Parallel Approaches to the String Matching Problem on the GPU

Saman Ashkiani
University of California, Davis
sashkiani@ucdavis.edu

Nina Amenta
University of California, Davis
amenta@cs.ucdavis.edu

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

ABSTRACT

We design a family of parallel algorithms and GPU implementations for the exact string matching problem, based on Rabin-Karp (RK) randomized string matching. We describe and analyze three primary parallel approaches to binary string matching: cooperative (CRK), divide-and-conquer (DRK), and a novel hybrid of both (HRK). The CRK is most effective for large patterns (>8K characters), while the DRK approach is superior for shorter patterns. We then generalize the DRK to support any alphabet size without loss of performance. Our DRK method achieves up to a 64 GB/s processing rate on 8-character patterns from an 8-bit alphabet on an NVIDIA Tesla K40c GPU. We next demonstrate a novel parallel two-stage matching method (DRK-2S), which first skims the text for a smaller subset of the pattern and then verifies all potential matches in parallel. Our DRK-2S method is superior for pattern sizes up to 64k compared to the fastest CPU-based string matching implementations. With an 8-bit alphabet and up to 1k-character patterns, we get a geometric mean speedup of 4.81x against the best CPU methods, and can achieve a processing rate of at least 53 GB/s.

1. INTRODUCTION

Classic PRAM algorithms are proving to be a fertile source of ideas for GPU programs solving fundamental computational problems. But in practice, only certain PRAM algorithms actually perform well on GPUs. The PRAM model and the GPU programming model differ in important ways. Particularly important considerations suggest choosing algorithms where parallel threads perform uniform computation without branching or waiting ("uniformity"); where memory accesses across neighboring threads access neighboring memory locations ("coalescing"); and where the algorithm and memory accesses can take advantage of the computational and memory hierarchy of the modern GPU ("hierarchy"). Although these choices are usually the foremost considerations in GPU *implementations*, as in Schatz and Trapnell [1] and Lin et al. [2], they rarely play an important role in the process of *algorithm design*. In other words, they are considered optimization opportunities for programmers rather than desirable features of the algorithm itself.

In this paper we address this issue through a case study. We consider the exact string matching problem, which is interesting in that it admits several different styles of parallelization. We focus on a classical PRAM algorithm of Karp and Rabin [3]. Forty years of research into serial string matching has resulted in a plethora of elegant serial algorithms that perform very well in practice, but parallel algorithms are less well explored, particularly in the context of newer hierarchical parallel architectures like GPUs. We implement several variants of the Rabin-Karp algorithm (RK) on NVIDIA GPUs using the CUDA programming environment. Our experience leads us to believe that our general results apply to a broader set of hierarchical parallel processors, with either a compute hierarchy (multiple parallel processors, each with multiple parallel cores or functional units), a memory hierarchy (globally shared memory shared by all processors and faster locally shared memory of limited size shared by all cores within a processor), or both. Examples include other GPUs, accelerators such as the Intel Xeon Phi, and parallel multicore CPUs with vector instructions.

We begin with two main approaches to exploiting parallelism: *cooperative* and *divide-and-conquer*. In the former, all threads cooperate in computing a global solution, and in the latter, the main task is divided into smaller pieces, each piece is processed separately, and the results are combined. Rabin-Karp makes a good test case because it can be parallelized in both ways. In RK, the pattern of length m and all text substrings of length m are hashed (e.g., into a positive integer), and then the hashes are compared instead of the strings. Sting matching is inherently amenable to divide-and-conquer; the text can be divided into as many parallel sub-texts as needed, and the only drawback is due to overlap. RK is also amenable to cooperative parallelization, as described in the original 1987 paper, because with a wise choice of hash functions, the text can be hashed using a cooperative algorithm based on scan (aka parallel-prefix) operations. Fast implementation of scan operations has been one of the great successes in GPU software development for general computation. In both cases, the RK algorithm fits well with the GPU model in terms of uniformity, coalescing, and memory hierarchy. The downside of RK is that it always reads the entire text, while the best sequential algorithms are designed to skip as much as possible, so that they work

SPAA '16, July 11–13, 2016, Pacific Grove, CA, USA

ISBN 978-1-4503-4210-0/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2935764.2935800

especially well on "real data", such as natural language text, genomics data, and internet traffic.

We experimented extensively with two different parallel versions of RK, one with the cooperative strategy (CRK), and the other on using a simple divide-and-conquer strategy (DRK). We find that for short patterns (up to about 800 characters), DRK is much faster than leading sequential codes running on an 8-core CPU. This is mostly because of its uniform load balance, relatively cheap operations, and high memory bandwidth due to coalesced memory accesses. The performance gap is especially large for very short patterns (e.g., 8 characters).

We then leverage the high performance of DRK's short-pattern matching to design a two-stage matching algorithm for longer patterns. It does a first round of searching on a small subset of the original pattern, and then groups of threads cooperate to verify all the potential matches. This new GPU-friendly algorithm outperforms the best multi-core sequential codes for patterns of size up to almost 64k characters. This approach opens up some very interesting opportunities for skipping parts of the text and for doing approximate matching.

Our work provides both good news and bad news for parallel algorithm design. The good news is that parallel algorithms that pay attention to memory access patterns and load balancing can lead directly to high-performance GPU implementations. The bad news is that although scan operations can now be implemented very efficiently in the GPU environment, complex scans are still not necessarily competitive with more straightforward parallelization approaches.

We divide this paper into the following parts:

- We propose three major RK-based binary matching algorithms: cooperative, divide-and-conquer, and a combination of both (hybrid). Using both theoretical demonstration and experimental validation, we address the following question: *for a given pattern and text size, what is the best method to use and why?* At a high level, we show that the divide-and-conquer approach is better for smaller patterns, while cooperative is superior for very large patterns (Sections 3–4).

- We provide several optimizations: both in terms of the general algorithm and computations, as well as some hardware specific ones (Section 6).

- We extend our RK-based algorithms to support characters from any general alphabets (Section 7.1).

- Using the divide-and-conquer approach, which is the fastest we have found for processing short patterns, we propose a novel two-stage matching method to process patterns of any size efficiently. In this method, we first *skim* the text for a random small subset of the main pattern, and then *verify* all potential matches efficiently in parallel (Section 7.2).

2. PRIOR WORK

A naive approach to solve the single-pattern scenario is to compare all the possible strings of length m in our text with our pattern ($O(mn)$ time steps). There are several methods proposed so far to improve such quadratic running time. For instance, the KMP algorithm [4] saves work by jumping to the next possible index in case of a mismatch. The Boyer-Moore (BM) algorithm [5] is similar to KMP but starts its comparison from right to left. Some algorithms combine the existing tricks in other algorithms to enhance their performance. For example, the HASHq algorithm [6] hashes (similar to RK) all suffixes of shorter length (e.g., at most of length 8) and performs the matching on them. In case of a mismatch, like KMP, it shifts to the next potential match and like BM, starts comparing from the rightmost characters of each word. Faro and Lecroq's survey provides a comprehensive introduction and comparison between these and other methods [7].

Some of the earliest parallel work was from Galil [8] and Vishkin [9], who proposed a series of parallel string matching methods over a concurrent-read-concurrent-write PRAM memory model. The GPU literature features numerous papers that target related problems, but none of them address the fundamental problem of matching a single non-preprocessed pattern, implemented entirely on the GPU. Both Lin et al. [2] and Zha and Sahni [10] target multiple patterns (which is easier to parallelize) with an approach that leverages offline pattern compilation; the latter paper concludes that CPU implementations are still faster. Schatz and Trapnell used a CPU-compiled suffix tree for their matching [1]. Seamans and Alexander's virus scanner targeted multiple patterns with a two-step approach, but performed the second step on the CPU [11]. Kouzinopoulos and Margaritis [12] implement a brute-force non-preprocessed search and compare it to preprocessed-pattern methods.

3. PRELIMINARIES

In this section we explain some of the basic algorithms and primitive parallel operations. Readers familiar with the graphics processing unit (GPU) terminology (Section 3.1), the serial RK algorithm (Section 3.2), and its parallel formulation (Section 3.3), can skip to Section 4.

In the basic scenario, $X = x_0 \dots x_{m-1}$ is a binary pattern of length m to be found in a binary text $Y = y_0 \dots y_{n-1}$ of length $n \geq m$. If $Y[r] = y_r y_{r+1} \dots y_{r+m-1}$, then our objective is to find all indices r such that $Y[r] = X$ for $0 \leq r < n - m + 1$. We will later generalize our results to arbitrary alphabets.

3.1 Graphics Processing Units (GPUs)

The GPU is a highly parallel processor that features both a computational hierarchy and a memory hierarchy. In this work we use NVIDIA's CUDA programming model and its related terminology [13]; parenthetical values indicate specific values for the NVIDIA Tesla K40c that we use in evaluating this work. NVIDIA GPUs have one or more parallel *cores* (15 in the K40c, called "streaming multiprocessors" or SMs). Within each core, the hardware runs a *warp* of 32 threads together as a single SIMD unit. These GPUs thus exhibit parallelism across cores and within each core. Programmers express GPU programs as parallel threads that are grouped into blocks (virtualized cores), and typical GPU programs are run on many blocks at the same time. The GPU hardware efficiently assigns blocks to cores as cores become available. The memory hierarchy has three levels, ordered from fastest/smallest to slowest/largest: *registers*, local to each thread (up to 1 KB); *locally shared* memory, shared by threads within a block (up to 48 KB); and *globally*

shared memory, available to all threads (12 GB). Bandwidth to globally shared memory is maximized with *coalesced* memory accesses, where nearby threads access nearby memory addresses.

3.2 Serial Rabin-Karp

The main idea of the RK algorithm is to hash the pattern X and all the possible strings of length m in the text Y. The hashes are known as *fingerprints*, and any two strings with the same fingerprint match with high probability. Karp and Rabin proposed two families of hash functions [3]. The first produces integers as its fingerprints: for any $X \in \{0,1\}^m$: $F(X) = 2^{m-1}x_0 + \cdots + 2x_{m-2} + x_{m-1}$. In order to avoid overflow, the result is computed modulo a random prime number $p \in (1, nm^2)$, so that $F_p(X) \stackrel{p}{\equiv} F(x)$. The second family of fingerprints is defined as follows. Define two matrices,

$$\mathbf{K}(0) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{K}(1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}. \tag{1}$$

Then for any binary string $X = x_0 \ldots x_{m-1} \in \{0,1\}^m$, we define its fingerprint to be the product $\mathbf{K}(X) = \mathbf{K}(x_0) \ldots \mathbf{K}(x_{m-1})$. Again to avoid overflows, the result is computed modulo a random prime number $p \in (1, mn^2)$ (i.e., $\mathbf{K}_p(X) \stackrel{p}{\equiv} \mathbf{K}(X)$). Regardless of which fingerprints we use, the RK algorithm can then be summarized as (1) choose a random prime number $p \in (1, mn^2)$; (2) compute all fingerprints (e.g., $\mathbf{F}_p(X)$, and $\mathbf{F}_p(Y[r])$ for $r \in [0, n-m+1)$); and (3) compare: e.g., if $\mathbf{F}_p(Y[r])$ and $\mathbf{F}_p(X)$ are equal, then X and $Y[r]$ are equal with high probability, otherwise there is no match.

Step 2 can be done quite efficiently in a sequential manner [3]. In case we use the first class of fingerprints, $\mathbf{F}_p(Y[r+1])$ can be computed by using $\mathbf{F}_p(Y[r])$:

$$\mathbf{F}_p(Y[r+1]) \stackrel{p}{\equiv} 2\left(\mathbf{F}_p(Y[r]) - 2^{m-1}y_r\right) + y_{r+m+1}. \tag{2}$$

In case of the second class of fingerprints, we compute $\mathbf{K}_p(Y[r+1])$ based on $\mathbf{K}_p(Y[r])$ by a few operations:

$$\mathbf{K}_p(Y[r+1]) \stackrel{p}{\equiv} \mathbf{A}_p(y_r)\mathbf{K}_p(Y[r])\mathbf{K}_p(y_{r+m+1}), \tag{3}$$

where for any binary value $x \in \{0,1\}$, $\mathbf{A}_p(x)$ is defined as the left inverse of $\mathbf{K}_p(x)$ modulo p (i.e., $\mathbf{A}_p(x)\mathbf{K}_p(x) \stackrel{p}{\equiv} \mathbf{I}$ where $\mathbf{I}$ is an identity matrix). For each read text character, equation (2) requires two multiplications (each can be done by bitwise shift operation), two summations, and a modulo operation. Equation (3), on the other hand, requires 16 integer multiplications, 8 summations and 4 modulo operations per text character. As a result, the first class of fingerprints are significantly simpler to compute. We refer to sequential RK with first class of fingerprints as SRK algorithm. On the other hand, the second class of fingerprints can be updated using an associative operator, which can be leveraged in the following parallel formulation.

3.3 Cooperative Rabin-Karp

Karp and Rabin showed that the second class of fingerprints can be computed in parallel [3] using scan operations. It is well known that the *scan operation* takes a vector of input elements $\mathbf{u}$ and an associative binary operator $\oplus$, and returns an output vector $\mathbf{v}$ of the same size as $\mathbf{u}$. In exclusive (resp., inclusive) scan, $v_i = u_0 \oplus \cdots \oplus u_{i-1}$ (resp., 0 to i). A parallel scan on a vector of length n can be done with $O(\log n)$ depth and in $O(n)$ computations. With p processors, it is easy to show that a scan can be computed in $O(n/p + \log p)$ time steps. On the GPU, there are now highly optimized generic scan implementations that take an arbitrary associative operator as an input.

Let $\mathcal{K} = [\mathbf{K}_p(y_i)]_{i=0}^{n-1}$ and $\mathcal{A} = [\mathbf{A}_p(y_j)]_{j=0}^{n-m}$ represent vectors of all the required fingerprints and their inverses. We then define $\mathcal{S}$ and $\mathcal{T}$ vectors as follows:

$$\mathcal{S} = [\mathbf{K}_p(y_0), \mathbf{K}_p(y_0)\mathbf{K}_p(y_1), \ldots, \mathbf{K}_p(y_0)\ldots\mathbf{K}_p(y_{n-1})],$$
$$\mathcal{T} = [\mathbf{I}, \mathbf{A}_p(y_0), \mathbf{A}_p(y_1)\mathbf{A}_p(y_0), \ldots, \mathbf{A}_p(y_{n-m})\ldots\mathbf{A}_p(y_0)].$$

So $\mathcal{S}$ is an inclusive scan over vector $\mathcal{K}$ with right matrix multiplication modulo prime p as its associative operator. Similarly, $\mathcal{T}$ is an exclusive scan over vector $\mathcal{A}$ with left matrix multiplication modulo prime p as its associative operator. Then, it is clear that for $0 \le r < n-m+1$:

$$\begin{aligned} \mathbf{K}_p(Y[r]) &= \mathcal{T}[r]\mathcal{S}[r+m-1] && (4)\\ &= [\mathbf{A}_p(y_{r-1})\ldots\mathbf{A}_p(y_0)] \times \\ &\quad [\mathbf{K}_p(y_0)\ldots\mathbf{K}_p(y_{r-1})\mathbf{K}_p(y_r)\ldots\mathbf{K}_p(y_{r+m-1})] && (5)\\ &= \mathbf{K}_p(y_r)\ldots\mathbf{K}_p(y_{r+m-1}), && (6) \end{aligned}$$

Thus, after computing two scans (i.e., computing $\mathcal{S}$ and $\mathcal{T}$), we can compute any required fingerprint by computing a single matrix multiplication (as in (4)). Based on Eq. (5), it is important to use different matrix multiplications (i.e., left/right) for each operator in order that each pair of matrices and their inverses correctly cancel each other out. Throughout this paper, we refer to this method as Cooperative Rabin-Karp (CRK), because scan operations can be computed cooperatively by a set of independent threads or processors.

Our cooperative implementation uses this second fingerprint, but we should point out that it might be possible to improve the performance of the cooperative algorithm using a variant of the first fingerprint. Blelloch [14] points out, among many other things, that any sequential scan operation whose output can be represented by a recurrence of the form $x_i = a_i x_{i-1} + b_i$, where a_i and b_i are fixed input arrays and can be computed by a scan operation. In our case, $a_i = 2$, we have $b_i = y_i - 2^m y_{i-m}$, which can be computed in $O(1)$ from the input y, and all operations are done modulo p. Specialized to this situation, Blelloch's method is to define an associative operator $\cdot$ on pairs $c_i = (a_i, b_i)$, where $c_i \cdot c_j = (a_i a_j, b_i a_j + b_j)$. He proves that this is an associative operator on the set of pairs, and so it can be implemented using a scan operation. In addition to requiring many fewer operations than the second fingerprint, implementing this would only require one scan operation, not two. In practice, we have observed that this function produced more false positives due to hash collisions than the matrix-fingerprint, unless we used more expensive 64-bit operations. Thus our initial attempt to implement this did not lead to much of an improvement over our current CRK implementation. In any case, we expect neither approach would compete with our best methods, as we shall see in Section 8.

4. DIVIDE-AND-CONQUER STRATEGY

Cooperative RK (Section 3.3) elegant exploits parallelism by having all of the processors cooperate through the two large scan operations. These parallel scans require intermediate communication between different processors and cores (threads from different blocks). However, CRK benefits little from the computational or memory hierarchy in its implementation (all operations are device-wide and global), so we

also explored a straightforward *divide-and-conquer* approach. Dealing with smaller subproblems gives us a lot of flexibility in our design choices (e.g., work distribution among processors/cores, and exploiting the memory hierarchy), which allows us to tailor the computation to the GPU's strengths.

Divide and conquer is a natural approach to string matching; the easiest way to divide the work is to assign different parts of the text to different processors and process each part seperately. The final result is simply a union of matching results for each subproblem. In order to maintain independence between different subproblems, and to avoid extra communication between processors, this division should be done in an intelligent way. We must consider an overlap of length $m-1$ characters between consecutive subtexts to avoid losing any potential matches at the boundaries. Let L denote the number of subtexts, or equivalently, the granularity of the divide-and-conquer approach. Then, if we show each subtext as Y^ℓ, for $0 \leq \ell < L$, the division process can be shown as:

$$Y^\ell = \underbrace{y_{\ell g} y_{\ell g+1} \cdots y_{\ell g+g-1}}_{\text{exclusive characters}} \underbrace{y_{(\ell+1)g} \cdots y_{(\ell+1)g+m-1}}_{\text{overlapped characters}}, \quad (7)$$

where each subtext has $g = (n-m+1)/L$ *exclusive* characters, plus $m-1$ *overlapped* characters. For simplicity we assume that $(n-m+1)$ is a multiple of L, otherwise we can easily extend our text (increasing n with enough dummy characters) to be such a number. Now, each of these subtexts have $\bar{n} = g+m-1$ characters and are independent from each other [1].

In general, the possible cases for the number of subtexts is from $1 \leq L \leq n-m+1$. We define *Divide-and-conquer RK* (DRK) as a method in which each processor performs the SRK on its own subtext individually. $L=1$ corresponds to SRK applied to the whole text. We can also combine ideas of both the CRK and the DRK methods and form a *hybrid* method. We define *Hybrid RK* (HRK) as a method in which the main text is divided into subtexts (Eq. (7)) and then each subtext is assigned to a group of processors, which will perform CRK on it. If $L=1$, HRK will be identical to CRK. Figure 1 shows our four possible formulations—serial, cooperative, divide-and-conquer, and hybrid—schematically.

Intuitively, a larger number of subtexts L (i.e., a finer granularity) yields more parallelism. At the same time, it means more potential overlaps and hence more redundant work because the total number of overlapped characters is $L(m-1)$. One important question in this work is to determine the optimal choice of L depending on the method and other machine and problem characteristics.

4.1 Theoretical analysis with finite processors

In the following, we want to investigate each method's depth-work analysis under equal and finite number of p processors. We also assume that all processors have access to a globally shared memory.

[1]This approach has commonality with stencil methods that operate over a large domain on hierarchical parallel machines: the domain is divided across cores and parallelism is exploited within a core. Typically, each core is assigned not only its own subdomain but also an overlapping region with its neighbors (the "halo"). Larger halos result in less communication at the cost of more redundant computation. In string matching, the overlap size is fixed, but the total overlap (i.e., total redundant work) is changed by the granularity of our division. Our objective will then be reaching a trade-off between minimizing redundant work (overlap) and maximizing achieved parallelism (maximizing the granularity).

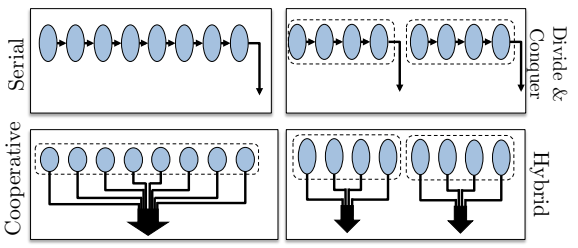


Figure 1: Schematic view of the four approaches considered in this paper. The picture is simplified and shows the work distribution in each method. Dashed lines indicate subproblems.

Method	Step complexity	Work complexity
CRK	$O(n/p + \log p)$	$O(n)$
DRK	$O((n+mL)/p)$	$O(n+mL)$
HRK	$O((n+mL)/p + (L/p_1)\log p_2)$	$O(n+mL)$

Table 1: Step-work complexity for our methods with p processors (p_1 groups with p_2 processors in each).

We know that scan operations can be done with $O(n/p + \log p)$ depth complexity and $O(n)$ work. The CRK method requires two scan operations, followed by a constant round of final multiplications and comparisons, and $O(n)$ added work. Overall, the CRK requires $O(n)$ work and its running time will be

$$T_{\text{CRK}}(n,p) = O(n/p + \log p). \quad (8)$$

In the DRK method, each subtext is assigned to a processor. If we have $L > p$ subtexts of length $g+m-1$, each subtext can be processed sequentially using SRK in $O(g+m-1)$ time steps. Total work will be $O(n+Lm)$. Since at most p processors can be used at a time, $T_{\text{DRK}}(n,p)$ will be

$$T_{\text{DRK}}(n,p) = \frac{L}{p} O(g+m-1) = O(\frac{n+Lm}{p}). \quad (9)$$

For the HRK, we assign each subtext to a group of processors, and each group runs CRK with its processors. Here we have a new degree of freedom: fewer groups of processors with more processors per group or vice versa. We assume p_1 groups and p_2 processors per group, for a total of $p = p_1 p_2$ processors. Subtexts are assigned to groups (p_1 at a time), while each subtext can be processed in parallel as in CRK with $O((g+m-1)/p_2 + \log p_2)$. Overall, the running time $T_{\text{HRK}}(n,p)$ will be

$$\begin{aligned} T_{\text{HRK}}(n,p) &= \frac{L}{p_1} O(\frac{g+m-1}{p_2} + \log p_2) \\ &= O(\frac{n+Lm}{p} + \frac{L}{p_1}\log p_2). \end{aligned} \quad (10)$$

For a fixed n, m, L and p, the runtime component $(L/p_1)\log p_2$ is minimized if $p_2 = 1$ and $p_1 = p$. It is interesting to note that in such case, HRK will be identical to the DRK method. For HRK, then, it is better to have more groups with fewer processors in each than fewer groups with more processors. Also, total work will be $O(n+Lm)$.

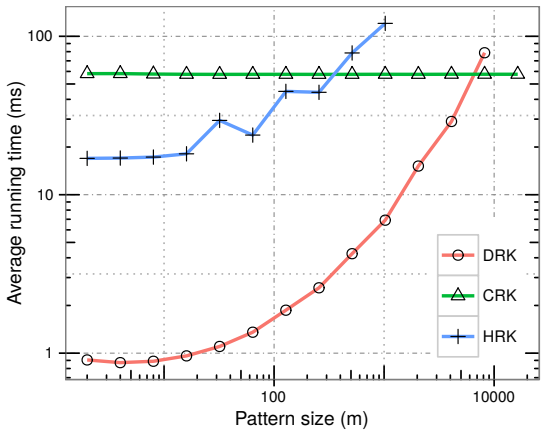


Figure 2: Average running time (ms) versus pattern size m, for a fixed binary text of size $n = 2^{25}$.

4.2 Theoretical Conclusions

Table 1 summarizes the step and work complexity for our three parallel methods with an equal number of processors. Although we have considered some simplifying assumptions in the above analysis, we can make some interesting general conclusions. First of all, since the sequential RK method is linear in time, as text size increases we expect to see an approximate linear increase in the running time of all our parallel alternatives (because of the $O(n/p)$ term in Eq. (8), (9), and (10)).

As the pattern size m increases, more overlapped characters are included in DRK and HRK. These overlaps are redundant work and thus, there always exists a pattern size for which CRK is superior to the other algorithms, and for all larger patterns. This can also be seen in step complexities and the fact that CRK does not have any factor of pattern size in it. Also, with fixed number of processors (fixed p) and for large sizes of text and pattern our asymptotic analysis gets more accurate and hence we expect the DRK method to be asymptotically superior to the HRK method. However, for smaller text or pattern sizes, the order may differ.

5. SUMMARY OF RESULTS

At this point in the paper, we have introduced and analyzed three parallel algorithm formulations based on the RK algorithm: cooperative (CRK), divide-and-conquer (DRK), and hybrid (HRK). In Section 4.1 we analyzed the theoretical behavior of these algorithms and in Section 4.2, predicted their implementation behavior. Figure 2 shows the performance of these three algorithms, all implemented on an NVIDIA K40c GPU, with a fixed text size and varying pattern sizes. We observe that DRK is superior to the other methods for small patterns, but degrades as pattern size increases. Because CRK's performance is independent of pattern size, we note a crossover point where for all larger patterns the CRK method is superior. While HRK is better than CRK for small patterns, it is always inferior to the DRK method.

In the next three sections, we will build from the algorithms we detailed above, both analyzing and improving their performance as well as extending them to address practical concerns such as non-binary alphabets. We address the following issues:

1. In Section 6, we provide implementation details for each of our parallel algorithms and describe a series of optimizations that improve their performance.

2. Karp and Rabin [3] only discussed binary alphabets and at this point in the paper, so have we. In Section 7.1 we modify our algorithms to support any alphabet size.

3. Figure 2 demonstrates a large performance gap between DRK and CRK: DRK is up to 65x faster than CRK for small patterns. However, as pattern size increases, this performance gap narrows. We wish to preserve DRK's high performance for any pattern size and in Section 7.2, we propose a novel two-stage method (DRK-2S) based on the original DRK method that achieves this goal.

4. In Section 8 we thoroughly analyze the performance of all of our algorithms. Section 8.1 identifies the best parallel algorithm for any text and pattern size for binary alphabets. Then in Section 8.2, we compare our binary algorithms with the existing fastest CPU methods, and in Sections 8.3 and 8.4, extend this comparison for general alphabets. All our experiments up to this point are on randomly generated texts. In Section 8.5, we expand our performance analysis to real-world text.

6. IMPLEMENTATION DETAILS

In the previous sections, we concentrated on our theoretical approach to parallel RK-based pattern matching. In this section, we dive into our implementation strategy. Since all our implementations are based on C, we assume that any character has a size of one byte; binary strings are handled as sequences of bytes. Throughout this section and beyond, some of our decisions we make are influenced by the CUDA programming model and GPU hardware.

6.1 Cooperative RK

For the CRK method we described in Section 3.3, we used the second class of fingerprints. We begin by mapping the input text to form the intermediate vectors $\mathcal{K}$ and $\mathcal{A}$. Then we perform two device-wide scan operations on these intermediate vectors to compute $\mathcal{S}$ and $\mathcal{T}$. In order to perform these scans, we use the scan operations provided by the Thrust library [15], with input and output stored in global memory and left/right matrix multiplications modulo a given prime number as their associative operators, respectively. All intermediate results are in the form of arrays of 2-by-2 integer matrices (as shown in Section 3.3): $\mathcal{K}$, $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{T}$ each requires 16B per input character. As a result, our CRK implementation not only uses the more computationally heavy fingerprints (second class), but also requires more global memory accesses per text character read.

6.2 Divide-and-Conquer RK

Our DRK implementations use the first class of fingerprints. Modulo operators do not have hardware implementations on current GPUs and hence are very slow compared to other operations. The only reason that we need this operator in equation (2) is to avoid overflows and hash the results; however, we can avoid these operations by making sure that an overflow never happens. For binary patterns of size less

than or equal to 32 characters, we can use 32-bit registers (or 64-bit registers for $m \leq 64$) and guarantee no overflows. This change can significantly improve the performance for small patterns compared to the original updates. First, let us consider the most general case. Consider equation (2) and suppose we have previously computed 2^{m-1} modulo prime p and stored it in a register (called `base`). Then, if we denote y_r by `y_old` and y_{n+r-1} by `y_new`, the updated fingerprint `F` will be `F` $\leftarrow$ `((F + p - R)` $\ll$ `1) + y_new) % p`, where `R` $\leftarrow$ `(base * y_old) % p`.

For 32-bit registers, if $p < 2^{30}$, then `R` and `F` never overflow (`base` is itself modulo `p`). Originally we should have `F-R` instead of `F+p-R`. But since `F` and `R` are both positive and less than `p`, we add a `p` value (neutral to modulo operator) to assure the result is positive before shifting ($0 \leq F+p-R < 2p$). The choice of the maximum of 30 bits is required because the result of both lines must not overflow before the modulo operator. Now, for patterns smaller than 32 characters we can have `F` $\leftarrow$ `((F & mask)` $\ll$ `1) + y_new`, where `mask` $\leftarrow$ $\sim$`(1` $\ll$ `m)`. Since we use the binary representation of characters as their fingerprints (exact hashing with theoretical zero collision), we can mask out the effect of y_r by an AND operation and avoid using an extra subtraction here. `mask` can be computed once, so on average we have one bitwise AND, one shift and one summation per character in this case. We refer to this version as DRK without modulo (DRK-WOM-32bit). With minimal (but non-zero) performance degradation, we can instead use 64-bit registers and support binary patterns up to 64 characters (DRK-WOM-64bit).

In DRK (or DRK-WOM), each thread is supposed to process $g+m-1$ consecutive characters from the text and verify whether or not there is a match for its g exclusive strings (Section 4). This is not an ideal memory access pattern for high-performance GPU implementation. We would instead prefer coalesced accesses, where consecutive threads within a warp access consecutive elements. To alleviate this problem, with N_T threads per block, we initially load N_T consecutive subtexts into each block's shared memory in a coalesced manner (total of $gN_T + m - 1$ characters per block), and then each thread reads its own subtext locally from much faster shared memory (where coalescing is less important). In GPUs, memory accesses are in 4B alignments. Therefore we read elements as `char4` vectors (4 consecutive characters) to better exploit the available memory bandwidth[2].

6.3 Hybrid RK

As discussed in Section 4, HRK assigns each subtext to a group of processors, where a CRK method is performed on each subtext. On the GPU, we assign each subtext to a block, and threads within each block perform CRK on their own subtext. We use a block-wide scan operation from the CUB library [17]. In order to use CRK on each subtext, we need to store intermediate vectors in shared memory ($\mathcal{S}$ and $\mathcal{T}$ when using the second fingerprint). Current GPUs have very limited shared memory of at most 48 KB per block, so the extra storage forces us to either have fewer threads per block (small N_T) or very small subtext sizes (small g) and equivalently too many blocks (large L). In neither scenario are we utilizing our hardware's full potential, and consequently our HRK implementation compares poorly to our DRK- and CRK-based implementations.

[2] `char4` is a built-in CUDA vector type with 4B alignment [16, Ch. B.3.].

7. EXPANDING BEYOND THE SIMPLE RK

7.1 General alphabets

In their original paper, Karp and Rabin only considered binary alphabets [3]; we now extend this to general alphabets. Suppose we have an alphabet set Σ such that each character $a \in \Sigma$ requires at most $\sigma = \lceil \log \Sigma \rceil$ bits of information. Then, for any arbitrary pattern $X = x_0 \dots x_{m-1} \in \Sigma^m$, we can extend our first class of fingerprints to be $F_p(X) \stackrel{p}{\equiv} \sum_{i=0}^{m-1} x_i 2^{\sigma(m-i-1)}$. So, for this first class of fingerprints, the new update equation will be

$$F_p(Y[r+1]) \stackrel{p}{\equiv} 2^{\sigma} \left(F_p(Y[r]) - 2^{\sigma(m-1)} y_r \right) + y_{r+m+1}. \quad (11)$$

This update equation has a similar computational cost as equation (2). Its implementation will also be similar to what we outlined in Section 6.2. The `base` register we used in Section 6.2 is now precomputed to be $2^{\sigma(m-1)}$ modulo p. In order to avoid overflows, with the same reasoning as we had before, we should make sure that $p \leq 2^{31-\sigma}$. We also update our DRK-WOM implementation to incorporate σ: `F` $\leftarrow$ `((F & mask)` $\ll$ σ`) + y_new`, where `mask` $\leftarrow$ $\sim$`(((1`$\ll$ σ`) - 1)` $\ll$ `m)`. Here `mask` is intended to wipe out any possible remainder from the oldest added character. By using 64-bit registers, we can support pattern lengths of $m \leq 64/\sigma$, i.e., up to eight 8-bit ASCII characters.

For the second class of fingerprints, we first identify our fundamental fingerprints $\mathbf{K}_0 \dots \mathbf{K}_{2^\sigma - 1}$. Suppose we have an arbitrary character $z = c_{\sigma-1} \dots c_0$ where $c_i \in \{0, 1\}$. Then, $\mathbf{K}_z = \mathbf{K}_{c_{\sigma-1}} \times \cdots \times \mathbf{K}_{c_0}$. Inverse fingerprints are similarly defined as $\mathbf{A}_p(z) = \mathbf{A}_p(c_0) \times \cdots \times \mathbf{A}_p(c_{\sigma-1})$. These fundamental fingerprints and their inverses can either be precomputed and looked up from a table at runtime or else can be computed on the fly. In either case, the amount of computation can become up to σ times more expensive and not any better than the binary CRK method, and hence for general alphabets, we prefer the first class of fingerprints and the DRK method.

7.2 Two-stage matching

For small patterns, our DRK implementations outperform our other methods on GPUs as well as the fastest CPU-based codes (Section 8 has more details). DRK-WOM, for instance, achieves up to a 66 GB/s processing rate. However, it works only for fairly small patterns (up to $64/\sigma$ characters if we use our 64-bit version). As the pattern size increases, DRK is the superior method up to almost 512 characters (for $1 \leq \sigma \leq 8$). However, DRK's performance degrades gradually as the pattern size increases for two reasons. 1) As m increases, the total number of characters that each thread must read until it can process g strings increases linearly ($g + m - 1$). 2) As m increases, the total number of characters stored in shared memory also increases ($gN_T + m - 1$) and hence our device occupancy (and overall performance) gradually decrease.

As we have discussed earlier in this paper, the RK algorithm does not depend on the content of either the text or the pattern. This is a desirable property: the performance of RK implementations only depends on text and pattern sizes (n and m). We make no runtime decisions (e.g., branches) based on pattern or text content, which is beneficial for the uniformity goal we outlined in the introduction. In practice, most of our resources can be assigned statically at compile time (like the amount of required shared memory per block).

But an RK-based implementation processes every possible substring of length m in the text. Many well-known sequential matching algorithms (such as those introduced in Section 2) use different techniques to avoid this unnecessary work. In this subsection, we propose an algorithm that balances between the uniform workload of RK and avoids the unnecessary work of checking every possible substring.

Our *two-stage* DRK matching method (DRK-2S) first *skims* the text for a random small substring of the pattern and then *verifies* the potential matches that we identify. Suppose we search for a smaller substring of the pattern ("subpattern") of length $\bar{m} < m$. This subpattern can be chosen arbitrarily: it can be a prefix, a suffix, or any other arbitrary alignment from the main pattern. We then search our text using one of our DRK methods (preferably the DRK-WOM). Each thread stores its potential matches, if any, into a local register-level stack. We note that any sequential method could be used for the skimming stage, but we focus on using one of our DRK methods because their implementations on GPUs balance load efficiently across the GPU.

For final verification, we expect that by choosing a large enough subpattern size, the expected number of potential matches will be much less than the text size n. As a result, we group threads together to verify potential matches altogether in parallel rather than individually and sequentially. In our implementation, we use a warp (32 threads) as the primary unit of verification. By doing so, we can have an efficient implementation by using warp-synchronous programming features and warp-wide voting schemes.

Initially we divide our pattern into several consecutive chunks of 32 characters. For each chunk, we compute the substring's fingerprint as $F_p(x_0 \dots x_{31}) \stackrel{p}{\equiv} \sum_{i=0}^{31} x_i 2^{\sigma(31-i)}$ by using a random prime number p as before. For example, for a pattern of length $m = 128$, we would compute 4 different fingerprints: $F_p(x_0 \dots x_{31}), \dots, F_p(x_{96} \dots x_{127})$. Now, we should verify whether for each potential match beginning at index r, $F_p(y_r \dots y_{r+31}), \dots, F_p(y_{r+96} \dots y_{r+127})$ are all equal to their pattern's counterparts.

After the skimming stage, each thread has a local stack of its own potential matches. Threads can find out about the existence of any non-empty stack within their warp (using a ballot). We start the verification based on threads' lane IDs (priority from 0 to 31). At this point all threads ask about the beginning index (say r) of that specific potential match (by using a shuffle) and then continue by coalesced reading of a chunk of 32 consecutive characters (from y_r until y_{r+31}). Then, by using at most five rounds of shuffles (performing a warp-wide reduction), we can compute the fingerprint corresponding to that chunk and compare it to that of the pattern's. In case it does not match, that specific potential match is not a final match and it is popped out of its thread's local stack. Otherwise, we continue to the next chunk of characters until all chunks are verified. The whole process is continued until all local stacks are empty. Algorithm 1 shows the high-level procedure of the DRK-2S.

We expect the DRK-2S method will significantly outperform the DRK method in dealing with large patterns, because we incur the cost of verifying a potential match (including reading m characters from the text) *only* when we first identify a subpattern. Consequently, we significantly reduce the memory bandwidth requirements compared to DRK, and as we will see in the next section, achieve large speedups in practice.

Algorithm 1 Two-stage matching method (DRK-2S)

```
1: for each warp w parallel do
2:     for each thread t within w parallel do
3:         Search for subpattern of length m̄ < m using DRK
4:         S_t ← potential matches
5:         R_t ← beginning indices of S_t
6:     end for
7:     for each index r ∈ ⋃_{t∈w} {R_t} do
8:         for 0 ≤ k < ⌈m/32⌉ do
9:             Compute F ← F_p(y_{(r+32k)} ... y_{(r+31+32k)})
10:            if F is not equal to F_p(x_{32k} ... x_{(32k+31)}) then
11:                r is not a match
12:            end if
13:        end for
14:        if all above k fingerprints matched then
15:            r is a match
16:        end if
17:    end for
18: end for
```

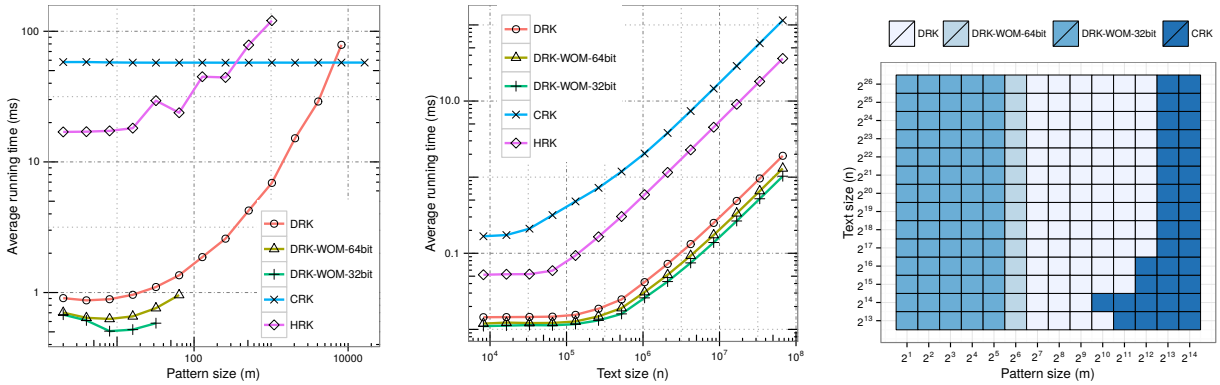
8. PERFORMANCE EVALUATION

In this section, we experimentally assess the performance of all our proposed algorithms. Unless otherwise stated (Section 8.5), we use randomly generated texts and patterns with arbitrary alphabet sizes[3]. All our parallel methods are run on an NVIDIA Tesla K40c GPU, and compiled with NVIDIA's nvcc compiler (version 7.5.17). All our sequential codes are run on an Intel Xeon E5-2637 v2 3.50 GHz CPU, with 16 GB of DDR3 DRAM memory. We used OpenMP v3.0 to run our CPU codes in parallel over 8 cores (2 threads per core). All parallel GPU implementations are by the authors, while all CPU codes are from SMART library [18] and run in a divide-and-conquer fashion over our multi-core platform.

In this section, all our experiments are averaged over at least 200 independent random trials. All results in this section—average running times (ms) and processing rates (GB/s)—are measured without considering the transfer time (from disk or from CPU to GPU). In other words, for all GPU methods we assume that text is already stored on the off-chip DRAM memory of the device. As we will see shortly, our processing rates are indeed currently much faster than PCIe 3.0 transfer time. However, the recently announced CPU-GPU NVLink would allow us to eliminate this bottleneck by feeding the GPU at the rate of our fastest string matching implementations.

All our GPU codes are run with $N_T = 256$ threads per block. Based on the utilized number of registers per thread and also the size of shared memory per block, this amount of N_T has given us a very high occupancy ratio on our GPU device. Another important parameter in our divide-and-conquer methods is the total number of subtexts L, i.e., the total number of launched threads. Instead of optimizing for L, we instead optimize for $g = (n-m+1)/L$, since it directly expresses the amount of work assigned to each thread, i.e., g is the total number of exclusive characters in each subtext. An advantage of RK-based methods is that by running a

[3]Because the RK algorithm is independent of the content of the text or pattern, the runtime for any text or pattern for a fixed (m, n) will be identical.



(a) Avg. running time vs. m; $n = 2^{25}$ (b) Avg. running time vs. n; $m = 16$ (c) Superior Algorithms

Figure 3: Average running time vs. (a) pattern length (m) and (b) text length (n). (c) Superior algorithms in the input parametric space (m, n), where m is the pattern length and n is the text length. All GPU methods are implemented by the authors.

few tests on randomly generated texts and patterns, we can tune the optimum amount of g to fully exploit available GPU resources. Our simulations showed that the range of $12 \leq g \leq 44$ for $2 \leq m \leq 2^{16}$ achieves the best results.

8.1 Algorithm behavior in problem domains

In this part, we revisit the question of "which method is best suited for text size n and pattern size m?" We begin with a fixed text size n and varying pattern sizes m for a binary alphabet. Based on our earlier discussions, we expect all of our DRK family algorithms to perform better than CRK and HRK because of 1) significantly cheaper computations with the first class of fingerprints and 2) fewer and coalesced global memory accesses. All DRK methods only read text characters once and report results back to global memory only if necessary; however, CRK (and to some extent HRK) first process the text into 2-by-2 integer matrices and perform several device-wide computations over these intermediate vectors (requiring additional global memory operations). However, we also predicted that as pattern size becomes large, DRK methods fall behind CRK because of 1) excessive work due to overlap and 2) the limited capacity of available per-block shared memory in current GPUs.

Figure 3a shows the average running time over 200 independent random trials on texts with 32 M characters and various pattern sizes ($2 \leq m \leq 2^{16}$). In this simulation, the DRK methods are superior to CRK for $m < 2^{13}$. In addition, HRK performs better than CRK for $m \leq 256$ (because most intermediate operations are performed locally in each block), but it is never competitive to the DRK methods. All DRK methods have similar memory access behavior; the only difference is between their update costs. Consequently, on pattern sizes where they are applicable, DRK-WOM-32bit is better than DRK-WOM-64bit, and both are better than the regular DRK method, because they avoid expensive modulo operators.

We expected that all parallel algorithms would show linear performance with respect to text size (summarized in Table 1). Figure 3b depicts average running time versus varying text sizes and a fixed pattern size $m = 16$ for binary characters. This expected linear behavior is evident after we reach a certain text size, which is large enough to fully exploit GPU resources under each scenario's algorithmic and implementation limitations.

Let's turn from the above snapshots from the (m, n) problem domain to the overall behavior of our algorithms across the whole problem domain. Figure 3c shows the superior algorithms across the (m, n) space. As we expect, the DRK family is superior for small patterns, and the CRK method for larger patterns, but the crossover point differs based on the text size. Within the DRK family, the DRK-WOM-32bit has the cheapest computational cost and is hence dominant. However, it can only be used for patterns with $m \leq 32$ characters. The next best choice is the DRK-WOM-64bit, which supports up to $m \leq 64$ characters. For larger patterns we should choose the regular DRK method, per Section 6.2.

8.2 Binary sequential algorithms

Now that we have studied the competitive behavior of our parallel algorithms with each other, we compare their performance with the best sequential binary matching algorithms. Faro and Lecroq's comprehensive survey [7] covers the fastest sequential methods. We begin with available source codes from the SMART library [18][4] and parallelize them in a divide-and-conquer fashion (as described in Section 4) but distributed over 8 CPU cores (total of 16 CPU threads). We compile and run the results in parallel using OpenMP v3.0 with –O3 optimizations. Figure 4a shows the average running time of the best CPU implementations against our parallel algorithms implemented on the GPU.

We note that the CPU algorithms perform better as the pattern size increases. Most of these algorithms use different techniques to avoid unnecessary processing of text based on the content of read characters and the pattern (as we also leverage in our DRK-2S from Section 7.2). As the pattern size increases, this strategy avoids more work on average (the methods can take larger jumps through the text). On the other hand, as we noted previously, our DRK method

[4]We compare against all top-performing algorithms from Faro and Lecroq's survey [7] and SMART library [18] except for SSEF, which in our experiments did not deliver competitive performance.

	Algorithm	Pattern size (m) 4	16	64	256	1024
CPU	SRK	0.95	1.11	1.15	1.16	1.16
	SA	**1.57**	3.41	4.1	4.09	4.01
	AOSO2	0.80	4.25	4.53	4.54	4.54
	HASH5	—	**4.69**	5.25	5.68	5.63
	HASH8	—	4.62	**5.93**	**6.33**	**5.98**
GPU	CRK	0.58	0.58	0.58	0.58	0.58
	DRK	38.5	34.91	24.75	**12.98**	**4.86**
	DRK-WOM-32bit	**54.84**	**64.50**	—	—	—
	DRK-WOM-64bit	52.40	51.07	**35.06**	—	—
Speedup (GPU/CPU)		34.9x	13.8x	5.9x	2.1x	0.8x

Table 2: Processing rate (throughput) in GB/s. Texts and patterns are randomly chosen using a binary alphabet.

	Algorithm	Pattern size (m) 4	16	64	256	1024
CPU	SA	3.96	4.20	4.92	4.91	5.26
	HASH3	2.24	9.69	11.16	12.89	12.08
	HASH5	–	7.22	10.29	11.63	12.21
	HASH8	–	7.33	11.56	10.69	12.18
	FJS	4.27	5.80	7.13	6.95	7.77
	SBNDM-BMH	4.82	5.22	6.90	7.03	7.80
	GRASPm	3.89	5.69	7.32	7.01	8.25
	FS	4.01	6.19	7.22	6.70	8.05
	SSECP	**7.66**	9.43	9.32	9.71	9.87
	EPSM	7.58	5.34	10.35	12.08	3.27
	FSBNDM	7.71	**11.00**	**12.55**	**12.92**	**14.59**
	EBOM	5.49	8.04	9.48	4.18	2.59
GPU	DRK	40.31	34.94	24.75	12.98	4.86
	DRK-2S	**62.68**	**53.77**	**53.73**	**53.51**	**53.04**
Speedup (GPU/CPU)		8.18x	4.89x	4.28x	4.14x	3.63x

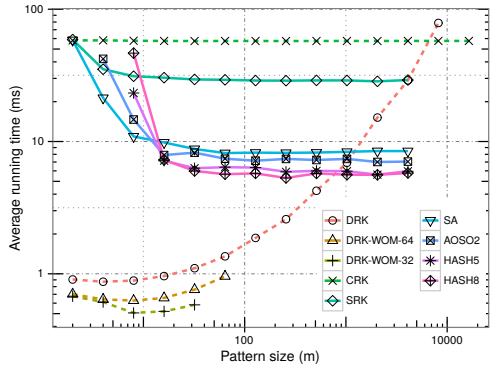
Table 3: Processing rate (throughput) in GB/s. Text and patterns are randomly chosen with alphabet $\sigma = 8$ (8-bit ASCII characters).

degrades as the pattern size increases. As a result, our DRK family of implementations is superior to all binary CPU methods for patterns up to almost 1024 characters. Table 2 reports processing rate (throughput) for some specific pattern sizes measured over random texts with 2^{25} binary characters.

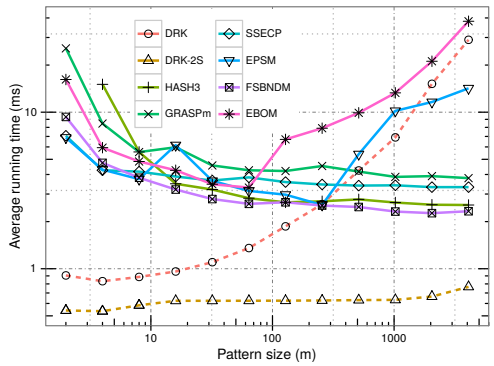
8.3 General sequential algorithms

Except for very minor low-level differences, our DRK implementations with any non-binary alphabet should perform as well as our binary DRK implementation. Figure 4b shows some of the fastest matching algorithms for $\sigma = 8$ (based on the survey from Faro and Lecroq [7]). As with our binary sequential comparison, we parallelize codes from the SMART library [18] across 8 CPU cores/16 threads. For the DRK-2S method, we used the DRK-WOM-64bit as our matching method for the skimming stage. Here, since we have $\sigma = 8$, we can use up to $\bar{m} = 8$ characters for our subpatterns.

Table 3 summarizes our achieved processing rates on random texts with 2^{25} 8-bit characters. Our DRK-2S method achieves a geometric mean speedup of 4.81x over the fastest CPU methods. Although we consider patterns up to 1024 characters in this table, the DRK-2S continues to be superior to all the CPU methods up to patterns of almost 64k characters. For example, with 32k characters DRK-2S achieves 9.19 GB/s while the best CPU method, the FSBNDM, achieves 5.89 GB/s. The main reason behind our performance degradation for very large patterns is an increased number of memory accesses for each potential match



(a) Binary alphabet ($\sigma = 1$)



(b) 8-bit alphabet ($\sigma = 8$)

Figure 4: Average running time (ms) versus pattern size m with fixed text size $n = 2^{25}$ for (a) $\sigma = 1$ and (b) $\sigma = 8$. Solid and dashed lines correspond to CPU and GPU implementations respectively.

($O(m/32)$ accesses). We could potentially address this issue by modifying DRK-2S to consider chunks of 128 characters (instead of 32) and use `char4` memory accesses in its verification stage as well; this would reduce the number of accesses by a factor of 4. However, for realistic pattern sizes this modification would not significantly improve our performance.

8.4 Dependency on alphabet size

In the last two parts, we have evaluated our parallel GPU methods against the best CPU methods for two cases of binary characters (Section 8.2) and general 8-bit characters (Section 8.3). In this section, we focus on the DRK-2S method and continue our discussion over randomly generated texts over various alphabet sizes ($\Sigma = 2^{\sigma}$). Table 4 shows the processing rate of the DRK-2S for $1 \leq \sigma \leq 8$ and $4 \leq m \leq 1024$. For the skimming stage, we use a subpattern length of $\bar{m} = 16$ for $1 \leq \sigma \leq 4$ and $\bar{m} = 8$ otherwise.

It is interesting to note that with a fixed pattern size, our performance increases as the alphabet size increases. This is mainly due to the fact that for randomly generated texts and with a fixed pattern size, as alphabet size increases, we get less likely to visit a match in our skimming stage (probability of $\sigma^{-\bar{m}}$), and hence fewer verifications are required in the second stage. It is also interesting to note the sudden performance degradation for $m > \bar{m}$ with each alphabet size, which is

	Pattern size (m)						
σ	4	8	16	32	64	256	1024
1	52.40	56.55	55.04	49.96	49.86	49.74	49.23
2	59.98	58.91	55.75	51.04	51.03	51.01	50.15
3	60.57	58.94	55.86	51.13	51.11	50.91	50.45
4	60.58	58.85	55.78	51.09	51.01	50.71	49.82
5	60.59	58.88	53.47	53.52	53.44	53.38	52.76
6	60.63	58.94	52.36	53.46	53.42	53.30	52.29
7	60.59	58.87	53.50	53.45	53.42	53.18	52.11
8	62.68	63.66	53.77	53.69	53.73	53.51	53.04

Table 4: Processing rate (throughput) in GB/s of our DRK-2S method for various alphabet sizes ($\Sigma = 2^\sigma$) and different pattern lengths (m).

	Pattern size (m)						
Algorithm	4	8	16	32	64	256	1024
SA	16.68	17.23	17.51	17.19	15.08	16.20	15.67
HASH3	20.95	10.97	6.29	5.75	**4.67**	4.98	**3.77**
HASH5	–	14.56	7.40	5.99	5.32	4.96	5.77
HASH8	–	49.15	10.00	5.97	5.46	5.57	5.31
FJS	12.98	9.80	7.95	6.50	6.31	7.58	4.82
SBNDM-BMH	10.19	9.88	7.73	6.53	6.22	6.16	6.78
GRASPm	13.17	9.31	7.04	6.41	6.20	6.78	6.60
FS	13.06	9.58	7.43	5.73	6.06	6.38	6.52
FSBNDM	8.51	6.31	5.21	**4.92**	4.97	4.61	5.16
EBOM	7.55	6.17	**5.05**	5.42	5.25	5.49	6.47
SSECP	5.35	6.97	6.27	6.23	6.99	7.05	5.81
EPSM	**5.05**	**5.15**	5.69	5.43	4.75	**3.78**	6.64
DRK	2.46	2.62	2.84	3.25	3.99	7.59	20.34
DRK-2S	**1.58**	**1.63**	**1.88**	**1.84**	**1.84**	**1.84**	**1.83**
Speedup	3.2x	3.16x	2.69x	2.67x	2.54x	2.05x	2.06x

Table 5: Average running time (ms) for finding patterns of different sizes (m) on a sample text from Wikipedia with 100 M characters. The last row reports the achieved speedup of our DRK-2S method over the best CPU method.

directly related to whether or not we have to perform the verification stage. For $m > \bar{m}$ as the pattern size increases, we witness a very mild rate decrease mainly due to the number of times that a consecutive number characters are verified (line 8 in Alg. 1). Overall, as we discussed earlier, we do not expect our performance to be affected much by change in alphabet size (evident in update equation (11)), which is also clear from this table as well.

8.5 Real-world scenarios

So far, we have only used randomly generated texts and patterns for our experiments, because all of our RK-based methods have performance independence of the content of the text and the pattern. For DRK-2S, however, the total number of verifications directly depends on the number of potential matches found in the skimming stage. As a result, the algorithm is no longer data-independent. In this part, we consider some real-world scenarios for evaluation:

Natural language text: Table 5 shows the average running time (ms) of the DRK-2S method as well as all the high-performing CPU algorithms from Section 8.3 in finding several patterns of different sizes in a 100 MB sample text from Wikipedia[5]. Here DRK-2S achieves a geometric mean speedup of 2.59x against the best CPU codes.

[5]The enwik8 dataset includes the first 100 MB text from Wikipedia as of March 2006: http://prize.hutter1.net/.

	Pattern size (m)						
Algorithm	4	8	16	32	64	256	1024
SA	1.10	1.01	1.03	1.01	0.98	0.98	1.00
HASH3	1.35	0.80	0.60	**0.52**	0.49	**0.46**	0.52
HASH5	–	1.05	0.60	0.54	**0.44**	0.49	0.45
HASH8	–	2.74	0.73	0.54	0.45	0.52	**0.44**
FJS	1.83	1.51	1.55	1.47	1.64	1.59	1.50
SBNDM-BMH	1.32	1.01	0.79	0.56	0.58	0.57	0.65
GRASPm	1.36	1.09	1.07	0.83	0.74	0.71	0.67
FS	1.37	1.16	1.07	1.02	0.90	0.82	0.72
FSBNDM	1.24	0.88	0.64	0.56	0.54	0.58	0.61
EBOM	0.96	0.92	0.74	0.65	0.61	0.82	1.21
SSECP	**0.52**	**0.50**	0.74	0.65	0.66	0.65	0.65
EPSM	0.54	0.76	**0.50**	0.61	0.50	0.48	0.67
DRK	0.162	0.121	0.137	0.175	0.264	0.855	0.954
DRK-2S	**0.075**	**0.077**	**0.093**	**0.093**	**0.093**	**0.093**	**0.098**
Speedup	6.93x	6.49x	5.38x	5.59x	4.73x	4.94x	4.49x

Table 6: Average running time (ms) for finding patterns of different sizes (m) on E. coli genome with 4.6M characters. The last row reports the achieved speedup of our DRK-2S method over the best CPU method.

	Pattern size (m)						
Algorithm	4	8	16	32	64	256	1024
SA	0.84	0.84	0.84	0.88	0.78	0.78	0.78
HASH3	1.18	0.67	0.49	**0.43**	0.44	0.44	0.43
HASH5	–	0.86	0.60	0.49	0.46	0.47	0.50
HASH8	–	1.96	0.57	0.44	0.46	**0.43**	0.44
FJS	0.72	0.63	0.61	0.54	0.49	0.51	0.56
SBNDM-BMH	0.76	0.68	0.56	0.44	0.50	0.49	0.48
GRASPm	0.88	0.68	0.55	0.55	0.56	0.48	0.45
FS	0.79	0.60	0.54	0.51	0.46	0.46	0.49
FSBNDM	0.62	0.47	**0.45**	0.50	**0.41**	0.46	**0.43**
EBOM	0.51	0.48	0.53	0.55	0.53	0.63	1.46
SSECP	**0.45**	**0.41**	0.49	0.49	0.51	0.48	0.49
EPSM	0.56	0.53	0.55	0.50	0.46	0.53	0.66
DRK	0.080	0.086	0.098	0.125	0.188	0.609	0.684
DRK-2S	**0.053**	**0.055**	**0.065**	**0.065**	**0.065**	**0.065**	**0.071**
Speedup	8.49x	7.45x	6.92x	6.61x	6.31x	6.62x	6.01x

Table 7: Average running time (ms) for finding patterns of different sizes (m) on the Homo sapiens protein sequence with 3.3M characters. The last row reports the achieved speedup of our DRK-2S method over the best CPU method.

DNA sequences: Table 6 shows the average running time (ms) of regular DRK and DRK-2S methods as well as some of the best sequential algorithms over the E. coli genome sequence[6]. This is a text with almost 4.6 million characters drawn from four different possible characters ($\Sigma = 4$). The DRK-2S method achieves a geometric mean speedup of 5.45x against the best CPU codes.

Protein sequences: Table 7 shows the average running time (ms) of our DRK-based methods compared to the best CPU methods over processing of the Homo sapiens protein sequence, with 3.3 million characters and alphabet of 19 characters[6]. The DRK-2S method achieves a geometric mean speedup of 6.88x against the best CPU codes.

8.6 False positives

The RK algorithm, and hence all the proposed methods in this article so far, are randomized algorithms whose randomness comes from our choice of prime number and hashing process. While RK algorithms will always be successful in finding matches, it is possible to have false positives, i.e., multiple strings that hash to the same value. In the original RK algorithm, it is shown that if the chosen prime number

[6]Data available in SMART library: http://www.dmi.unict.it/~faro/smart/corpus.php

is less than $n(n-m+1)^2$, then the probability of error is upper-bounded by $2.511/(n-m+1)$ [3]. Even by using 64-bit variables and operations, that leaves us with a non-zero probability of false positives. On the other hand, all our DRK-WOM methods assign a unique fingerprint to their strings, and hence false positives are not possible. Our DRK-2S has a great advantage here. First of all, by using a DRK-WOM method in the skimming stage, we significantly reduce the total number of potential matches. Secondly, DRK-2S divides the pattern into multiple smaller-sized chunks of size 32 and hashes them with a random prime number. So for a pattern to result in a false positive, all of its chunks should independently hash to the same values as the original pattern's. So the probability is exponentially decreased. We did not encounter any false positives in our experiments with the DRK-2S. Still, we could arbitrarily decrease the false positive probability even more by hashing each chunk with a different prime number (or even using multiple prime numbers and multiple hashing per chunk). This would increase our computational load in the verification stage, but since the total number of potential matches after the skimming stage is significantly fewer than n, the impact on the overall performance would be negligible.

9. CONCLUSION

Our final outcome from this work is a novel two-stage algorithm that addresses the string matching problem. The DRK-2S algorithm fully exploits GPU computational and memory resources in its skimming stage, where we efficiently process the whole text to identify potential matches. We then test these potential matches using cooperative groups of threads. The result is a highly efficient string matching algorithm for patterns of any size, which we believe to be the fastest string-matching implementation on any commodity processor for pattern sizes up to nearly 64k characters. Though very efficient, our string matching method can only be used for *exact* matching of a *single* pattern (or at least very few). We hope that this algorithm and the presented ideas embodied in it can be used as a cornerstone for other high-throughput string processing applications such as in regular expressions and approximate matching algorithms, as well as for multiple-pattern matching scenarios where a text is matched against a dictionary of patterns (e.g., in intrusion detection systems).

Acknowledgments

Thanks to Martin Farach-Colton for helpful comments on the paper. Also, thanks to NVIDIA for providing the GPUs that made this research possible. We appreciate the funding support from UC Lab Fees Research Program Award 12-LR-238449, Sandia LDRD award #13-0144, and NSF award CCF-1017399.

10. REFERENCES

[1] M. C. Schatz and C. Trapnell, "Fast exact string matching on the GPU," University of Maryland, Tech. Rep., 2007.

[2] C.-H. Lin, C.-H. Liu, and S.-C. Chang, "Accelerating regular expression matching using hierarchical parallel machines on GPU," in *IEEE Global Telecommunications Conference*, ser. GLOBECOMM 2011. IEEE, Dec. 2011, pp. 1–5.

[3] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, Mar. 1987.

[4] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[5] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[6] T. Lecroq, "Fast exact string matching algorithms," *Information Processing Letters*, vol. 102, no. 6, pp. 229–235, Jun. 2007.

[7] S. Faro and T. Lecroq, "The exact online string matching problem: A review of the most recent results," *ACM Computing Surveys (CSUR)*, vol. 45, no. 2, p. 13, Feb. 2013.

[8] Z. Galil, "Optimal parallel algorithms for string matching," *Information and Control*, vol. 67, no. 1–3, pp. 144–157, Oct.–Dec. 1985.

[9] U. Vishkin, "Optimal parallel pattern matching in strings," *Information and Control*, vol. 67, no. 1–3, pp. 91–113, Oct.–Dec. 1985.

[10] X. Zha and S. Sahni, "GPU-to-GPU and host-to-host multipattern string matching on a GPU," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1156–1169, Jun. 2013.

[11] E. Seamans and T. Alexander, "Fast virus signature matching on the GPU," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, Aug. 2007, ch. 35, pp. 771–783.

[12] C. S. Kouzinopoulos and K. G. Margaritis, "String matching on a multicore GPU using CUDA," in *13th Panhellenic Conference on Informatics*, ser. PCI '09. IEEE, Sep. 2009, pp. 14–18.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, pp. 40–53, Mar./Apr. 2008.

[14] G. E. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, Nov. 1990.

[15] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU Computing Gems*, W. W. Hwu, Ed. Morgan Kaufmann, Oct. 2011, vol. 2, ch. 26, pp. 359–371.

[16] NVIDIA Corporation, "NVIDIA CUDA C programming guide," Sep. 2015, pG-02829-001_v7.5.

[17] D. Merrill, "CUB," 2011. [Online]. Available: nvlabs.github.io/cub/

[18] S. Faro and T. Lecroq, "Smart: a string matching algorithm research tool," 2011. [Online]. Available: http://www.dmi.unict.it/~faro/smart/