the choice of t did not significantly affect the performance of string matching.

6.7 Thread divergence

We show the performance of split optimization on the K40 and C2070 for workload A1 in Fig. 16.

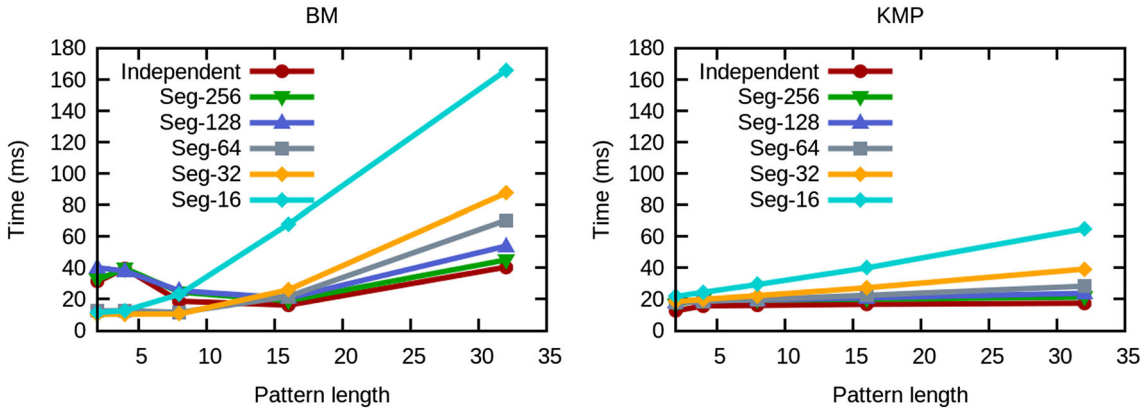

Fig. 15 Performance for increasing pattern length for Independent and Seg-k-t (8) implementations

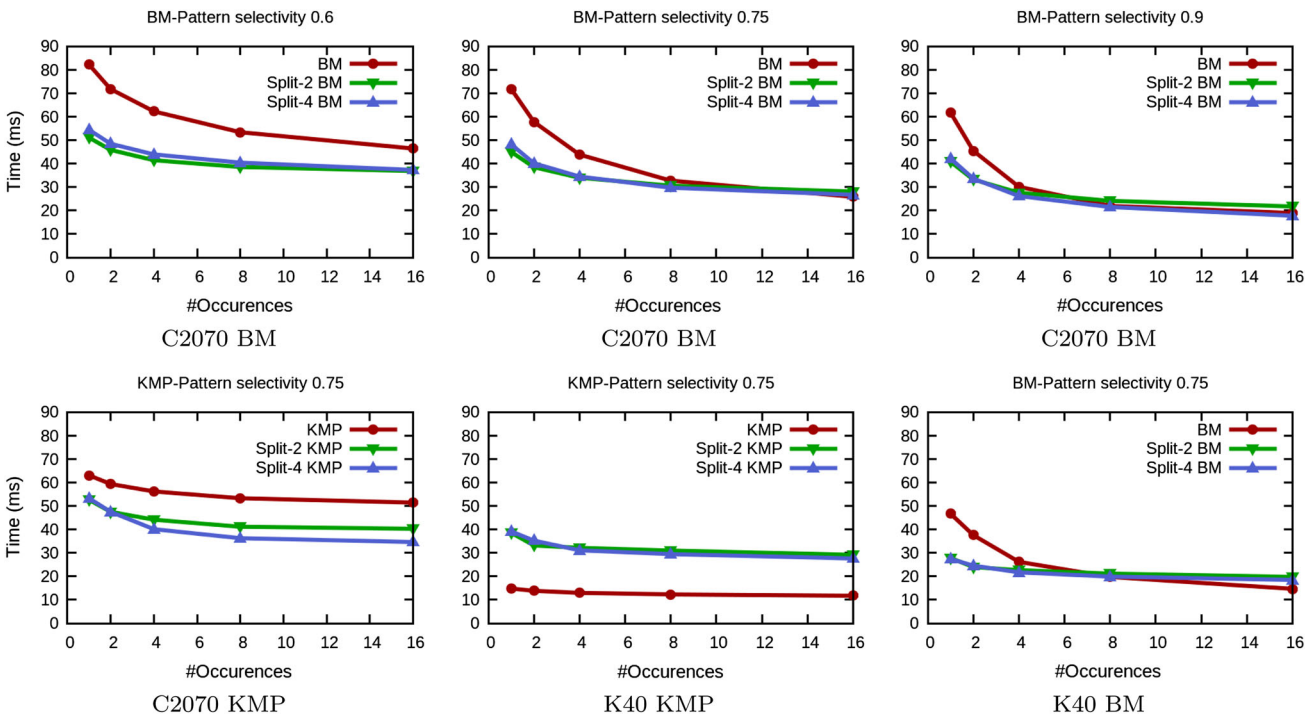

Fig. 16 Performance of Split-k optimization on BM for varying number of string occurrences in the input. The first subfigure is for selectivity 0.6, the second for 0.75, and the last for selectivity 0.9

Figure 16 compares the performance of independent string search to split-k for different k and for a varying number of repetitions of the pattern in the strings. Each subfigure in the top row corresponds to BM on the C2070 with a different operator selectivity: the first for selectivity 0.6, the second for 0.75, and the last for selectivity 0.9. For less selective conditions, there is lower overhead of writing in shared memory in the intermediate steps and more potential for bypassing computation, on the other hand the cost of string search is lower because more threads finish searching early. For selectivity 0.9 the speedup is up to 40 %, while for selectivity 0.6 the speedup is 65 %.

The right chart in the bottom row of Fig. 16 shows corresponding results for BM at selectivity 0.75 on the K40 GPU, which are qualitatively similar to the C2070. The remaining two charts show the KMP performance on the two GPUs under various Split-k configurations. Surprisingly, the GPUs differ significantly in the relative performance. While split-k optimizations are helpful on the C2070, the same is not true on the K40. The reasons for this difference are subtle, and illustrate the complexities of optimizing GPU performance.

Consider the performance parameters of the two devices in Table 4. If an algorithm is memory bound, it can expect at best a 2X (288/144) improvement moving from the C2070 to the K40. Similarly, if the algorithm is bound by the L2 bandwidth, an improvement of 1.7X $((1024 * 745)/(384 * 1150))$ is possible. If the algorithm is bound by accesses to shared memory, then a factor of 2 improvement is possible. On the other hand, an algorithm that is not memory bound has the potential for a 4.2X $((2880 * 745)/(448 * 1150))$ speedup due to the much larger number of cores on the K40. The K40 can issue two instructions per warp so depending on how well the pipeline slots can be filled, the speedup potential is up to 8.4X. The KMP algorithms with the split optimization are shared memory bound, because the intermediates used by this optimization need to be written to and read from shared memory. On the other hand, the KMP algorithm without the split optimization has been engineered to be cache resident and to avoid memory performance pitfalls. As a result it can achieve a speedup closer to the 4.2X potential speedup. BM, being memory bound due to the high cache miss rate, cannot achieve the same speedup.

6.8 Comparison with CPUs

Investing in the use of a GPU depends on more factors than just the raw query performance. We compare GPUs and CPUs holistically in terms of raw performance, performance per $ and energy consumption for the subquery Q16_1. In addition to comparing CPU and a GPU processors, we also show the estimated performance of a combined system that uses both CPUs and a K40 GPU in the following way: It initially executes different instances of the same query on the CPUs and the GPU. We run a query per CPU hardware thread and another query instance on the CPU delegating its work to the GPU. The CPU threads use for each query the fastest CPU matching library among the algorithms that have linear time complexity. Whenever a query completes execution we start a new query to ensure that all processors are kept busy. This process is executed for five seconds and in the end we compute the average bandwidth and energy consumption per query. Typically, the throughput of the combined system is the sum of the measured query throughput of all the processors when executing independently. There is only a small (5–10 %) overhead when the CPU thread delegating its workload to the GPU has to copy back the query results.

We use all 24 CPU hardware threads and set the frequency policy to maximum.[4] We evaluate two different popular CPU libraries, RE2, and Boost and we show the performance of the fastest of the two. For RE2 each thread is independent operating on a separate re2 object, so the pattern is compiled once for each thread. For the Boost library we use the object-based interface for each method and the pattern is again compiled once for each CPU thread. We also implement a CPU matching method based on the CMPISTRI SSE instruction for patterns that fit in a SSE register. In the worst case this method has $O(n \times m)$ time complexity but in the special case of short patterns it has good performance. This method scans the string in segments of 16 bytes (the size of the SSE register) until a full match is found. If a partial match is found the CMPISTRI instruction returns an offset to the beginning of the partial match; we then load the next 16 bytes and check whether the following segment matches the remaining subpattern.

Tables 7 and 8 summarize our CPU versus GPU comparison for the two different column sizes of the TPC-H workload. We focus on longer strings so we use the results of Table 7 in our later analysis but the results also favor GPUs for the shorter string experiments.

The power rating of the K40 is 235W. For the two CPUs the aggregate power rating is 190W. The energy consumption does not include the RAM memory energy consumption while for the GPU the rating includes all GPU components. Tables 7 and 8 show that the estimated GPU energy consumption is at least 1.85x less than the CPU energy consumption for long and medium length strings. For long strings the energy consumed for the Q16_1 execution is 2.35 J for the CPU implementation based on CMPISTRI and 1.27 J for the GPU, so the GPU consumes 1.85x times less the energy than the CPU even without including the energy consumed from the CPU RAM. We also note the power efficiency for the latest GPUs seems to be improving: The K80 processor has 1.65x the memory bandwidth of the K40 (480 GB/s) and

[4] This the maximum base frequency, not a boosted frequency that depends on power or temperature headroom.

Table 7 This table summarizes the CPU versus GPU comparison for Q16_1 and string size 1024 bytes

	CPU (RE2)	CPU (CMPISTRI)	GPU	CPU + GPU
Price ($)	952	952	3100	4052
Query performance (GB/s)	40.75	43.1	98.7	138.7
Energy consumed (J)	2.49	2.35	1.27	1.78
Performance/$	42.8	45.28	31.89	34.23

The GPU query performance includes the time to transfer the query results back to the CPU over PCI/E

Table 8 This table summarizes the CPU versus GPU comparison for Q16_1 and original string size (63 bytes)

	CPU (Boost BM)	CPU (CMPISTRI)	GPU	CPU + GPU
Price ($)	952	952	3100	4052
Query performance (GB/s)	20.87	28.36	80.56	100.2
Energy consumed (J)	0.29	0.21	0.1	0.14
Performance/$	21.92	29.8	26.01	26.87

The GPU query performance includes the time to transfer the query results back to the CPU over PCI/E

1.73x the number of cores while the power rating is only 1.27x of that of the K40 used in our experiments [54].

We compute the (MB/s)/($) rate to quantify the performance per $. The price of the K40 is $3100. The price for each CPU processor is $421 and to that cost we must add the price for 12 GB DDR3 RAM, which is $110, so the total price for the CPU system is $952.[5] Using the above costs the performance/$ rate for the CPU is 45.28 and for the GPU 31.89 for long strings (Table 7). However, for medium length strings GPUs have similar performance per $ ratio: 26.01 versus 29.8 for the CPU system (Table 8). If we limit our comparison against CPU algorithms with linear worst-time complexity, the GPU has actually better performance per $. We found similar results for a Q13 TPC-H subquery, operating on shorter strings (average string length 47 characters). The results for this subquery can be found in the "Appendix".

Figure 17 shows the performance of the CPU libraries and the GPU pivoted string matching methods for varying pattern selectivity. The performance difference between GPU and CMPISTRI is less significant for the synthetic dataset but Pivot-KMP method's performance is still 20–45 % faster for any selectivity value. We also notice that RE2 has worse performance for the artificially generated dataset: This happens because RE2 can advance fast over a string if the first character of the pattern being searched does not appear in the input string. For example, when searching for the wildcard predicate '%Customer%Complaints' in Q16_1, character 'C' occurs only in about a third of the input strings so RE2 performance is relatively high. For the artificially generated dataset, typically all 26 characters appear in each of the input strings so this optimization cannot be applied.

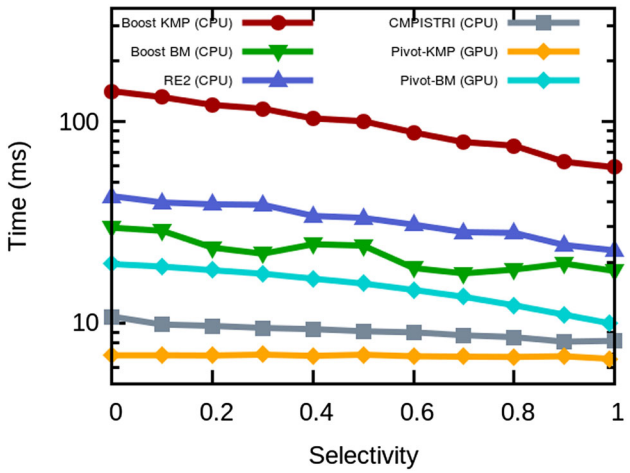

Fig. 17 Time performance of CPU and GPU string matching for A1 and a 8-character pattern. The *y*-axis is logarithmic

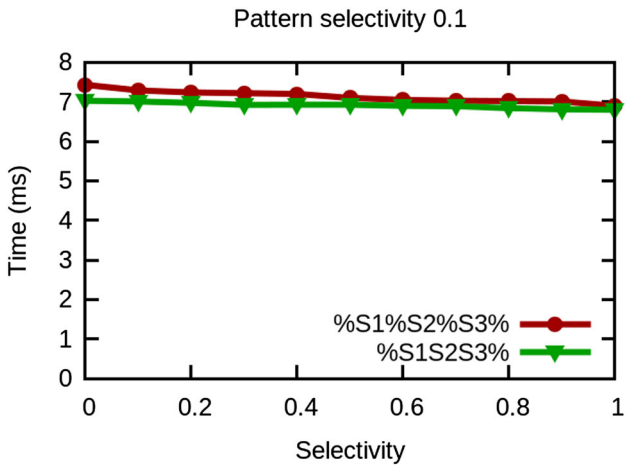

Fig. 18 Time performance of KMP for two different predicates: '%S1%S2%S3%' versus '%S1S2S3%'

[5] Prices were taken on 05/08/2015 from amazon.com.

6.9 More general queries

Figure 18 compares the performance of KMP for a LIKE `'%S1S2S3%'` query and a LIKE `'%S1%S2%S3%'` query. In the second query all string searches are implemented in one kernel. All three patterns are preprocessed and stored in an array of patterns. Each time a substring is found the array index is incremented and an overall match is found if all substrings are matched. KMP performance degrades slightly for queries involving a sequence of patterns because of the extra bookkeeping cost.

Figure 19 shows the performance of pivoted AC for varying selectivity, implementing the following LIKE predicate: LIKE `'%(S1|S2|S3)%'`. We compare the performance of AC to the performance of: (a) KMP for the same query using split optimization, where each step matches one of the three patterns consecutively; (b) single-pattern `KMP_hybrid` that performs a simpler query; (c) `KMP_step` matching all three patterns by keeping a separate state for each pattern; and (d) PFAC library using all optimizations [35]. We observe that AC, when storing the DFA in the shared memory, does not have significant overhead compared to single-pattern KMP, and that it is the superior method for multi-pattern matching. PFAC loads the first row in the shared memory which we have not implemented in our AC implementation using texture memory. Our AC implementation using shared memory is slightly faster than PFAC for a small number of patterns. The most significant advantage is that its performance does not degrade dramatically for adversarial inputs: PFAC performance for worst-case inputs degrades by up to 20x [35], while our performance degrades by less than 2x.

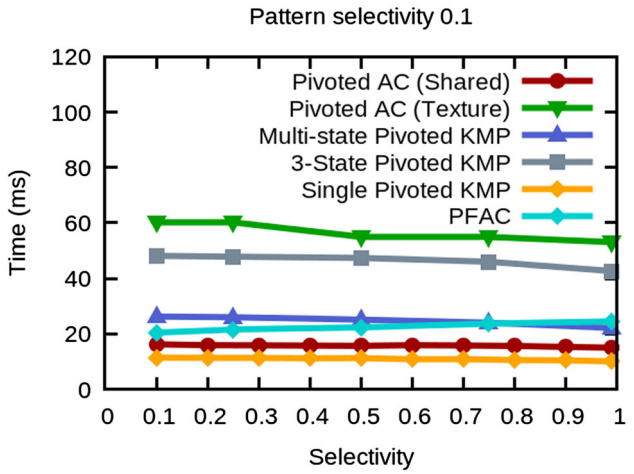

Fig. 19 Performance of pivoted AC, pivoted KMP against PFAC for varying selectivity

Table 9 compares published performance numbers and the measured performance of our GPU implementation on workload R2. This table shows that we significantly outperform prior methods, even when adjusting for differences in GPU capabilities. Self-pivot KMP has similar performance to Pivot-KMP with the additional benefit that it can coalesce memory even when GPU operates on a subset of the table given by an rid-list. Table 10 summarizes which algorithm-parallelism combination is optimal for different query parameters under workload A1. We observe that Pivoted KMP is the best algorithm because it has the best performance in eight of nine cases. The competing algorithm is Pivot-BM for low selectivity queries and longer patterns: BM can effectively skip over large segments of text. For the worst-case inputs KMP has always superior performance making it the best choice overall.

Table 9 This table compares the performance (GB/s) of our matching methods to the published performance of other GPU libraries (bottom three lines)

Library	Machine	Cores	Bandwidth (GB/s)	Perf. ($m = 20$) (GB/s)	Perf. ($m = 25$) (GB/s)	Cost ($)[a]	(Avg. perf.) / $
Self-Pivot-KMP	Tesla C2070	448	144	15.21	15.02	1250	12.1
Self-Pivot-KMP	Kepler K40	2880	288	55.37	55.13	3100	17.82
Pivot-KMP	Tesla C2070	448	144	14.93	15.27	1250	12.08
Pivot-KMP	Kepler K40	2880	288	54.53	55.06	3100	17.67
PFAC	Kepler K40	2880	288	26.3	26.3	3100	8.5
KMP [5]	Tesla K20m	2496	208	19.76	N/A	2685	7.39
BMH [30]	GTX 280	240	141.7	N/A	1.2	120	10

Performance per $ is computed by the average performance for the two different pattern sizes (20 and 25) over the GPU price.[a] To produce numbers more friendly to the reader we used the performance in MB/s to compute this rate

[a] Prices were taken on 05/08/2015 from amazon.com

Table 10 Best average-case performance for workload A1 for different query parameters

	Short pat. (4)	Medium pat. (16)	Long pat. (64 chars)
Low sel. (0.05)	Pivot-KMP	Pivot-KMP	Pivot-BM
Medium sel. (0.5)	Pivot-KMP	Pivot-KMP	Pivot-KMP
High sel. (0.9)	Pivot-KMP	Pivot-KMP	Pivot-KMP

7 Conclusions and future work

We advocate using the GPU as a co-processor for string matching and KMP as the preferred string matching algorithm. String matching is an interesting application to evaluate the effect of thread and memory divergence on GPU kernel performance which has a fair number of different dimensions. We suggest multiple parallelism methods for string matching and study the performance of the state-of-the art algorithms on two different GPUs. We analyze alternative string layouts in the global memory and suggest different performance optimizations for string matching algorithms. Our solution optimizes string search by selecting the right parallelism granularity and string layout for the different algorithms. The performance of our proposed methods exceeds that of other CPU and GPU implementations. In future work we are interested in SIMD-ifying string matching algorithms for CPUs and also string matching on compressed data to enable fitting larger datasets into the GPU memory.

Acknowledgments This material is based upon work supported by National Science Foundation Grant IIS-1218222, an IBM Ph.D. Fellowship, an Onassis Foundation Scholarship, and by an equipment gift from Nvidia Corporation. We would like to thank MSc student Le Chang for the initial implementation of the code.

Appendix 1

Here, we show two additional examples of the segmentation and pivoting methods. The pattern being searched is 'ATG' and the input strings are 16 characters long.

Figure 20 shows the execution of the Pivot-4 KMP-Hybrid method. Different threads are processing different strings.

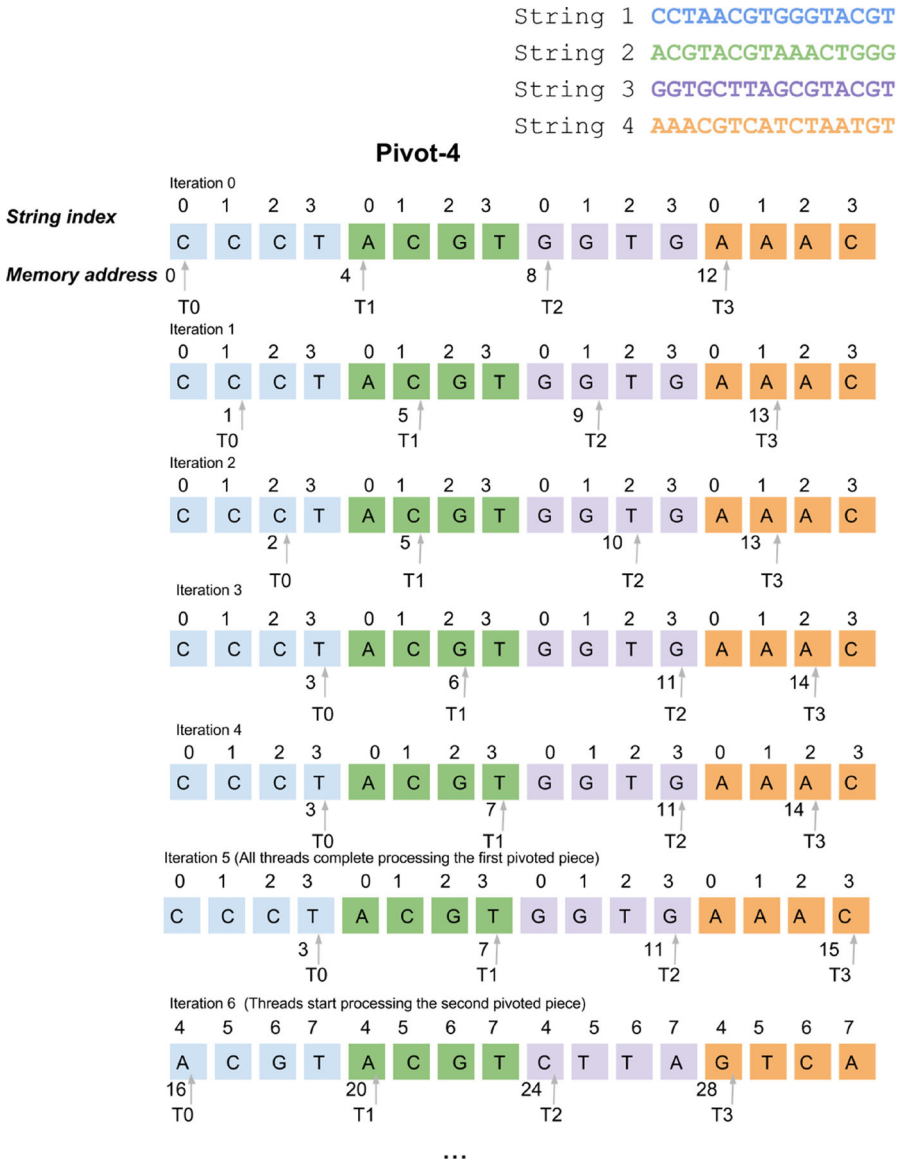

Fig. 20 Execution of Pivot-4 method for 'ATG' pattern using the KMP-Hybrid string matching method and 4 GPU threads. For each iteration, the numbers in the *top line* show the string indexes and the numbers in the *bottom line* the memory addresses. There are partial matches in some strings so in the third iteration some threads will scan different indexes of the first pivoted piece. Specifically, T1 and T3 fall behind after the second iteration because they have to shift the pattern. When all threads finish processing the first pivoted piece (iteration 6), threads synchronize and scan the first character of the second pivoted piece

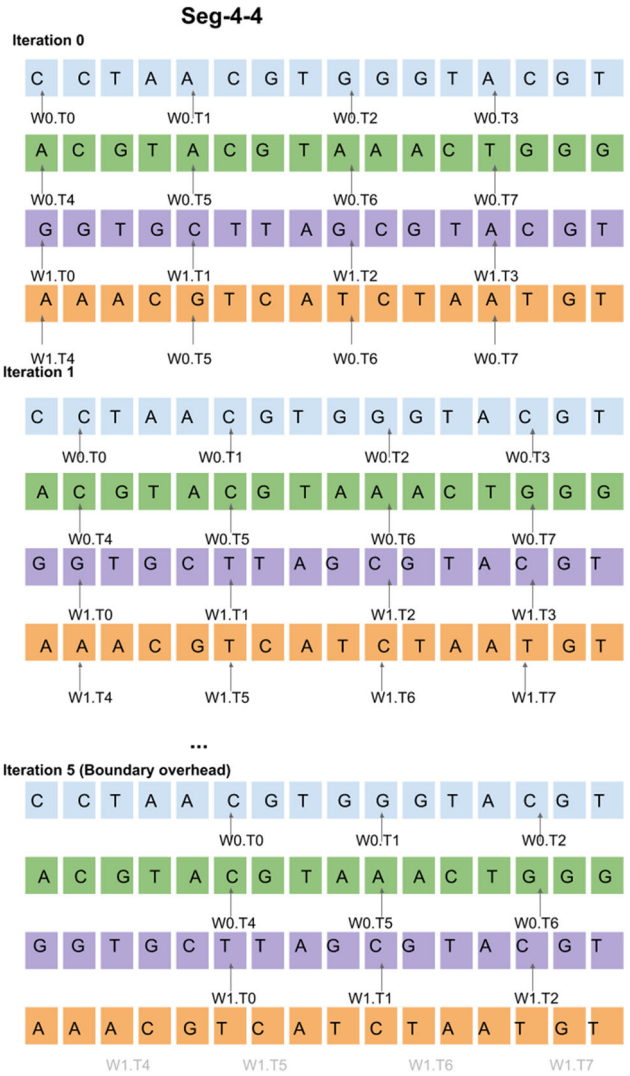

Fig. 21 Execution of Seg-4-4 method for 'ATG' pattern. Each warp has 8 threads for reasons of simplicity. The input strings are 16 characters long, and four threads process each input string. The search pattern 'ATG' is present in the last string processed by the last four threads of W1. Each thread will process six characters to include the boundary search cost between different segments. After the third iteration threads T5-8 of the second warp (W1) are inactive because T7 of the second warp has located the pattern

Threads start scanning the first character of the first pivoted piece. In the internal loop of KMP-Hybrid the input is advanced either if the comparison of the current pattern character to the input succeeds or if it fails on the first character of the pattern ($j == 0$). T1, T3 match their input in first iteration to the pattern. In the second iteration the comparison fails for both T1 and T3 so they have to consult the nxt table to shift the pattern. In the third iteration T1 and T3 will compare the second input character again after shifting the pattern. Threads synchronize again in the seventh iteration after all threads process the first piece and scan the first character of the second pivoted piece.

Figure 21 shows the execution of Seg-4-4 method: The segment size is four characters and four threads are processing concurrently each input string and two warps in total process the four input strings. Each thread, but the last in each group, is processing six characters in total to locate pattern occurrences that might occur between different segments.

Appendix 2

Figure 22 shows the long-term worse-case memory access behavior of KMP_Hybrid for a string of 512 characters. The behavior stabilizes after 128 characters (the length of the search pattern used in this worse-case example).

Appendix 3

Table 11 shows the results for the following subquery of TPC-H Q13.

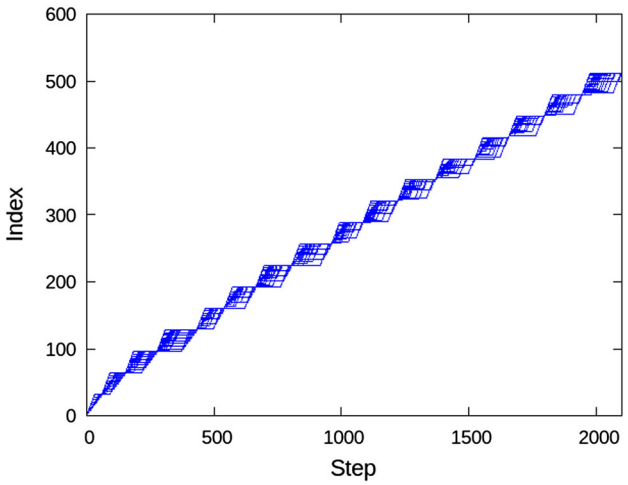

Fig. 22 Worst-case memory access behavior of KMP_Hybrid for a string of 512 characters

Table 11 This table summarizes the CPU versus GPU comparison for Q13_1

	CPU (Boost BM)	CPU (CMPISTRI)	GPU	CPU + GPU
Price ($)	952	952	3100	4052
Query performance (GB/s)	16.5	26.6	83.05	98.29
Energy consumed (J)	0.27	0.17	0.07	0.1
Performance/$	17.33	27.9	26.79	24.25

The GPU query performance includes the time to transfer the query results back to the CPU over PCI/E

```
Q13_1 SELECT count(o_orderkey) FROM Orders
WHERE  o_comment not like %special%packages%
```

GPU has more than 3x times better query performance and 2.42x less the energy consumption.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Apostolico, A., Giancarlo, R.: The Boyer Moore Galil string searching strategies revisited. SIAM J. Comput. **15**(1), 98–105 (1986). doi:10.1137/0215007
3. Bakkum, P., Chakradhar, S.: Efficient Data Management for GPU Databases. http://hgpu.org/?p=7180 (2012)
4. Bakkum, P., Skadron, K.: Accelerating SQL database operations on a GPU with CUDA. In: GPGPU (2010). doi:10.1145/1735688.1735706
5. Bellekens, X., Andonovic, I., Atkinson, R., Renfrew, C., Kirkham, T.: Investigation of GPU-based pattern matching. In: The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013) (2013)
6. Bhargava, A., Kondrak, G.: Multiple word alignment with profile hidden Markov models. In: ACL, Companion Volume: Student Research Workshop and Doctoral Consortium, Association for Computational Linguistics, Boulder, Colorado, pp. 43–48. http://www.aclweb.org/anthology/N/N09/N09-3008 (2009)
7. Boost Library. http://www.boost.org/ (2014)
8. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10) (1977). doi:10.1145/359842.359859
9. Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., Saake, G.: GPU-accelerated database systems: survey and open challenges. T Large Scale Data Knowl. Cent. Syst. **15**, 1–35 (2014). doi:10.1007/978-3-662-45761-0_1
10. Carrillo, S., Siegel, J., Li, X.: A control-structure splitting optimization for GPGPU. In: CF '09, pp. 147–150 (2009). doi:10.1145/1531743.1531766
11. Cascarano, N., Rolando, P., Risso, F., Sisto, R.: iNFAnt: NFA pattern matching on GPGPU devices. SIGCOMM Comput. Commun. Rev. **40**(5), 20–26 (2010). doi:10.1145/1880153.1880157
12. Crochemore, M., Lecroq, T.: Pattern-matching and text-compression algorithms. ACM Comput. Surv. **28**(1), 39–41 (1996). doi:10.1145/234313.234331
13. Dbpedia. http://wiki.dbpedia.org/Downloads2014 (2014)
14. Design and Analysis of Algorithms Lecture Notes. http://www.ics.uci.edu/~eppstein/161/960227.html (1996)
15. Diamos, G., Ashbaugh, B., Maiyuran, S., Kerr, A., Wu, H., Yalamanchili, S.: SIMD re-convergence at thread frontiers. In: MICRO (2011). doi:10.1145/2155620.2155676
16. Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N.K., Luo, Q., Sander, P.V. GPUQP: query co-processing using graphics processors. In: SIGMOD, pp. 1061–1063 (2007)
17. Farivar, R., Kharbanda, H., Venkataraman, S., Campbell, R.: An algorithm for fast edit distance computation on GPUs. In: Innovative Parallel Computing (InPar), pp. 1–9 (2012). doi:10.1109/InPar.6339593
18. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005). doi:10.1145/1082036.1082039
19. Fisk, M., Varghese, G.: Applying fast string matching to intrusion detection. Tech. rep., http://woozle.org/~mfisk/papers/setmatch-raid (2004)
20. Fung, W.W.L., Sham, I., Yuan, G., Aamodt, T.M. Dynamic warp formation and scheduling for efficient GPU control flow. In: MICRO (2007). doi:10.1109/MICRO.2007.12
21. Han, T.D., Abdelrahman, T.S.: Reducing branch divergence in GPU programs. In: GPGPU, pp. 3:1–3:8 (2011). doi:10.1145/1964179.1964184
22. Horspool, R.N.: Practical fast searching in strings. Softw. Pract. Exp. **10**(6), 501–506 (1980). doi:10.1002/spe.4380100608
23. Hummel, M.: Parstream—A Parallel Database on GPUs. http://www.nvidia.com/content/gtc-2010/pdfs/4004a_gtc2010 (2010)
24. Intel 64 and IA-32 Architectures Software Developer's Manual. http://download.intel.com/design/processor/manuals/253665 (2011)
25. Iorio, F., van Lunteren, J.: Fast pattern matching on the cell broadband engine, workshop on cell systems and applications. In: The 35th International Symposium on Computer Architecture (ISCA), Beijing, China (2008)
26. Jacob, N., Brodley, C.: Offloading IDS computation to the GPU. In: ACSAC, pp. 371–380 (2006). doi:10.1109/ACSAC.2006.35
27. Kaldewey, T., Lohman, G.M., Mueller, R., Volk, P.B.: GPU join processing revisited. In: DaMoN (2012)
28. Karkkainen, J., Ukkonen, E.: Sparse suffix trees. In: Cai, J.Y., Wong, C. (eds.) Computing and Combinatorics, LCNS, vol. 1090, pp. 219–230 (1996). doi:10.1007/3-540-61332-3_155
29. Knuth, D.E., Morris Jr, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
30. Kouzinopoulos, C., Margaritis, K.: String matching on a multicore GPU using CUDA. In: PCI, pp. 14–18 (2009). doi:10.1109/PCI.2009.47
31. Li, J., Chen, S., Li, Y.: The fast evaluation of hidden Markov models on GPU. In: IEEE International Conference on Intelligent Computing and Intelligent Systems, 2009 (ICIS 2009), vol. 4, pp. 426–430 (2009)
32. Ligowski, L., Rudnicki, W.: An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: IEEE International Symposium on Parallel Distributed Processing, 2009 (IPDPS 2009), pp. 1–8 (2009). doi:10.1109/IPDPS.2009.5160931
33. Lin, K.J., Huang, Y.H., Lin, C.Y.: Efficient parallel knuth-morris-pratt algorithm for multi-GPUs with CUDA. In: Pan, J.S., Yang, C.N., Lin, C.C. (eds.) Advances in Intelligent Systems and Applications, vol. 21, pp. 543–552 (2013). doi:10.1007/978-3-642-35473-1_54
34. Lin, C.H., Tsai, S.Y., Liu, C.H., Chang, S.C., Shyu, J.M.: Accelerating string matching using multi-threaded algorithm on GPU. In: GLOBECOM, pp. 1–5 (2010). doi:10.1109/GLOCOM.2010.5683320
35. Lin, C.H., Liu, C.H., Chien, L.S., Chang, S.C.: Accelerating pattern matching using a novel parallel algorithm on GPUs. IEEE Trans. Comput. **62**(10), 1906–1916 (2013). doi:10.1109/TC.2012.254
36. Liu, Y., Maskell, D., Schmidt, B.: CUDASW++: optimizing Smith–Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Res. Notes **2**(1), 73 (2009). doi:10.1186/1756-0500-2-73
37. Marziale III, L., Richard, G.G., Roussev, V.: Massive threading: using GPUs to increase the performance of digital forensics tools. Digit. Investig. **4**, 73–81 (2007). doi:10.1016/j.diin.2007.06.014
38. Meng, J., Tarjan, D., Skadron, K.: Dynamic warp subdivision for integrated branch and memory divergence tolerance. SIGARCH Comput. Archit. News **38**(3), 235–246 (2010). doi:10.1145/1816038.1815992
39. Mostak, T., Graham, T.: Map-D Data Redefined. http://on-demand.gputechconf.com/gtc/2014/webinar/gtc-express-map-d-webinar (2014)
40. Narasiman, V., Shebanow, M., Lee, C.J., Miftakhutdinov, R., Mutlu, O., Patt, Y.N.: Improving GPU performance via large warps

and two-level warp scheduling. In: MICRO, pp. 308–317 (2011). doi:10.1145/2155620.2155656
41. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (2001). doi:10.1145/375360.375365
42. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. **48**(3), 443–453 (1970). doi:10.1016/0022-2836(70)90057-4
43. Netzer, O.: Getting Big Data Done on a GPU-Based Database. http://on-demand.gputechconf.com/gtc/2014/presentations/S4644-big-data-gpu-based-database (2014)
44. Pirk, H., Manegold, S., Kersten, M.: Waste not...; efficient co-processing of relational data. In: 2014 IEEE 30th International Conference on Data Engineering (ICDE), pp. 508–519 (2014). doi:10.1109/ICDE.2014.6816677
45. Pyrgiotis, T., Kouzinopoulos, C., Margaritis, K.: Parallel implementation of the Wu–Manber algorithm using the OpenCL framework. Artif. Intell. Appl. Innov. **382**, 576–583 (2012). doi:10.1007/978-3-642-33412-2_59
46. Rauhe, H., Dees, J., Sattler, K.U., Faerber, F.: Multi-level parallel query execution framework for CPU and GPU. In: Catania, B., Guerrini, G., Pokorny, J. (eds.) Advances in Databases and Information Systems, Lecture Notes in Computer Science, vol. 8133, pp. 330–343. Springer, Berlin (2013). doi:10.1007/978-3-642-40683-6_25
47. Re2 Regular Expression Library. http://code.google.com/p/re2/ (2014)
48. Sartori, J., Kumar, R.: Branch and data herding: reducing control and memory divergence for error-tolerant GPU applications. TMM **15**(2), 279–290 (2013). doi:10.1109/TMM.2012.2232647
49. Scarpazza, D.P., Villa, O., Petrini, F.: Peak-performance DFA-based string matching on the Cell processor. In: IEEE International on Parallel and Distributed Processing Symposium, 2007 (IPDPS 2007). IEEE, pp. 1–8 (2007)
50. Sitaridi, E.A., Ross, K.A.: Optimizing select conditions on GPUs. In: Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN'13). ACM, New York, NY, USA, pp. 4:1–4:8 (2013). doi:10.1145/2485278.2485282
51. Smith, T., Waterman, M.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981). doi:10.1016/0022-2836(81)90087-5
52. Sunday, D.M.: A very fast substring search algorithm. Commun. ACM **33**(8), 132–142 (1990). doi:10.1145/79173.79184
53. Taylor, R., Li, X.: Software-based branch predication for AMD GPUs. SIGARCH Comput. Archit. News **38**(4), 66–72 (2011). doi:10.1145/1926367.1926379
54. Tesla K80 GPU Accelerator. http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05 (2015)
55. Tian, Y., Tata, S., Hankins, R.A., Patel, J.M.: Practical methods for constructing suffix trees. VLDB J. **14**(3), 281–299 (2005). doi:10.1007/s00778-005-0154-8
56. TPC-H Benchmark. http://www.tpc.org/tpch/ (2014)
57. Using Regular Expressions in Oracle Database. http://docs.oracle.com/cd/B19306_01/appdev.102/b14251/adfns_regexp.htm (2014)
58. Vasiliadis, G., Polychronakis, M., Ioannidis, S.: Parallelization and characterization of pattern matching using GPUs. In: IISWC, pp. 216–225 (2011). doi:10.1109/IISWC.2011.6114181
59. Weiner, P.: Linear pattern matching algorithms. In: Swat, IEEE Computer Society, pp. 1–11 (1973). doi:10.1109/SWAT.1973.13
60. Wu, H., Diamos, Gr., Sheard, T., Aref, M., Baxter, S., Garland, M., Yalamanchili, S.: Red Fox: an execution environment for relational query processing on GPUs. In: International Symposium on Code Generation and Optimization (CGO) (2014)
61. Yersinia Pestis Chromosome. ftp://ftp.sanger.ac.uk/pub/project/pathogens/yp/Yp.dna (2001)
62. Zhang, E.Z., Jiang, Y., Guo, Z., Shen, X.: Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In: ICS (2010). doi:10.1145/1810085.1810104
63. Zhang, E.Z., Jiang, Y., Guo, Z., Tian, K., Shen, X.: On-the-fly elimination of dynamic irregularities for GPU computing. In: ASPLOS (2011). doi:10.1145/1950365.1950408
64. Zha, X., Sahni, S.: GPU-to-GPU and host-to-host multipattern string matching on a GPU. IEEE Trans. Comput. **62**(6), 1156–1169 (2013). doi:10.1109/TC.2012.61
65. Zu, Y., Yang, M., Xu, Z., Wang, L., Tian, X., Peng, K., Dong, Q.: GPU-based NFA implementation for memory efficient high speed regular expression matching. PPoPP (2012). doi:10.1145/2145816.2145833
66. Zukowski, M.: Balancing Vectorized Query Execution with Bandwidth-Optimized Storage. PhD thesis, Universiteit van Amsterdam (2009)