Parallel Solutions to the *k*-difference Primer Problem

Leandro Feuser and Nahri Moreano

School of Computing, Federal University of Mato Grosso do Sul, Brazil
Email: leandrofeuser@gmail.com, nahri@facom.ufms.br

Abstract. This paper presents parallel solutions to the k-difference primer problem, targeting multicore processors and GPUs. This problem consists of finding the shortest substrings of one sequence with at least k differences from another sequence. The sequences found in the solution are candidate regions to contain primers used by biologists to amplify a DNA sequence in laboratory. To the authors' knowledge, these are the first parallel solutions proposed for the k-difference primer problem. We identified two forms, coarse- and fine-grained, of exploiting parallelism while solving the problem. Several optimizations were applied to the solutions, such as synchronization overhead reduction, tiling, and speculative prefetch, allowing the analysis of very long sequences in a reduced execution time. In an experimental performance evaluation using real DNA sequences, the best OpenMP (in a quad-core processor) and CUDA solutions produced speedups up to 5.6 and 72.8, respectively, when compared to the best sequential solution. Even when the sequences length and the number of differences k increase, the performance is not affected. The best sequential, OpenMP, and CUDA solutions achieved the throughput of 0.16, 0.94, and 11.85 billions symbol comparisons per second, respectively, emphasizing the performance gain of the CUDA solution, which reached 100% of GPU occupancy.

Keywords: Inexact matching, High performance computing, Parallelism, Multicore processor, GPU

1 Introduction

Advances in DNA sequencing technologies have been causing biological databases to grow almost exponentially. Given this huge amount of data and the long length of biological sequences, high performance solutions to sequence analysis problems have been proposed in order to allow biologists to extract useful information from these data. Approximate string comparison is an essential operation in biological sequence analysis and serves as basis for several more complex manipulations. It properly models changes that happen in DNA sequences through the evolution process, such as insertions, deletions, and substitutions of nitrogenous bases [1].

The k-difference primer problem is one of such manipulations and consists of, given two sequences α and β and an integer k, find for each position j in α, the shortest substring of α that begins at j and has at least k differences from any substring of β [2]. For instance, assume $\alpha = ACTG$, $\beta = AGCAAG$, and $k = 2$. The substrings $\alpha_{1..3} = ACT$ and $\alpha_{2..4} = CTG$ form the solution, since they have at least two differences from any segment of β. The sequences found in the solution of the k-difference primer problem are candidate regions to contain primers [2]. Primers are short strands of DNA that bind (hybridize) to a DNA sequence and are used by biologists to amplify that sequence in laboratory, through the Polymerase Chain Reaction technique [7]. For instance, in order to identify the causative agent of a disease, it is necessary to select a primer that hybridizes to the DNA sequence of the causative agent and that does not hybridize to the DNA sequence of the infected organism or other pathogens.

This paper presents parallel solutions to the k-difference primer problem, targeting multicore processors and GPUs. The solutions are able to analyze very long sequences in a short execution time. The multicore and GPU solutions achieve speedups up to 5.6 and 72.8, respectively, when compared to the best sequential solution.

The paper is organized in six sections. Section 2 describes two sequential algorithms to the k-difference primer problem and identifies forms to exploit parallelism in it. In Sections 3 and 4 we present our parallel solutions targeting multicore processors and GPUs, respectively. We also describe the optimizations applied to them and analyze their performance. Section 5 reviews previous works in parallel solutions to the approximate string matching problem, which is closely related to the problem studied here. Finally, Section 6 summarizes the results.

2 Solutions to the k-difference Primer Problem

The solution to the k-difference primer problem described in [2] is based on the resolution of several instances of the approximate string matching problem. The latter problem consists in, given two sequences and an integer k, find all occurrences of one sequence in the other with at most k differences. The differences can correspond to insertions, deletions, or substitutions of symbols in the strings. The idea is that, in order to solve the k-difference primer problem, i.e., to find the shortest substrings of α that have at least k differences from any substring of β, we can find the longest substrings of α with $k-1$ differences from β and add one symbol to these substrings [3]. Two solutions to the approximate string matching problem, presented in [4, 5], are adapted and used here as substeps, producing two solutions to the k-difference primer problem, referred as *conventional* and *alternative* solutions.

Figure 1 shows the main function of both sequential k-difference primer solutions. For each position r of sequence α of length m, we solve an instance of the approximate string matching problem, invoking a subroutine which finds the longest prefix of $\alpha_{r..m}$ with $k-1$ differences from all substrings of sequence β, of length n, and add one symbol to the prefix found. At each iteration r, a shorter suffix of α is processed by the subroutine, which returns the length of the prefix found. If no solution is found at a certain iteration, we can stop the execution because the next iterations will not produce solutions either. The conventional and alternative solutions differ only in the algorithm used for the subroutine.

```
*Initializations*
**while** $r \leq m-k+1$ **and not** *stop* **do**
    $c :=$ *longest prefix with differences*$(\alpha_{r..m}, \beta, k)$
    **if** $c \neq 0$ **and** $r+c < m$ **then** Solution $\alpha_{r..r+c+1}$ found
    **else** *stop* $:= 1$
    $r := r+1$
```

Fig. 1. k-difference primer sequential solution, for sequences α and β: m=$|\alpha|$ and n=$|\beta|$

Figure 2(a) and (b) shows the subroutines *longest prefix with differences* used in the conventional and alternative solutions, respectively. The first subroutine computes a dynamic programming matrix D with dimensions $(m+1)\times(n+1)$. The rows and columns of D correspond to symbols of α and β, respectively. The cell $D[i,j]$ represents the number of differences between $\alpha_{1..i}$ and any substring of β ending at β_j. The algorithm searches the highest row i with a cell that satisfies $D[i,j] = k-1$. Using this subroutine, the conventional solution to the k-difference primer problem has time complexity $O(m^2 \times n)$. The space complexity is $O(n)$, since the same matrix D is reused in all invocations of the subroutine, which is optimized in order to reduce the amount of memory needed, by allocating only one row for D, and reusing it for all iterations of the outer loop.

```
Initializations
for $i := 1$ to $m$ do
    for $j := 1$ to $n$ do
        if $\alpha_i \neq \beta_j$ then
            $t := 1$
        else
            $t := 0$
        $D[i,j] := \min \begin{cases} D[i,j-1]+1 \\ D[i-1,j]+1 \\ D[i-1,j-1]+t \end{cases}$
        if $D[i,j] = k-1$ then
            $c := i$
return $c$
```

(a)

```
Initializations
for $e := 0$ to $k-1$ do
    for $d := -e$ to $n-1$ do
        $row := \max \begin{cases} L[d-1,e-1] \\ L[d+1,e-1]+1 \\ L[d,e-1]+1 \end{cases}$
        $row := \min(row, m)$
        while $row < m$ and $row + d < n$
            and $\alpha_{row+1} = \beta_{row+1+d}$ do
            $row := row + 1$
        $L[d,e] := row$
        if $e = k-1$ and $L[d,e] > c$ then
            $c := L[d,e]$
return $c$
```

(b)

Fig. 2. Subroutine *longest prefix with differences*, used in the (a) conventional and (b) alternative solutions, computes dynamic programming matrix D and L, respectively, in order to find the longest prefix of α with $k-1$ differences from all substrings of β

The subroutine *longest prefix with differences* used in the alternative solution (Figure 2(b)) computes a dynamic programming matrix L with dimensions $(n+k+2)\times(k+2)$. The rows and columns of L correspond to diagonals of matrix D and number of differences, respectively. A diagonal d of D is formed by cells $D[i,j]$ such that $j-i=d$. The cell $L[d,e]$ represents the highest row i of D such that $D[i,j]=e$ and $D[i,j]$ belongs to diagonal d. Then, e is the number of differences between the prefix $\alpha_{1..L[d,e]}$ and any substring of β that ends at $\beta_{L[d,e]+d}$. The algorithm searches the maximum value in column $k-1$ of matrix L, which represents the highest row i of D with a cell that satisfies $D[i,j]=k-1$. Using this subroutine, the alternative solution to the k-difference primer problem has time complexity $O(m^2 \times n \times k)$. The space complexity is $O(n+k)$, since the same matrix L is reused in all invocations of the subroutine, which is also optimized in order to reduce the amount of memory needed, by allocating only one column for L, and reusing it for all iterations of the outer loop.

2.1 Optimizations and Preliminary Results

We developed two optimizations that can reduce the number of cells of matrices D and L that need to be computed in the conventional and alternative solutions. In the first optimization, referred as *optimization 1* and applied only to the conventional solution, when executing the subroutine *longest prefix with differences* (Figure 2(a)), if we find a row i of matrix D, such that $D[i,j] \geq k$, for all j, we conclude we have already found the solution and there is no point in computing the remaining cells of D. This optimization is implicit in the alternative solution, since matrix L is computed only up to column $k-1$. The second optimization, referred as *optimization 2*, is applied only to the alternative solution. When executing the subroutine *longest prefix with differences* (Figure 2(b)), if we find an occurrence of α in β with less than k differences, we conclude no solution will be found and we do not compute the remaining cells of L.

We evaluated the sequential solutions and the proposed optimizations on a computer with an Intel Xeon quad-core processor and 32GB RAM, using GCC with -O3 optimization option. In all experiments, our biological input data set consists of DNA sequences homologous to the IL1RAPL1 gene, from *Homo sapiens* chromosome X, and obtained from the HomoloGene database, available at NCBI (National Center for Biotechnology Information) [9]. In a final experiment in Section 4.3, huge sequences are used.

Table 1 compares the conventional and alternative sequential solutions and the optimizations applied. The execution times correspond to the arithmetic mean of several executions, which produced a standard deviation of only 1.04. The table also shows the number of comparisons of symbols from α and β performed by each solution. The conventional non-optimized solution computes the matrix D entirely in every call to the subroutine *longest prefix with differences*, leading to approximately 230 trillions comparisons, which make its execution impracticable. Optimization 1 applied to this solution enables an almost 100-fold reduction in the number of comparisons. Despite having a higher worst-case time complexity than the conventional solution, the alternative solution produced smaller execution times, since the computation based on diagonals produces less comparisons than the conventional solution. Optimization 2 applied to the alternative solution produces a small reduction in the comparisons, since it allows us not to compute completely only the last matrix L. Therefore, it results in a slightly shorter execution time.

Table 1. Evaluation of sequential solutions and optimizations: execution time and number of comparisons of α and β symbols, for $|\alpha|$=43,606, $|\beta|$=241,494, and k=100

Solution+optimization	Execution time (s)	# of symbol comparisons ($\times 10^{10}$)
Conventional	*	∼22 960
Conventional+1	12 216.1	∼243
Alternative	8 376.7	∼137
Alternative+2	8 230.9	∼137

* Not measured due to extremely long execution time.

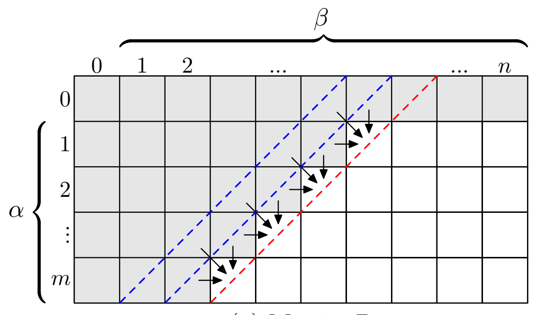
2.2 Exploiting Parallelism

Although they have polynomial time complexity, both conventional and alternative solutions can be very computationally demanding, due to the long length of biological sequences. Therefore, we seek high performance solutions that compute cells of the dynamic programming matrices in parallel, in order to reduce the execution time. Analyzing the data dependences for computing these cells, we identify two forms to exploit parallelism in the k-difference primer problem.

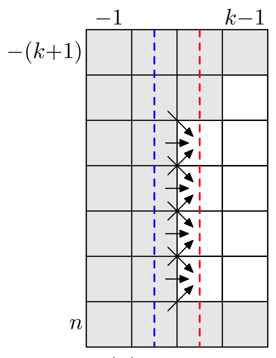
We can execute in parallel different calls of the subroutine *longest prefix with differences* (for both conventional and alternative solutions), since the computation of each matrix is independent from the others. This way, several matrices (D or L) are computed in parallel, which we call *coarse-grained parallelism.* We can also exploit *fine-grained parallelism* by computing different cells in a same matrix (D or L) in parallel. Figure 3(a) and (b) illustrates the computation of matrices D and L, respectively, in conventional and alternative solutions. The arrows represents data dependences. We can compute in parallel all cells in a same anti-diagonal of D (or column of L), since they are independent from each other, while different anti-diagonals of D (or columns of L) are computed sequentially. Both forms of parallelism can be exploited in conjunction.

3 OpenMP Solutions to the *k*-difference Primer Problem

Based on the two forms of parallelism identified, we developed parallel solutions to the k-difference primer problem, targeting multicore processors and using the OpenMP parallel programming model [11]. Figure 4(a) shows how to exploit coarse-grained parallelism in the conventional and alternative solutions, using OpenMP. The while loop of Figure 1 must be transformed into a for loop, so we can use the directive *omp parallel for*, which creates a parallel region and distributes the loop iterations among the threads. Each thread calls different instances of subroutine *longest prefix with differences*, computes different matrices D or L and produces its results separately from other threads. The flag *stop* is shared among the threads, in order to stop the execution of subsequent loop iterations when, at a certain iteration, no solution is found.



(a) Matrix D



(b) Matrix L

Fig. 3. Data dependences and fine-grained parallelism: (a) cells in the same anti-diagonal of matrix D computed in parallel in conventional solution; (b) cells in the same column of matrix L computed in parallel in alternative solution

Figure 4(b) shows how to use OpenMP to exploit fine-grained parallelism in subroutine *longest prefix with differences* of the alternative solution. The directive *omp parallel for* is used around the inner loop, creating a parallel region and distributing the iterations among the threads. Each thread computes different cells in a column of matrix L. At the end of this region, an implicit barrier synchronization guarantees that successive columns are computed sequentially. The directive *omp critical* creates a critical section and ensures the shared variable c (which holds the result) is updated with mutual exclusion by the threads.

```
#pragma omp parallel for \
schedule(static,1) \
shared(result,stop) private(c) \
firstprivate(alpha,beta,m,n,k)
for(r = 0; r <= m−k; r++)
  if(r < stop){
    c = longest_prefix_w_differences(
          alpha,beta,k);
    if((c != 0) && (r+c < m))
      result[r] = r+c;
    else stop = r;
  }
```

(a)

```
for(e = 1; e <= k; e++)
  #pragma omp parallel for shared(L,c) \
  schedule(static) private(row)\
  firstprivate(alpha,beta,k,m,n,e)
  for(d = k−e+1; d < n+k; d++){
    row = max(L[d−1][e−1],
        L[d+1][e−1]+1, L[d][e−1]+1);
    if(row > m) row = m;
    while((row<m)&&(row+d−k<n)&&
        (alpha[row]==beta[row+d−k]))
      row = row+1;
    L[d][e] = row;
    if(e == k)
      #pragma omp critical
      if(row > c) c = row;
  }
```

(b)

Fig. 4. OpenMP implementations exploiting: (a) coarse-grained parallelism for the conventional and alternative solutions; and (b) fine-grained parallelism in the subroutine *longest prefix with differences* of the alternative solution

In order to exploit fine-grained parallelism in the conventional solution, using OpenMP, the subroutine *longest prefix with differences* in Figure 2(a) must be adapted in order to compute matrix D by anti-diagonals. An outer loop computes successive anti-diagonais sequentially, while an inner loop computes the cells in a same anti-diagonal. The directive *omp parallel for* is used around this inner loop, creating a parallel region and distributing the iterations among the threads, so that each thread computes different cells in an anti-diagonal of D. An implicit barrier synchronization at the end of this parallel region guarantees that the next anti-diagonal is not computed before the current one is completed.

The optimization that reduces the amount of memory needed to compute matrix D (or L), used in the sequential solutions, is also applied to the OpenMP solutions (for simplicity, it is not shown in Figure 4). If we exploit only coarse-grained parallelism, one row of D (or column of L) is needed. Exploiting fine-grained parallelism, we need three anti-diagonals of D (or two columns of L).

3.1 Optimizations and Preliminary Results

Optimization 1 (from Section 2.1) can also be applied to the conventional coarse-grained OpenMP solution. Nevertheless, it cannot be applied to the conventional fine-grained OpenMP solution because the solution scans matrix D by anti-diagonals and the optimization needs to check D by rows. Since this optimization produced a huge reduction in the symbol comparisons, the execution of this solution, even in parallel, was impracticable. Optimization 2 (from Section 2.1) can be applied to both alternative coarse- and fine-grained OpenMP solutions.

In order to reduce the synchronization overhead, another optimization, referred as *optimization 3*, is applied to OpenMP fine-grained conventional and alternative solutions. It eliminates the critical section that guards the shared variable c update shown in Figure 4(b). A vector with one position for each thread is used, so that each thread stores its result in a different position of the vector, instead of sharing the variable c. Therefore, the critical section is no longer needed. After finishing the execution of the nested loops, a small loop finds the maximum value of the vector, which is then assigned to variable c.

Table 2 compares the parallel OpenMP conventional and alternative solutions, exploiting coarse- and fine-grained parallelism, and the optimizations applied. We used the same biological input data employed for the sequential solutions, as well as the same platform (a quad-core processor now running 8 threads). The speedups compare the parallel solutions to the best sequential one (alternative+optimization 2). The alternative coarse-grained solutions achieved better performance than the conventional coarse-grained solution with optimization 1 because it performs less symbol comparisons, as we have seen in Table 1. Optimization 2 applied to the alternative solution produces a very small reduction in the execution time. Comparing coarse- and fine-grained approaches used in alternative solution+optimization 2, the latter produces worse results, because it is not suitable to the processor reduced number of cores. Nevertheless, when optimization 3 is applied to the alternative fine-grained solution, it produces a significant improvement in performance, almost doubling the speedup and showing the impact of the synchronization overhead reduction. Given the limited parallel processing capability of the quad-core processor, the combination of coarse- and fine-grained parallelism produces worse results.

Table 2. Evaluation of OpenMP parallel solutions and optimizations, using 4 cores and 8 threads: execution time and speedup wrt. best sequential solution, for $|\alpha| = 43{,}606$, $|\beta| = 241{,}494$, and $k = 100$

Parallel solution+optimizations	Execution time (s)	Speedup
Conventional coarse-grained+1	2 067.2	4.0
Alternative coarse-grained	1 509.8	5.4
Alternative coarse-grained+2	1 481.1	5.6
Alternative fine-grained+2	3 091.2	2.7
Alternative fine-grained+2+3	1 544.1	5.3
Alternative coarse/fine-grained+2+3	2 114.8	3.9

Figure 5 shows how the performance of the best OpenMP solution (alternative coarse-grained+optimization 2) scales as the number of threads increases, from 1 (sequencial solution) to 8, using the quad-core processor. The speedups compare the parallel solution to the best sequential one (alternative+optimization 2). The speedup grows linearly with the number of threads, up to four threads, matching the number of cores available. For 6 or 8 threads, the speedup grows more slowly, since the threads have to share the cores.

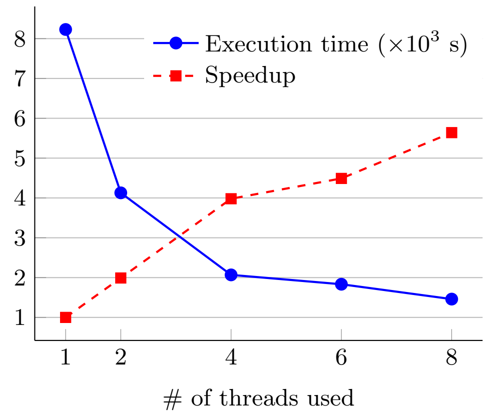


Fig. 5. Performance of best OpenMP solution (speedup wrt. best sequential one), using 4 cores and varying the number of threads, for $|\alpha| = 43,606$, $|\beta| = 241,494$, and $k = 100$

4 CUDA Solutions to the k-difference Primer Problem

We also developed parallel solutions to the k-difference primer problem, targeting GPUs and using the CUDA programming model [10]. Since GPUs have a large number of cores, they are suitable for exploiting both forms of parallelism identified. Each matrix D or L is computed by a block, in parallel to the other matrices, exploiting coarse-grained parallelism. The threads in a block compute the cells in an anti-diagonal of D (or a column of L) in parallel, exploiting fine-grained parallelism. If the number of matrices that have to be computed is larger than the maximum number of blocks which can be created in a kernel invocation, we compute these matrices in batches. We have one kernel invocation for each batch, where the matrices in the same batch are computed in parallel and successive batches are executed sequentially.

We started with two base CUDA solutions, derived from the conventional and alternative approaches. In both solutions, the data structures used for sequences α and β and matrix D or L are allocated in GPU global memory, since the sequences are extremely long and GPU shared memory has limited capacity. Optimization 1, presented in Section 2.1, cannot be applied to the conventional CUDA base solution because this solution computes matrix D by anti-diagonals and the optimization needs to check D by rows. When optimization 2, presented in Section 2.1, is applied to the alternative CUDA solution, it worsens the execution time, therefore we discarded it. Since it provides only a small reduction in the symbol comparisons and the GPU computes many more comparisons in parallel than the sequential and OpenMP solutions, the intrinsic optimization overhead surpassed the performance gain.

Optimization 3, presented in Section 3.1, is applied to both conventional and alternative CUDA solutions. For each block computing a different matrix, we keep a vector in GPU shared memory with one position for each thread to store its result in a different position. Therefore, no synchronizations are needed. Before finishing the kernel execution, a reduction operation is performed in parallel by the block threads, in order to find the maximum value of the vector, which is the result for this matrix.

4.1 Tiling Optimization

In order to take advantage of GPU memory hierarchy, the tiling technique, referred as *optimization 4*, is applied to both conventional and alternative CUDA solutions. Matrix D (or L) is divided into tiles, so that, inside a tile, we compute in parallel the cells in an anti-diagonal of D (or column of L), however successive tiles are computed sequentially. Therefore, we no longer have to allocate an entire anti-diagonal of D (or column of L), in order to compute the tile cells. The data structure used to keep this cells is reduced and can be allocated in GPU shared memory, avoiding access to global memory which is much more slower.

Figure 6(a) and (b) shows matrix D and L, respectively, split into tiles, where the tile size is determined by the number of threads per block used. The thick lines in the figure represent the tiles separation. A tile in matrix D has a rectangular shape, while in matrix L it has a parallelogram shape, due to the different data dependences pattern in this matrix. The shaded areas in the figure represent the cells that must be saved after finishing to compute a tile, and that are used for computing the next tile. For the conventional solution and matrix D, this structure has size $O(n)$, which is the length of input sequence $|\beta|$. Consequently, this structure must be allocated in GPU global memory. However, for the alternative solution and matrix L, this structure has size $O(k)$ (the minimum number of differences) and can be allocated in shared memory.

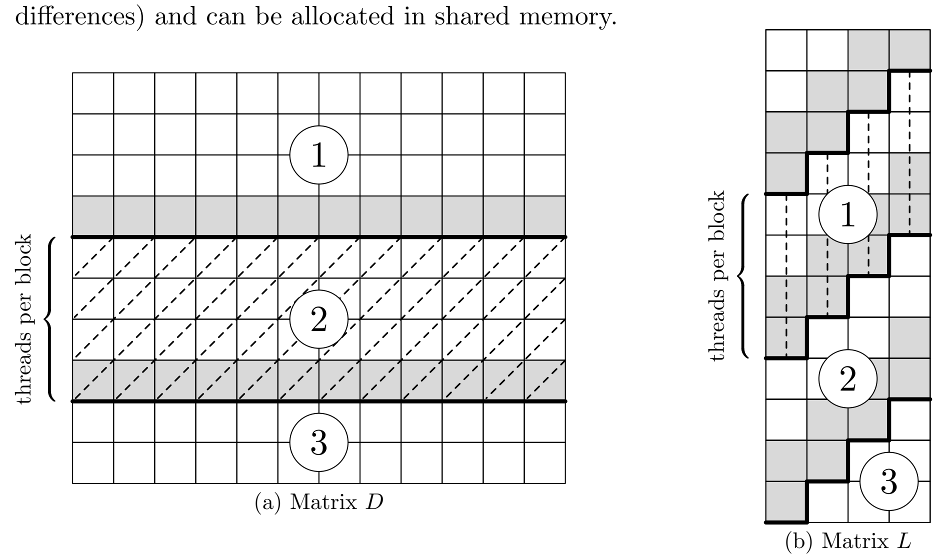


Fig. 6. Tiling technique: (a) matrix D split into rectangular tiles for conventional CUDA solution; (b) matrix L split into parallelogram-shaped tiles for alternative CUDA solution. Tile size determined by the number of threads per block and successive tiles computed sequentially

Another advantage of optimization 4 is that it enables us to apply optimization 1 on the conventional CUDA solution. Inside a tile, matrix D is computed by anti-diagonals and optimization 1 needs to check D by rows. However, we can check the cells saved after computing a tile (shaded cells) and, depending on the result, we do not compute the remaining tiles.

4.2 Prefetch and Speculation Optimizations

Sequences α and β are allocated in GPU global memory because they are too long to fit on shared memory. However, before computing each tile, we can prefetch to shared memory segments of theses sequences that will be used during the tile calculation, in order to avoid access to global memory and to improve performance. We refer to the prefetch of segments of α and β from global memory to shared memory as *optimizations 5* and *6*, respectively.

In the conventional CUDA solution, the division of matrix D into tiles allows us to know exactly which segment of α is used for computing a tile, then we prefetch this segment to shared memory before computing the tile and no access to α in global memory is done while computing it. However, the entire sequence β is used while computing a tile of D, which prevents us from using prefetch on it, which remains being accessed from global memory. In the alternative CUDA solution, we cannot predict which symbols of α and β will be used while computing a tile of matrix L, therefore we developed a speculative mechanism, which prefetchs to shared memory segments of α and β that are likely to be used. During the tile calculation, when a thread accesses a symbol of α or β which is present on shared memory, we have a hit. Otherwise, we have a misprediction in our speculation mechanism and the global memory must be accessed.

Analyzing the alternative solution, we conclude that the initial symbols of α are the most likely to be used, while computing any tile of L. Therefore, we prefetch the initial segment of α from global to shared memory only once and use it for all tiles. We estimate the length $prefetch_\alpha$ of this segment as $\left\lceil \frac{c \times k}{threads\ per\ block} \right\rceil \times threads\ per\ block$, since the minimum number of differences k affects the number of symbols of α and β used. The constant factor c is determined through an experiment in Section 4.3 and we round the value to be multiple of the number of threads per block. We allocate on shared memory a segment of β of length $prefetch_\beta$, estimated as $prefetch_\alpha + threads\ per\ block$. However, before computing each tile, a new segment with only *threads per block* symbols is prefetched. During the tile calculation, this segment and previously prefetched ones are used.

4.3 Results

The execution platform used for the CUDA solutions consists of a GPU NVIDIA GeForce Quadro M4000 with 8GB RAM and 1664 cores, connected to the same computer (which acts as a host) used for the sequential and OpenMP solutions. The same biological input data set is used. The *nvprof* profiler tool was used to extract performance metrics used in the experimental evaluation. In all CUDA solutions, the time spent transferring data between host memory and GPU global memory is insignificant, when compared to the time spent executing the kernel invocations.

Figure 7 shows the performance of all conventional and alternative CUDA solutions, applying optimizations incrementally. The execution times were obtained through the arithmetic mean of several executions, which produced a standard deviation of only 0.02. The speedups compare the parallel solutions to the best sequential one (alternative+optimization 2).

The tiling technique (optimization 4) has a great impact improving the performance of the conventional solution, while its impact on the alternative solution seems to be lower. The difference is that the tiling technique enables the application of optimization 1 on the conventional solution, reducing the number of matrix D cells computed. This optimization is implicit in the alternative base solution. Sequence α prefetch (optimization 5) has a great impact improving the performance of the alternative solution, however its impact on the conservative solution is small. The reason is that, in conventional solution, we prefetch a segment of α from global to shared memory before computing each tile, while in alternative solution, we prefetch a segment of α only once and use it for all tiles.

Table 3 compares the best conventional and all alternative CUDA solutions with respect to several performance metrics. The throughput measure *comparisons per second* indicates how many comparisons of α and β symbols are performed by the solution in one second. Figure 8 shows the number of load and store operations performed in GPU global and

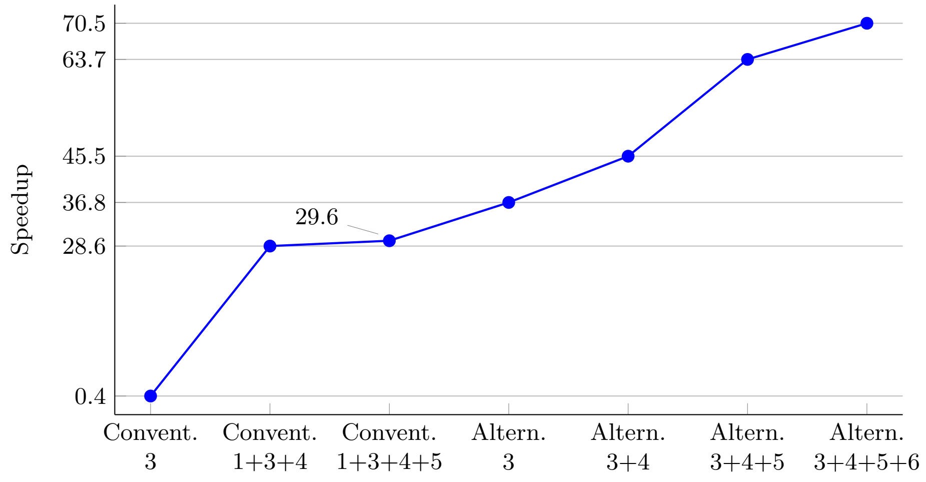


Fig. 7. Speedups of conventional and alternative CUDA solutions and optimizations, wrt. to the best sequential solution, for $|\alpha| = 43,606$, $|\beta| = 241,494$, and $k = 100$

shared memory by these solutions. Analyzing these results in conjunction enables us to evaluate the effect of the optimizations on the metrics and their impact on the solutions performance.

Table 3. Comparison of best conventional and all alternative CUDA solutions, for $|\alpha| = 43,606$, $|\beta| = 241,494$, and $k = 100$

Performance metrics	Convent. 1+3+4+5	Altern. 3	Altern. 3+4	Altern. 3+4+5	Altern. 3+4+5+6
# of symbol comparisons ($\times 10^{10}$)	∼274	∼137	∼137	∼137	∼137
Comparisons per second ($\times 10^{8}$)	∼98.5	∼61.5	∼76.0	∼106.5	∼118.0
# of instructions executed ($\times 10^{10}$)	∼910	∼402	∼432	∼366	∼352
Instructions per cycle	3.3	1.8	2.4	2.9	3.1
Issue slots utilization*	76 %	43 %	54 %	63 %	65 %

* Percentage of issue slots that issued at least one instruction.

All alternative solutions have better performance than all conventional ones. Even though the best conventional solution achieves good instructions per cycle rate and issue slot utilization, it performs nearly twice as many symbol comparisons as the alternative solutions. The best conventional solution has its memory access optimized, but even so it performs many more memory operations than the best alternative solution. The conventional solution keeps on global memory the cells that are saved, after computing a tile, and used for computing the next one, while the alternative solution uses shared memory for this. Besides, sequence β prefetch (optimization 6) is not applied to the conventional solution.

Each alternative solution achieves a better performance than the previous one, although they compute the same number of symbol comparisons. It produces a higher comparisons per second rate, executes more instructions per cycle, and increases the issue slots utilization. As more optimizations are applied, the number of access to global memory decreases a lot, while the access to shared memory increase more slowly. Using the tiling technique (optimization 4), the data structure used for computing matrix L is kept in shared memory, reducing drastically the access to global memory. The load and store operations in global memory become almost restricted, respectively, to access to α and β symbols and to saving the results produced. Prefetching α and β segments from global to shared mem-

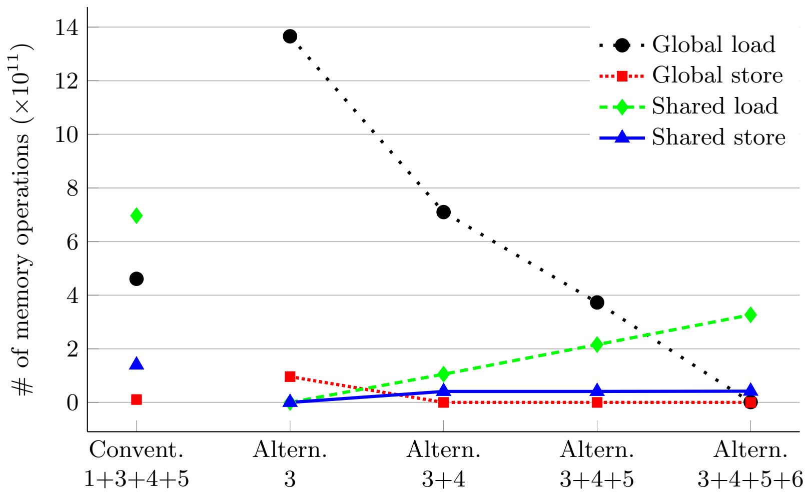


Fig. 8. GPU global and shared memory load and store operations of best conventional and all alternative CUDA solutions, for $|\alpha| = 43,606$, $|\beta| = 241,494$, and $k = 100$

ory (optimizations 5 and 6, respectively) exchanges many global memory load operations for less load operations in shared memory. Optimization 5 is more effective in improving performance than optimization 6 because we prefetch a segment of α only once and use it for computing all tiles, while a segment of β is prefetched for each tile computed.

The performance gain produced by the speculative prefetch mechanism depends on the accuracy of the prediction. We prefetch to shared memory segments of α and β, of lengths $prefetch_\alpha$ and $prefetch_\beta$ respectively, that are likely to be used. However, if a thread accesses a symbol of α or β that is not present on shared memory, we have a misprediction and an access to global memory is done. We measure the miss rates, which indicate the percentage of access that generate misses with respect to the total number of access to α or β, for different lengths $prefetch_\alpha$ and $prefetch_\beta$, defined using the constant factor c. Table 4 shows the results obtained using the best alternative solution and varying c from 1 to 3. For $c = 1$, the miss rate for α is very high, reflecting on the worst execution time. For $c = 3$, there are no more mispredictions in both α and β access, even though the prefetched segments are not very long, and we have the best execution time.

Table 4. Speculative prefetch of α e β symbols, using best alternative CUDA solution, for $|\alpha| = 107,280$, $|\beta| = 2,220,391$, $k = 256$, and 256 threads per block: constant factor c defines length $prefetch_\alpha$ and $prefetch_\beta$ of prefetched segments; miss rates correspond to symbols not found in shared memory and accessed in global memory

c	Execution time (s)	Prefetch$_\alpha$	Prefetch$_\beta$	Miss rate for α	Miss rate for β
1	8 193.3	256	512	49.0 %	0.1 %
2	7 453.7	512	768	1.5 %	0.0 %
3	7 401.4	768	1 024	0.0 %	0.0 %

Figure 9 shows the results obtained using the best alternative solution and varying the number of threads per block used for executing the GPU kernels. The number of threads per block defines the tile size and represents a compromise between exploiting fine- and coarse-grained parallelism. Using more threads per block, we compute in parallel more cells in a same matrix, however, the number of active blocks per GPU multiprocessor is reduced, so we compute fewer matrices in parallel. We reach the best trade-off for 256 threads per block, which produced the best execution times, for both $k = 50$ and 100.

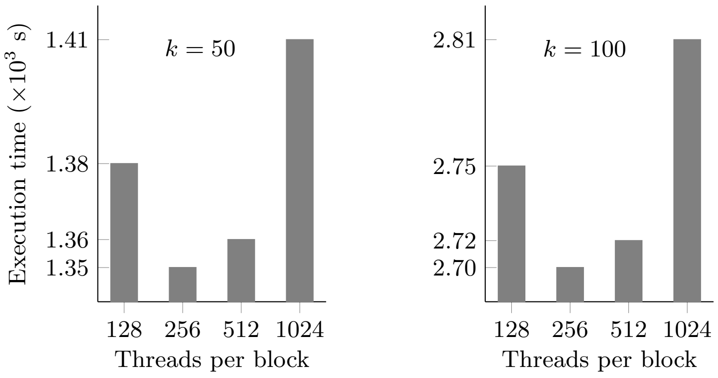


Fig. 9. Execution time of best alternative CUDA solution, for $|\alpha| = 107,280$, $|\beta| = 2,220,391$, and $k = 50$ and 100, varying the number of threads per block: best trade-off between fine- and coarse-grained parallelism achieved for 256 threads per block

Figure 10 compares the performance of the best sequential, OpenMP and CUDA solutions, for five test cases with huge sequences from our biological input data set. We generated synthetic sequences only for the last test case, with $|\alpha| = 10 \times 10^5$ and $|\beta| = 40 \times 10^5$. Due to the long execution time, the sequential solution was not executed for last two test cases. These execution times were estimated based on the speedups obtained for the other test cases. The CUDA solution reaches speedups that go from 70.0 to 72.8 as the lengths of α and β and the number of differences k increase, while the OpenMP solution speedups remain constant at approximately 5.6. The CUDA solution achieved a 100% GPU occupancy, indicating that the hardware resources were used properly.

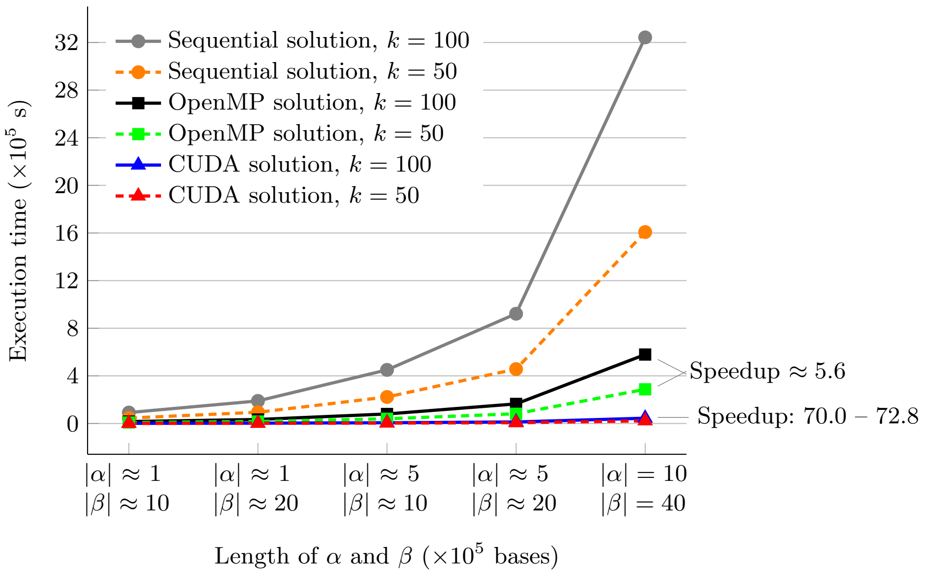


Fig. 10. Performance of best sequential, OpenMP and CUDA solutions, for sequences α e β of different lengths: speedups wrt. sequential solution

Table 5 shows the throughput measure comparisons per second produced by the best alternative sequential, OpenMP and CUDA solutions, stressing the performance gain achieved by the parallel solutions.

Table 5. Comparisons per second of best sequential, OpenMP and CUDA solutions, for $|\alpha| =$ 107,280, $|\beta| = 1,088,386$, and $k = 100$

Best alternative solution	Sequential	OpenMP	CUDA
Comparisons per second ($\times 10^8$)	$\sim$1.6	$\sim$9.4	$\sim$118.5

5 Related Work

We have not found in the literature works with parallel solutions to the k-difference primer problem. Nevertheless, there are parallel solutions to the approximate string matching problem, which is a substep in our problem.

Landau and Vishkin [4, 5] present a parallel algorithm to approximate string matching based on suffix trees, with a theoretical time complexity of $O(\log m + k)$ using n processors. Nakano [8] proposes solutions to the problem of finding the substring of β with the minimum number of differences from α. Two theoretical parallel computing models are used to reflect GPU memory hierarchy features, and the solutions have time complexity of $O(\frac{n \times m}{w} + n \times l)$ using m threads and w memory banks with memory latency l.

In [13] the authors present GPU and FPGA solutions to approximate string matching based on regular expression operators. The dynamic programming matrix is split into regions which are computed in parallel. Using a GPU NVIDIA Tesla C2070 and a FPGA Xilinx Virtex-4, they achieved speedups of 8,3 and 2,9, respectively, with respect to the sequential implementation, for a pattern of length 320. For a pattern of length 3200, the speedup was 18, using the GPU, while the FPGA solution could not be executed due to hardware limitations.

Rastogi and Guddeti [12] describe a GPU solution to approximate string matching of several short patterns to a long sequence with at most two differences, using the Burrows-Wheeler transform (BWT). Using a GPU NVIDIA NVS 300, they achieved speedups up to 8, compared to the sequential implementation, without taking into account the time spent with the BWT. In [6] the authors present GPU solutions to approximate string matching, using the Hamming distance, instead of the edit distance, to compute the number of differences. Therefore, insertions and deletions in the sequences are not allowed, resulting in a simpler algorithm where only symbol substitutions are allowed. Using a GPU NVIDIA GeForce GTX 260, the best solution reached speedups between 40 and 80, with respect to the sequential implementation, while the other solutions achieved speedups of nearly 10.

6 Conclusion and Future Works

This paper presented parallel solutions to the k-difference primer problem, targeting multicore processors and GPUs. For both platforms, we developed several optimizations that allowed the analysis of very long sequences consuming a reduced execution time. To the authors' knowledge, these are the first parallel solutions proposed for the k-difference primer problem.

Starting with two different algorithms, we identified two forms of exploiting parallelism in the k-difference primer solutions, coarse- and fine-grained parallelism. Among the OpenMP solutions, the alternative coarse-grained solution was the one that produced the best performance results, reaching speedups up to 5.6, when compared to the best sequential solution and using a quad-core processor. Given the reduced number of cores, the fine-grained parallelism is not adequate for this platform.

Several optimizations were applied to the CUDA solutions in order to improve performance. The synchronization overhead is reduced by allowing each thread to produce its result separately and using a parallel reduction operation to find the final result. The tiling technique enabled the solutions to handle input data sets with very long sequences and reduced drastically the global memory access. A speculative prefetch mechanism improved even more the use of the GPU memory hierarchy and reached an accuracy of 100%.

The best CUDA solution produced impressive speedups up to 72.8, with respect to the best sequential solution, and this performance is not affected when the sequences length and the number of differences k increase. The best sequential, OpenMP, and CUDA solutions reached the throughput of 0.16, 0.94, and 11.85 billions comparisons per second, respectively, emphasizing the performance gain of the CUDA solution. Since this solution reached 100% of GPU occupancy, if executed on a more powerful GPU, it would achieve an even better performance, because more matrices and cells would be computed in parallel.

Despite the significant results achieved with our parallel solutions, an interesting research subject is to investigate other sequential algorithms for the k-difference primer problem and how to map them to a parallel platform.

References

1. Baxevanis, A., Ouellette, B.: Bioinformatics – A Practical Guide to the Analysis of Genes and Proteins. John Wiley & Sons, 3rd edn. (2005)
2. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
3. Ito, M., et al.: A polynomial-time algorithm for computing characteristic strings under a set of strings. Systems and Computers in Japan **26**(3), 30–38 (1995)
4. Landau, G., Vishkin, U.: Introducing efficient parallelism into approximate string matching and a new serial algorithm. In: Proceedings of the Annual ACM Symposium on Theory of Computing. pp. 220–230 (1986)
5. Landau, G., Vishkin, U.: Fast parallel and serial approximate string matching. Journal of Algorithms **10**(2), 157–169 (1989)
6. Liu, Y., et al.: Parallel algorithms for approximate string matching with k mismatches on CUDA. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum. pp. 2414–2422 (2012)
7. Mandoiu, I., Zelikovsky, A.: Bioinformatics Algorithms: Techniques and Applications. John Wiley & Sons (2008)
8. Nakano, K.: Efficient implementations of the approximate string matching on the memory machine models. In: Proceedings of the International Conference on Networking and Computing,. pp. 233–239 (2012)
9. NCBI: National Center for Biotechnology Information. `https://www.ncbi.nlm.nih.gov`
10. NVIDIA Corporation: CUDA Parallel Computing Platform. `http://www.nvidia.com.br/object/cuda_home_new_br.html`
11. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 4.5. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`
12. Rastogi, P., Guddeti, R.: GPU accelerated inexact matching for multiple patterns in DNA sequences. In: Proceedings of the International Conference on Advances in Computing, Communications and Informatics. pp. 163–167 (2014)
13. Utan, Y., et al.: A GPGPU implementation of approximate string matching with regular expression operators and comparison with its FPGA implementation. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications. pp. 1–7 (2012)