Thread-cooperative, Bit-parallel Computation of Levenshtein Distance on GPU

Alejandro Chacón*, Santiago Marco-Sola†, Antonio Espinosa*, Paolo Ribeca†, and Juan Carlos Moure*

Abstract—Approximate string matching is a very important problem in computational biology; it requires the fast computation of string distance as one of its essential components. Myers' bit-parallel algorithm improves the classical dynamic programming approach to Levenshtein distance computation, and offers competitive performance on CPUs. The main challenge when designing an efficient GPU implementation is to expose enough SIMD parallelism while at the same time keeping a relatively small working set for each thread.

In this work we implement and optimise a CUDA version of Myers' algorithm suitable to be used as a building block for DNA sequence alignment. We achieve high efficiency by means of a cooperative parallelisation strategy for (1) very-long integer addition and shift operations, and (2) several simultaneous pattern matching tasks. In addition, we explore the performance impact obtained when using features specific to the Kepler architecture.

Our results show an overall performance of the order of tera cells updates per second using a single high-end Nvidia GPU, and factor speedups in excess of 20× with respect to a sixteen-core, non-vectorised CPU implementation.

Index Terms—SIMD, GPU, CUDA, Myers' algorithm

1 Introduction

Current DNA sequencing technologies produce billions of sequence reads, with hundreds of bases per read, in a single instrument run. Downstream data analysis for resequencing projects requires alignment (or *mapping*) of all these reads to a reference genome. Sequencing errors, sequence divergence from the reference and the growing length of reads require efficient approximate pattern matching algorithms.

Recent sequence alignment software tools, like BWA [10] or GEM [16], use a two-step alignment strategy. The first step (based on a seeded search in the case of BWA, or on filtration in the case of GEM) extracts substrings from the query (or *read*); such substrings are searched in the reference genome (which has been previously turned into an indexed form allowing fast pattern matching, for instance an FM-index [3]) generating candidate match positions. The second step uses online approximate string matching [18] to verify the similarity between the query and the region adjacent to every candidate position; it returns as valid matches the regions that differ from the query, in terms of some string distance, by less than a value specified by the user.

In the context of biological sequence alignment one often employs *Levenshtein distance*, i.e. the minimum number of *edit operations* needed to transform the query into the match. Each operation can be either a substitution, or an insertion, or a deletion of a single character. Levenshtein distance is typically evaluated in terms of *dynamic programming* (DP, [22]), which casts the problem into the computation of (a subset of) a suitable integer-valued matrix. Improving upon a vast previous literature, Myers [17] devises an algorithm to compute the DP matrix using bit-wise operations; each multi-bit operation can handle several matrix cells simultaneously, thus reducing both the total computational work and memory storage requirements.

*Universitat Autònoma de Barcelona, Bellaterra 08193, Spain.
†Centro Nacional de Análisis Genómico, Barcelona 08028, Spain.
{alejandro.chacon, antoniomiguel.espinosa, juancarlos.moure}@uab.es
{santiagomsola, paolo.ribeca}@gmail.com

CUDA-enabled Graphic Processing Units (GPUs) are high-performance, cost-effective and power-efficient many-core architectures appropriate for accelerating the execution of a wide range of algorithms [11]. They provide overall peak computation throughput and memory bandwidth about an order of magnitude higher than general-purpose latency-oriented processors. GPUs need Single Instruction Multiple Data (SIMD) or vector parallelism and Multiple Instruction Multiple Data (MIMD) parallelism to feed their computational cores. They also strongly rely on H/W multithreading (excess of Thread-Level Parallelism, or TLP) to hide the latency of memory accesses and pipeline dependencies. In general an algorithm must exhibit massive and regular data-level parallelism, converted into both SIMD/vector parallelism and TLP, to achieve high GPU execution efficiency.

The larger computation capability of GPUs comes at the expense of having an order of magnitude less on-chip memory capacity (registers and cache memories) than that of CPUs. As a result, GPUs put more pressure on reducing the overall working set of running threads to make it fit into fast on-chip memory; applications with moderate ratio between computation and memory operations will need to reuse data stored in fast on-chip memory not to become memory-bound.

A typical read-mapping job turns billions of query sequences into tens of billions of candidate regions. This provides plenty of task-level parallelism in the form of multiple DP matrix calculations. While *inter-task parallelism* is a simple way of benefiting from the MIMD and H/W multithreading capabilities of GPUs, however, it is not adequate to efficiently exploit their SIMD/vector potential. In addition, running a pattern matching task per thread would not scale with the query size, due to the impossibility of fitting the working sets of the threads into available on-chip GPU memory even for relatively short queries.

Within the low-memory DP framework of Myers', we propose and analyse a scheme to make several threads cooperate on one or multiple pattern matching tasks (through *intra-task parallelism*). This approach allows us to tune the

amount of data per thread, which enables the efficient usage of GPU registers and shared memory. We test different cooperative mechanisms, among them the new Kepler *shuffle* instruction.

Finally, we present a performance analysis methodology to identify the most relevant bottlenecks of our GPU algorithm. From it, we derive a new solution that uses register memory effectively by means of thread cooperation, and we are able to (1) overcome the memory-bandwidth bottleneck and (2) achieve a more efficient use of computational resources.

Our main contributions can be summarised as follows:

- We develop an algorithmic approach to solve the problem of computing Levenshtein distance in a thread-cooperative way, suited to a SIMD-based computational model. It relies upon a fast method to communicate carries by means of collective very-long integer add and shift operations
- We provide a CUDA-specific implementation of our algorithm, describing our optimisation strategies on the GPU
- We present an in-depth performance analysis showing that our CUDA code is computation-bound and scalable, and more efficient than simpler task-parallel CPU and GPU implementations. Performance is on the order of TCUPS (Tera Cells Updated Per Second).

In section 2 we review some terminology and prerequisites about Levenshtein distance, Myers' algorithm and GPU architectures. Section 3 contains our parallelisation proposal (first, by using a task-parallel approach, and next by introducing a thread-cooperative approach). In section 4, we present the experimental results we obtain when benchmarking our proposal on several GPU systems. Section 5 discusses related work and, finally, section 6 summarises our results, describing future work.

2 Background

2.1 Computing Levenshtein distance

Let Σ be an alphabet of size σ, and the *pattern* $P_{[1..m]}$ and the *text* $T_{[1..n]}$ two strings over Σ. DNA strings generated by sequencing machines can usually be represented with the alphabet {A,C,G,T,N}, where A,C,G and Ts encode bases adenine, cytosine, guanine and thymine, respectively, and N indicates a base which is unknown due to some technical problem occurred during sequencing.

Levenshtein distance can be computed with DP techniques by using the following recurrence [22] to fill a score matrix C, with $0 \leq i \leq m$ and $0 \leq j \leq n$:

$$\begin{cases} C_{i,0} = i, \;\; C_{0,j} = 0 \\ C_{i,j} = min\{C_{i-1,j-1} + \delta(i,j); C_{i-1,j} + 1; C_{i,j-1} + 1\} \end{cases} \tag{1}$$

where δ(i,j) is 0 if $P_{[i]} = T_{[j]}$, and 1 otherwise. A score value $C_{m,j} = k$ identifies an occurrence of P with Levenshtein distance k, ending at text character $T_{[j]}$. An example of score matrix is given in Table 1.a. The time complexity of the classical DP algorithm is $O(nm)$, i.e. proportional to the number of cells in matrix C.

We define the *maximum allowed error rate* as $\epsilon = k/m$.

2.2 Myers' bit-parallel algorithm

Ukkonen [25] noticed that adjacent values in matrix C can differ at most by ±1. A matrix of differences equivalent to C can be represented using two bits per cell. Table 1.b shows a matrix of vertical differences, Δv, where $\Delta v_{i,j} = C_{i+1,j} - C_{i,j}$. Myers [17] used these adjacency properties to exploit bit parallelism and compute difference cells using bit-wise logical, shift, and addition operations. Time complexity becomes $O(n)$ if an m-cell column of Δv fits into a computer word of size w (typically w=32 or 64). Otherwise, a block strategy achieves complexity $O(n\lceil m/w \rceil)$. Hyyrö et al. [7] improved Myers algorithm by reducing the number of bit-wise operations.

Function $\delta()$ can be implemented using a *query profile* (see Table 1.c). Each of the σ different columns is a bit-vector codifying the occurrences of each letter into the query. Also, if matrix C is constructed column-wise only one column needs to be kept in memory at a time, resulting in total memory space requirements of $O(\sigma \times m)$ (measured in bits).

Algorithm 1 shows pseudo-code for Myers' proposal. The main program and variables are at the top, while the time-consuming code, invoked once for each of the n columns, is at the bottom. PV and NV are w-bit vectors encoding positive and negative differences in a given column. Text T is scanned symbol by symbol, and each symbol $T_{[i]}$ determines the appropriate query profile in PEq[]. Function $advance_block()$ executes 17 logical/arithmetic operations to transform the input (i.e. the previous column encoded as PV and NV) into the next column, i.e. to compute m new vertical cells. It also provides a carry (the last cell in the column), which is the penalty to be added to the alignment score.

The basic algorithm assumes $m \leq w$ and is depicted in Fig.1. Patterns larger than w can be partitioned into w-bit blocks [17]. The block-based strategy needs to generate and

TABLE 1: Dynamic Programming tables for sequences P=TAGAC and T=ATCGAG

		A	T	C	G	A	G
	0	0	0	0	0	0	0
T	1	1	0	1	1	1	1
A	2	1	1	1	2	1	2
G	3	2	2	2	1	2	1
A	4	3	3	3	2	1	2
C	5	4	4	3	3	2	2

(a) C: Score Matrix

+1	+1	0	+1	+1	+1	+1
+1	0	+1	0	+1	0	+1
+1	+1	+1	+1	-1	+1	-1
+1	+1	+1	+1	+1	-1	+1
+1	+1	+1	0	+1	+1	0

(b) Δv: vertical-differences

	A	C	G	T
T	1	1	1	0
A	0	1	1	1
G	1	1	0	1
A	0	1	1	1
C	1	0	1	1

(c) Query profile $\equiv \delta()$

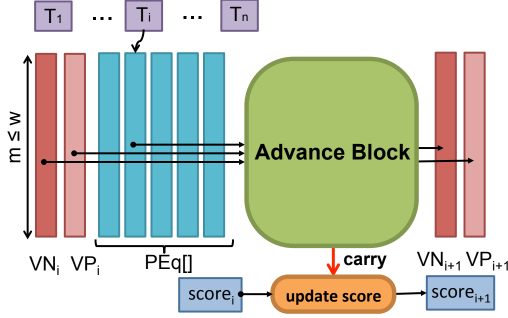

Fig. 1: Core operation of Myers' basic algorithm

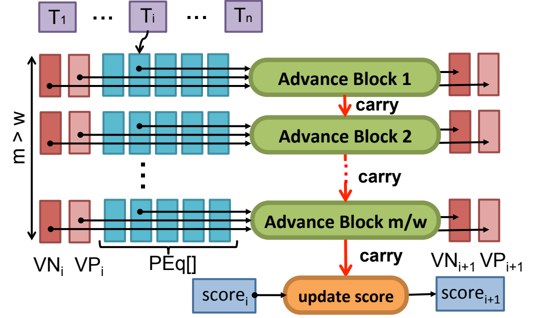

Fig. 2: Myers' blocked-based algorithm

send special carries between consecutive blocks, as shown in Fig.2. This is achieved by means of a slightly modified version of function *advance_block*().

Algorithm 1: Myers' algorithm for $m \leq w$

```
input : P=pattern, T=text, m=|P|, n=|T|, σ=|Σ|
output: (minScore, position) with lower # differences
begin
    bitvector<w> PV, NV, HMASK, EQ, PEq[ σ ]
    ( PV, MV ) ⟵ ( ~0, 0 )
    HMASK ⟵ 1 ≪ (m − 1)
    PEq[ σ ] ⟵ preprocess( P, σ )
    for i=1 to n do
        EQ ⟵ PEq[ T[i] ]
        (c, PV, NV) ⟵ advance_block(EQ, PV, NV)
        score ⟵ score + c
        if ( score < minScore ) then
            ( minScore, position ) ⟵ ( score, i )
        end
    end
    return ( minScore, position )
end

Function advance_block (bitvector<w> EQ, PV, NV )
begin
    bitvector<w> XV, XH, PH, NH
    XV ⟵ EQ | NV
    XH ⟵ ( ( (EQ & PV) + PV ) ∧ PV ) | EQ
    PH ⟵ NV | ~ ( XV | PV )
    NH ⟵ PV & XH
    carry ⟵ ( PH & HMASK ) − ( NH & HMASK )
    PH ⟵ PH ≪ 1
    NH ⟵ NH ≪ 1
    PV ⟵ NH |  ( XV | PH )
    NV ⟵ PH & XV
    return ( carry, PV, NV )
end
```

2.3 GPU Architectures

GPUs are composed of tens of processing components, called streaming multiprocessors (SMs) by Nvidia [11]. SMs share a L2 cache of hundreds of KBytes, and an external global memory of several GBytes. Each SM contains hundreds of SIMD cores that perform in-order execution of instructions. Each SM contains tens of KBytes of local storage that is partitioned into explicitly-managed registers and shared memory banks, and several implicitly-managed cache memories.

Tens of thousands of threads must be launched simultaneously to achieve high performance. The CUDA programming model is based on a hierarchy of threads executing the same program on different sets of data. A *thread-block* is a group of threads that may cooperate using the registers and shared memory available in a given SM. Thread-blocks in a *grid* (or *kernel*) are scheduled non-deterministically for independent MIMD execution into SMs. A thread-block is divided into batches of 32 threads, called *warps*, which are the smallest scheduled unit. Between 32 and 64 warps from one or multiple thread-blocks are dynamically scheduled for execution in the same SM. This mechanism, often known as H/W multithreading, is the main latency-hiding strategy on GPUs.

A warp is executed in a SIMD/vector fashion: threads in a warp are executed in a lock-step manner operating on 32 values in parallel. If threads in the same warp need to follow different control flows, all paths must be executed one after another, with some threads active and the remaining threads stalled. An instruction executed by a subset of the warp threads is said to be divergent. Divergence is an inherent performance limitation of SIMD architectures, and must be addressed when designing the algorithm.

Another critical performance issue is the memory access pattern of the algorithm. When executing a SIMD/vector load or store instruction, the memory addresses provided by all the threads in the same warp are combined, or *coalesced*, to generate one or multiple memory block access requests (memory blocks of 32 to 128 Bytes). High memory performance is achieved only when all the data requested from global memory is really needed by the program. In practice, that means requested data is coalesced into one or a few memory blocks. Warp-level, cooperative instructions may help reducing the need for global and shared memory accesses.

3 Parallelisation Analysis

This section describes and discusses two CUDA implementation strategies for Myers' bit-parallel algorithm: (1) task-parallel and (2) thread-cooperative. The work presented addresses the computation of Levenshtein distance for DNA strings, but can easily be extended to different alphabets.

3.1 Task Parallel Approach

We assume there is a large number of input sequence reads, and each query must be compared to multiple regions in a large genome text. Having lots of independent query-text comparisons provides a straightforward source of task parallelism. This approach has been used on GPUs very recently in [9] [24]. We have developed our own implementation, putting our best effort on optimising the code. Apart from some implementation details described at the end of this section, the most performance-critical issue is handling the local storage for each task.

Bit-vectors PV, NV and PEq[] are accessed n times during the algorithm execution. For the sake of performance, it is important to reuse this intermediate data, keeping them in on-chip memory and avoiding costly main memory transfers. The problem is that the aggregated size of this intermediate data grows both with the query size and with the number of running threads. For moderate and large query sizes either (1) memory performance suffers because intermediate data exceed the available on-chip GPU memory, or (2) GPU occupancy is sacrificed to make intermediate data fit into on-chip memory. Section 4 evaluates performance when storing intermediate data either in local memory or shared memory.

3.2 Thread Cooperative Approach

One way to deal with the previous problem is making threads cooperate on the same task (intra-task parallelism) so that the amount of intermediate data per thread is reduced. Another advantage of thread cooperation is to enable the allocation of GPU registers for all intermediate variables. Registers provide more storage capacity and throughput than any other kind of on-chip memory.

3.2.1 Intra-task SIMD vectorisation: 1 warp per task

Finding enough intra-task parallelism to be efficiently exploited by even a single warp (SIMD operation) is challenging. Dynamic programming approaches present a well-known dependence pattern: any cell of the score matrix can be computed only after the values of the left and above cells are known.

There is potential parallelism when computing cells on the same anti-diagonal, but it is difficult to exploit, since it grows and diminishes as the anti-diagonal enlarges and shrinks while traversing the score matrix. Having said that, Myers' method for computing Levenshtein distance is interesting, as it allows processing all cells in a column simultaneously.

We revisit Myers' idea to exploit bit parallelism not only at the word level, but also at the SIMD level. Each thread (or SIMD lane) holds a word-size slice of the column information stored in bit-vectors PEq[], PV and NV. This scheme reduces and fixes the total local memory required per thread, which is now independent of m, the query size. Then, the CUDA compiler can easily allocate registers for the local data of each thread.

Most of the bit-wise operations on Algorithm 1 are inherently parallel (and, or, xor, not ...) and are trivially converted to SIMD/warp instructions. The exceptions are the add and shift operations inside $advance_block()$ function. Algorithm 2 depicts the pseudo-code of our proposed thread-cooperative m-bit addition and shift operations. Each thread executes the code, receives a portion of each bit-vector input and generates a portion of the output. The cooperative shift requires one extra carry propagation step between neighbour threads. The cooperative addition uses a simple ripple-carry scheme. First, all threads perform a bit-wise addition of their corresponding portion of the input. Then, a cooperative loop of communication and carry addition steps iterates until no carries need to be propagated. Most times, it takes just one or two loop iterations to complete.

It is not surprising to find that most of the complexity falls in the addition operation. Indeed the "magic" of Myers's method resides in converting cell dependencies into the carry dependencies within the addition operation. This strategy ultimately benefits from the very efficient hardware implementation of the addition operation, which solves the carry chain dependence very quickly.

The 1-$warp$-per-$task$ strategy works reasonably well for certain query sizes, but fails with others. Fig. 3.a shows how a query of size m=400 is partitioned into 13 words, and exactly 13 threads cooperate on the matching task while the remaining 19 threads are idle. This case involves a disappointing thread utilisation of 39%. The next step to achieve high GPU performance requires the threads in a warp to cooperate on processing several queries.

3.2.2 Intra- and Inter-task SIMD: 1 warp per r tasks

Combining intra- and inter-task parallelism enables two types of performance improvements. First, several small queries may be used to "fill" a 1024-bit SIMD vector and provide useful work for as many threads in a warp as possible. Fig. 3.b shows how r=2 queries of size m=400 occupy 2×13=26 words (and threads), with an utilisation that raises to 78% (800 bits used from 1024).

Algorithm 2: Thread-Cooperative m-bit Addition and Shift functions executed by each thread

```
Function thread_cooperative_add (bitvector<w> a, b)
begin
    bitvector<w> result
    (result, c_add) ← a + b
    while (check_any_thread (c_add != 0)) do
        next_c ← send_to (threadID+1, c_add)
        (result, c_add) ← result + next_c
    end
    return (result)
end

Function thread_cooperative_shift (bitvector<w> a)
begin
    bitvector<w> result
    c_shft ← a ≫ (w - 1)
    next_c ← send_to (threadID+1, c_shft)
    result ← (a ≪ 1) | next_c
    return (result)
end
```

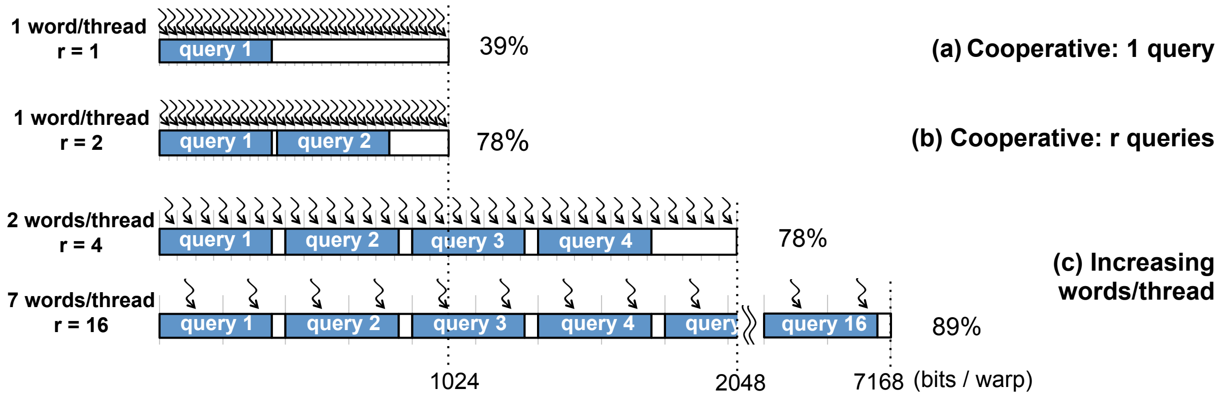

Fig. 3: Thread Cooperation: r queries (m=400) and varying #words processed per thread

Second, we can use a larger number of queries per warp in order to increase the total work per thread. In this case, more words are handled by each thread, as measured by the quantity $words/thread$. Increasing work per thread helps reducing query fragmentation and increase SIMD efficiency. Fig. 3.c shows examples for $words/thread$=2 and 7 (r=4 and 16 queries), with thread utilisation rising to 89%.

But the most important advantage of increasing the amount of work per thread is the reduction of the total number of overhead instructions: those not included in the 17 bit-wise original operations in Myers' algorithm. The extra instructions needed for inter-thread carry propagation represent an important portion of this overhead. The drawback of increasing $words/thread$ is that the amount of local memory required per thread also increases; this may compromise the efficient usage of GPU registers and GPU occupancy. As we will show in the next section, the best $words/thread$ configuration depends on the query size but also on the GPU architecture. An extreme thread-cooperative configuration with r=32 is in fact purely task parallel, as there is no actual need of thread cooperation. This option, however, only makes sense for small queries. An advantage with respect to previous proposals is that the static declaration of variables allows using GPU registers instead of local memory.

The mechanism to let several threads cooperate on several queries requires identifying those threads responsible of the last slice of each query. They must be inhibited on carry propagation phases, but are responsible for generating the final result for each query.

3.3 Optimisation details

We simplify the inner code loop as much as possible to reduce the amount of divergence and instruction overhead. We help the compiler to generate non-divergent code by replacing conditional control flow structures by computation.

Since the input text can be very large, it is stored in binary form, with several symbols packed into a single w-bit data word. Divergence appears when threads access multiple text regions simultaneously and extract symbols from different positions of a data word. We apply a loop peeling optimisation [14] to move the extra control instructions and the associated divergence out of the main loop.

Additionally, divergence and instruction overhead outside the main loop is further reduced by extending text regions to start and finish in aligned locations.

Query pre-processing is moved out of the main code, so that each query is preprocessed just once, and not once for each candidate text region. All query profiles are created and stored into global memory before running the comparison code. For small alphabet sizes, like DNA, query profiles are just slightly larger than the original query strings.

Special GPU assembly instructions ($addc$ and $add.cc$) implement carry propagation for local extended additions. Also, the Kepler-specific *funnelshift* instruction is used to propagate the carry in extended shift operations.

Thread-cooperative operations are implemented using thread communication at the warp level. We take advantage of the warp's lock-step execution to avoid synchronisation primitives. Several intra-warp communication techniques for carry propagation (shared memory, ballot and shuffle instructions) are implemented and evaluated. The Kepler-specific *shuffle* instruction is the most efficient alternative, with an improvement close to 20%.

4 Experimental Results

We ran several implementations of Myers algorithm on different multi-core and GPU platforms. We first assess overall performance and then present a detailed analysis in order to identify the main architectural bottlenecks.

4.1 Experimental setup and methodology

The experimentation platform is a heterogeneous CPU-GPU node. The CPU is a dual-socket Intel Xeon E5-2650, with eight 2-way hyperthreaded cores per socket running at 2.0Ghz. Most of the GPU measurements were done on an Nvidia GTX Titan with 14 Kepler SMs (993Mhz). We also used a Tesla 2090 with 16 Fermi SMs (1.3 Ghz) and a Tesla K20c with 13 Kepler SMs (705Mhz).

Commonly-used simulation tools [5] [20] are a standard way of providing the query input sets. Each input set contains a million reads. We have used a modified version of GEM [16] to generate all the candidate matching positions in the human genome (GRCh37) for such inputs. The accepted error rate is ϵ=0.2. At most 20 million query-candidate pairs

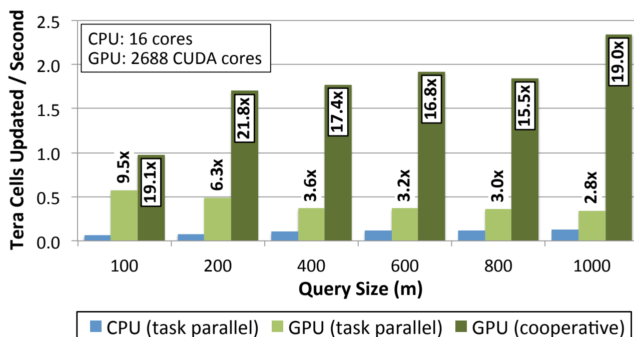

Fig. 4: Performance overview

(i.e., at most 20 candidates per query) are processed. The genome text and query profiles reside in CPU and GPU memory before starting execution measurements. Results are obtained by averaging over the 3 best executions, and expressed in terms of cell update operations per time unit. The variability of the measures is very low (on the level of the 1%).

The multi-core CPU implementation is task-parallel, with 16×2 threads (OpenMP) to exploit hyperthreading, and is not vectorized. GPU implementations set the thread-block size to 128 for Kepler and 256 for Fermi, since they provide the highest performance.

4.2 Overall Performance Results

Fig. 4 shows performance on CPU (task-parallel approach) and GPU (both using task parallelism and thread cooperation) for increasing query sizes (m from 100 to 1000). The presented results correspond to the best-performing configuration for each query size and implementation version.

The thread-cooperative GPU algorithm provides the best performance, surpassing the Tera-CUP barrier (from 1.0 up to 2.3). These results are between 15× and 22× better than those obtained by the multi-core CPU. Additionally, on the GPU the cooperative approach outperforms the task-parallel scheme by 2×-7×.

In general, longer queries provide better relative performance. This is expected since the relative weight of the initialisation phase and parallelisation overheads are reduced. However, the performance of the GPU task-parallel version reduces by a factor of up to 0.6× as the query length increases. This unexpected result is studied in detail in the next subsection. The analysis done helps understanding the reasons behind the thread-cooperative solution results.

4.3 Task Parallel: Performance limiters

The task parallel scheme uses one thread for each query-candidate pair. This is a coarse-grained approach that performs well on a CPU but not on GPUs. In the next sections, we analyse the performance bottlenecks of the GPU implementation, either using local or shared memory to describe the reasons for these results.

4.3.1 *Using Local Memory: high miss rate*

Square bars on Fig.5 quantify how performance degrades up to 1.41× when increasing query size and using local memory. The solid line indicates an increase of 1.7× in the number of GDRAM memory accesses, from 297 to 506 Bytes/cell. There is a clear correlation between increasing the amount of GDRAM accesses and performance reduction.

The amount of local memory needed by the application grows linearly with the number of simultaneous queries and the query size. The number of queries is determined by the total number of threads launched for execution. Increasing query size decreases temporal locality and the L1 and L2 GPU caches become less effective to filter GDRAM accesses. For example, with a query size of m=1000, 94% of L1 and 79.5% of L2 accesses are misses.

Once GDRAM memory is identified as the main performance bottleneck, we need to see if the problem is latency- or bandwidth-bound. We measured empirical GDDR5 bandwidth to be between 185 GB/s and 210 GB/s, which range between 85% and 95% of the maximum bound provided by the Nvidia bandwidth test. Therefore, we conclude that the task-parallel GPU implementation using local memory is bound by GDRAM bandwidth. In contrast, owing to larger on-chip caches the CPU implementation is not memory- but computation-bounded.

4.3.2 *Using Shared Memory: low GPU occupancy*

The classical solution to overcome GDRAM bandwidth memory problems is to foster data reuse by explicitly using

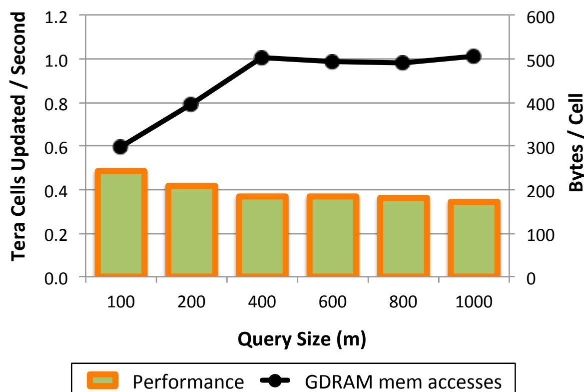

Fig. 5: GPU Task Parallel: local memory

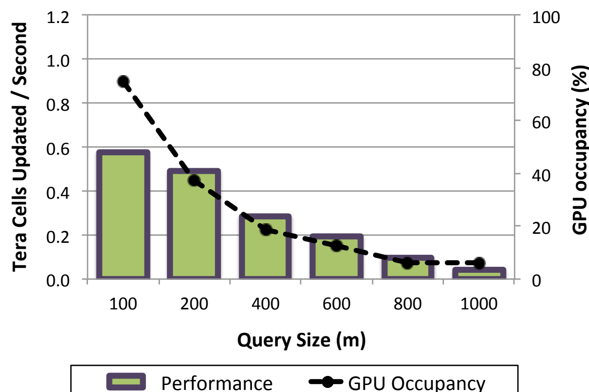

Fig. 6: GPU Task Parallel: shared memory

shared memory. The best performance is achieved when we store columns PV and NV in shared memory, but maintain query profiles, PEq[], in local memory. Measured GDRAM bandwidth values for query sizes m=100, 200, 400, 600, 800, 1000 are now 41.2, 8.30, 1.56, 0.98, 0.72, 0.57 GB/s. Therefore, using shared memory actually prevents GDRAM bandwidth from becoming a bottleneck.

Table 2 compares effective GDRAM memory accesses with an estimation of best local data reuse. The estimation assumes that all data requests imply no additional GDRAM accesses if elements are already placed in on-chip memory.

A task parallel approach with local memory exhibits very limited data reuse. The use of shared memory increases the latter, but there is a significant amount of requests that are still fetched from GDRAM and not from on-chip memory.

Fig. 6 shows the performance of the shared memory implementation. Bars indicate a performance degradation from 1.18× to 13.66× as query size increases. Again, this is due to the higher amount of local data, but now the effect is revealed by a reduction of GPU occupancy (i.e. the percentage of active versus potential running threads, depicted by the dashed line in Fig. 6). Shared memory is a scarce resource that must be assigned equally to each thread. The GPU cannot allocate the same amount of active threads if each thread requires more memory; as a result, GPU occupancy is reduced to levels that strongly reduce overall performance.

Comparing Fig. 5 and Fig. 6 we conclude that using shared memory only benefits small query size cases, $m \leq 200$, when GPU occupancy is high enough to hide memory latencies.

4.4 Thread Cooperative: Performance limiters

We analyse performance and limiting factors of the cooperative approach. We first address the case of assigning a slice of the column to each thread, using one word per thread. Subsequently, we explore the performance advantage of using several words per thread. Finally, we analyse the execution in detail to find out performance bottlenecks.

4.4.1 Cooperation: one word/thread

Fig. 7 presents results for the best combination of m (query size) and r (tasks or queries assigned to each warp). Performance varies between 0.6 and 1.0 TCUPS, always higher than the results obtained with the task parallel approach.

Table 2 shows that the cooperative approach drastically reduces the amount of GDRAM memory accesses, almost reaching the theoretical minimum. In fact, effective measured GDRAM bandwidth is lower than 7 GB/s for all query sizes. Also, all the executions achieve 100% GPU occupancy. Therefore, neither memory nor GPU occupancy are performance bottlenecks here.

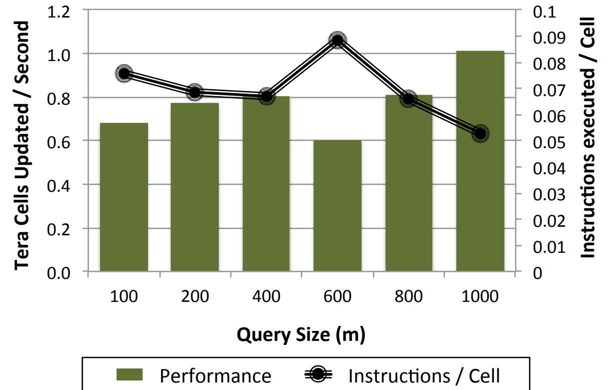

Fig. 7: GPU Thread Cooperative: 1 *word/thread*

We measured the total instruction count (in warp instructions) and computed the cell-normalised rate, that we denote by *instructions/cell*. This metric is depicted by the solid line in Fig.7 and exhibits a strong correlation with performance, which is inversely proportional to *instructions/cell*. This result suggests that GPU execution is now computation-bound.

In fact, the reason for the performance variations discovered in Fig. 7 has to be found elsewhere. Warp instructions can simultaneously operate with 32 bits × 32 threads = 1024 cells. For each query size m, we must adjust the number of simultaneous queries r to use a total number of bits as close to 1024 as possible. Fig. 3 was showing the problem of low thread utilisation. For the cases of Fig. 7, thread utilisation is 78%, 78%, 78%, 59%, 78% and 97%, respectively. Considering that overhead instructions are relatively less frequent for larger query sizes, thread utilisation correlates almost perfectly with *instructions/cell*.

4.4.2 Cooperation: several words/thread

Fig. 8 depicts the performance impact of increasing the amount of work per thread (measured in $words/thread$) by processing more queries per warp. For fixed values of m and $words/thread$ the optimal value of r is derived empirically. Results show performance speedups from 1.22× to 2.70× when increasing the amount of work per thread.

Also for this scenario we carried out an in-depth performance analysis, which can help generating new optimisation ideas. Fig. 9 shows the performance trade-off involved when increasing the amount of work assigned to each thread.

On one hand, *instructions/cell* is reduced between 1.39× and 2.33× when increasing $words/thread$. This is due to the reduction of the instructions devoted to communication and synchronisation among the cooperating threads, and explains why the overall performance increases.

TABLE 2: Ratio of effective GDRAM accesses versus estimated GDRAM accesses

Query size (m)	100	200	400	600	800	1000
Task parallel (Local Mem.)	54071×	145270×	368912×	546418×	724375×	931655×
Task parallel (Shared Mem.)	7515×	3042×	1149×	1082×	1059×	1058×
Cooperative (1 word/thread)	1.60×	1.28×	1.14×	1.11×	1.07×	1.05×

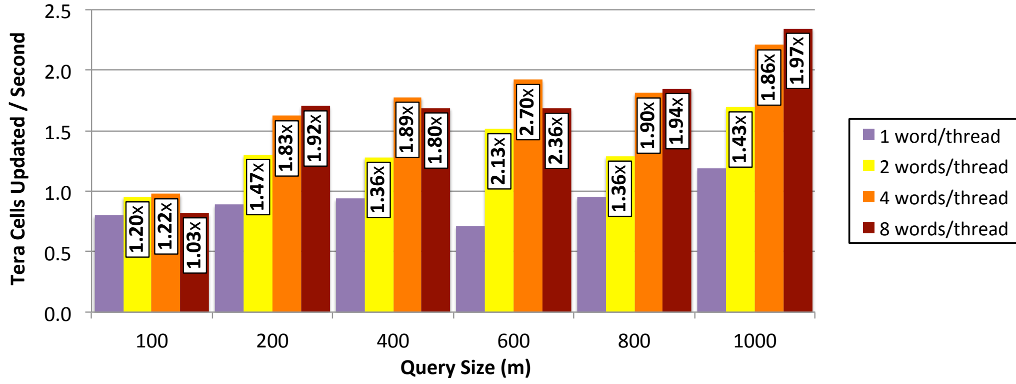

Fig. 8: Performance for varying *words/thread*

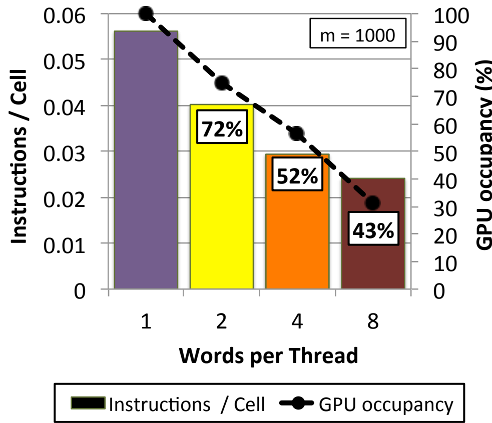

Fig. 9: Impact of varying *words/thread* on *instructions/cell* and GPU occupancy

On the other hand, GPU occupancy falls sharply. As local data increases, more registers per thread are required and, hence, GPU occupancy decreases. In the examples shown in the Figure, the numbers of allocated registers are 28, 38, 56, and 92, respectively. The sharp plunge of GPU occupancy explains why overall performance flattens and even worsens.

In summary, for each query size m one can find a configuration of r (number of queries) and $words/thread$ that maximises performance.

4.4.3 Detailed Performance Analysis

We also measured the performance impact of using Kepler-specific instructions such as *shuffle* and *funnelshift*. Execution time is improved up to 28% and an average of 18%, meaning that Kepler GPUs have an important performance advantage with respect to previous-generation Fermi GPUs.

Table 3 provides data from relevant experiments with selected maximum performance values of m and $words/thread$ to help understand the final performance limits of our GPU implementation. The first row of the table shows the empirical number of bitmap operations needed to compute a column, which varies between ~24 and ~33. The theoretical minimum is 17 bitmap operations [17] but this value does not consider the operations for score calculation, management of conditional structures, synchronisation and memory access. We conclude from those results that the parallelization overhead is limited and acceptable.

The second row of Table 3 shows the ratio between effective and estimated GDRAM accesses. This is a measure of data reuse, which is between 1.02 and 5.39. Effective GDRAM bandwidth is listed in the third row of the Table, and complements previous information. Measured bandwidth is found to be between 1.3 GB/s and 29 GB/s, very far from GPU memory system limits. From those results we conclude that memory reuse is very effective.

Finally, Table 3 shows an IPC (Instructions Per Cycle) value between 2.59 and 4.73. We consider these figures as quite close to the limit: the theoretical architecture maximum is 7, and many sources from Nvidia state that values above 4.5 are rarely obtained in real applications.

As a conclusion, the cooperative solution is computation-bound and exploits all GPU resources very efficiently.

4.4.4 Performance on different GPUs

We have repeated our performance analysis on different GPU architectures, namely Fermi and Kepler. Speedups with respect to the 16-core CPU are also included as a reference in Fig.10. The normalised performance obtained for all the GPUs is between 0.5 and 0.86 GCUPS per core and GHz. For a fixed query size, normalised performance (obtained by factoring out the architectural advantage of the Kepler instructions) is very similar in all three GPUs. This means

TABLE 3: Detailed performance metrics for best performing cases

(m, words/thread)	(100, 4)	(200, 8)	(400, 4)	(600, 4)	(800, 8)	(1000, 8)
Bitmap operations/Column	29.23	26.95	33.35	29.49	30.17	24.09
Effective/Estimated GDRAM accesses	5.39	1.57	1.12	1.07	1.03	1.02
Bandwidth (GB/s)	29.19	7.25	2.69	1.85	1.29	1.29
IPC	2.59	3.66	4.73	4.52	4.00	4.06

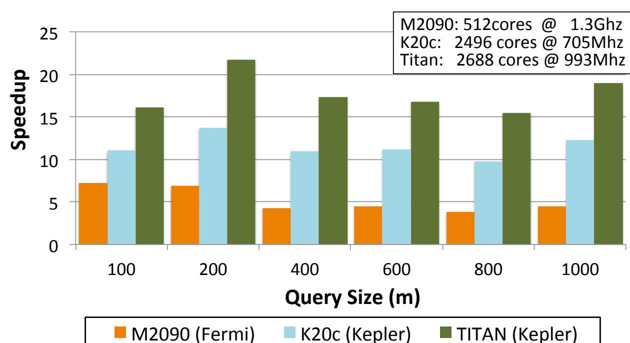

Fig. 10: Speedup of several GPUs vs CPU

that performance scales fairly well with the number of cores and clock frequency, even when using GPUs with different CUDA capabilities (Fermi and Kepler). Such results back the expectation that our proposal will show a good performance scaling even on future, more powerful GPUs.

5 Related Work

Two recent works implement a task-parallel scheme of Myers' algorithm on the GPU. On one hand Langner et al. [9] analyse GPU implementations of Levenshtein and Damerau distances. They aim to integrate the GPU code into SeqAn [1], a library of bioinformatics algorithms and data structures. They propose a task-parallel design, and evaluate the use of shared memory. On the other hand Tristam et al. [24] use a task-parallel design to evaluate the difficulty of the required GPU optimisations. Their setup includes an interleaved layout for the input strings, the usage of 16-Byte loads, and a tuned kernel scheduling. Due to the many differences it's difficult to directly compare such results with ours, but their performance hardly reaches 100 GCUPS.

Both works address the optimal usage of the different GPU memories. They recognise memory bandwidth as the main bottleneck, and the need for cooperative strategies; however, they apparently do not take into account register memory. In fact, with the present paper we demonstrate for the first time that an efficient use of register memory by means of thread cooperation is key to overcome the memory-bandwidth bottleneck. Of note, both works also propose the usage of banded and cutoff techniques [6] to reduce the number of cells to be computed; we too plan to include these techniques into our implementation in the future.

First works on GPU were focused on the Smith-Waterman algorithm [23] for general score functions, using a substitution matrix and affine gap penalties. Manavski and Valle [15] exploited task parallelism by allocating one pairwise sequence alignment task to each single thread. Liu et al. [12] [13] have implemented several versions of CUDASW++. The first version of CUDASW++ [12] adopted a task-parallel approach for small sequences (≤ 3072) and thread cooperation for large sequences, exploiting anti-diagonal (wavefront) parallelism. Hains et al. [4] improve the cooperative version using tiling and more efficient register usage. They achieve performance in the range of 10s of GCUPS. Liu et al. [13] employ GPU SIMD parallelisation using PTX instructions to gain additional data parallelism. They also address concurrent CPU and multi-GPU processing and present a performance of about 120 GCUPS on a single Kepler GPU (GTX 680). However, the usage of general distances prevents the use of simple bit-parallel strategies, making the parallelisation more complex. Farivar et al. [2] apply tiling strategies similar to [4], but using a global alignment algorithm [19].

Bit-level parallelism can also be exploited for the Longest Common Subsequence (LCS) problem, which is similar to, but simpler than, the problem of computing Levenshtein distance. Kawanami and Fujimoto [8] implement the first GPU solution, which exploits task parallelism. Ozsoy et al. [21] propose an improved GPU design that reaches 1 TCUPS using 3 Fermi GPUs. The cooperative scheme proposed in this paper should be useful to obtain a better solution for the LCS problem as well.

6 Conclusions and Future Work

Upcoming sequencing technologies will produce longer reads at reduced cost. This will put additional stress on current sequence alignment algorithms, that will quickly become the bottleneck of the pervasive analysis pipelines used to process resequencing data.

In this work we improve on the GPU Myers' algorithm, which computes the Levenshtein distance between two strings and constitutes a basic block of several popular aligners. Experimental results show that our best implementation obtains on a single GPU performance speedups of $20\times$ with respect to a sixteen-core, non-vectorised CPU version, providing a peak performance of 2.3 TCUPS.

The solution presented here is ready to be efficiently executed on any current GPU. To tune it to the target architecture it is sufficient to adjust the work-per-thread ratio; if more local memory is available on the GPU, an appropriate reconfiguration will improve performance.

From a methodological standpoint, this paper provides an example of how task-parallel CPU approaches can be redesigned into cooperative multi-thread algorithms adapted to many-core architectures like the GPU; the main principle guiding our implementation has been to get the most from local memory system and reduce the number of instructions. We have also demonstrated how specific Kepler architecture instructions can be used to further improve algorithmic performance.

From the standpoint of the analysis of sequencing data, we have shown that GPUs are computational platforms suitable to efficiently implement string-comparison algorithms. Our results indicate that GPUs can become an additional source of computational power in order to perform high-quality alignment of longer sequence reads in acceptable times. As future work, we will implement on the Intel MIC architecture a version of the cooperative-parallel algorithm that uses explicit SIMD instructions; its performance will provide us with a comparison of the benefits offered by the two architectures. Also, we plan to integrate our GPU algorithm into the GEM mapper [16], thus demonstrating the practical relevance of our results.

7 Acknowledgments

This research has been supported by MICINN-Spain under contract TIN2011-28689-C02-01. The authors would like to thank Nvidia for supporting our research with the donation of a K20c Kepler GPU card.

References

[1] A. Döring, D. Weese, T. Rausch, and K. Reinert. SeqAn an efficient, generic C++ library for sequence analysis. *BMC bioinformatics*, 9(1):11, 2008.

[2] R. Farivar, H. Kharbanda, S. Venkataraman, and R. H. Campbell. An algorithm for fast edit distance computation on GPUs. In *Innovative Parallel Computing (InPar), 2012*, pages 1–9. IEEE, 2012.

[3] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398. IEEE, 2000.

[4] D. Hains, Z. Cashero, M. Ottenberg, W. Bohm, and S. Rajopadhye. Improving CUDASW++, a parallelization of Smith-Waterman for CUDA enabled devices. *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011:490–501, 2011.

[5] M. Holtgrewe. Mason - A read simulator for second generation sequencing data. *Technical Report FU Berlin*, 2010.

[6] H. Hyyrö. A Bit-Vector algorithm for computing Levenshtein and Damerau edit distances. *Nord. J. Comput.*, 10(1):29–39, 2003.

[7] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, CPM '02, pages 203–224, London, UK, UK, 2002. Springer-Verlag.

[8] K. Kawanami and N. Fujimoto. GPU accelerated computation of the longest common subsequence. In R. Keller, D. Kramer, and J.-P. Weiss, editors, *Facing the Multicore - Challenge II*, volume 7174 of *Lecture Notes in Computer Science*, pages 84–95. Springer Berlin Heidelberg, 2012.

[9] L. Langner, K. Reinert, and D. Weese. Myers Bit-Vector Algorithm on GPU for SeqAn. Master's thesis, Freie Universität Berlin, 2011.

[10] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: a unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[12] Y. Liu, B. Schmidt, and D. Maskel. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3:93, 2010.

[13] Y. Liu, A. Wirawan, and B. Schmidt. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(1):117, 2013.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *ACM SIGMICRO Newsletter*, volume 23, pages 45–54. IEEE Computer Society Press, 1992.

[15] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[16] S. Marco-Sola, M. Sammeth, R. Guigo, and P. Ribeca. The GEM mapper: fast, accurate and versatile alignment by filtration. *Nature Methods*, 9(12):1185–1188, 2012.

[17] G. Myers. A fast Bit-Vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, May 1999.

[18] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, Mar. 2001.

[19] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol*, 48:443–453, 1970.

[20] Y. Ono, K. Asai, and M. Hamada. PBSIM: PacBio reads simulator—toward accurate genome assembly. *Bioinformatics*, 29(1):119–121, 2013.

[21] A. Ozsoy, A. Chauhan, and D. M. Swany. Achieving teraCUPS on longest common subsequence problem using GPGPUs. In *ICPADS*. IEEE Computer Society, 2013.

[22] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359 – 373, 1980.

[23] T. Smith and M. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147:195–197, 1981.

[24] D. Tristram and K. Bradshaw. Evaluating the acceleration of typical scientific problems on the GPU. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '13, pages 17–26, New York, NY, USA, 2013. ACM.

[25] E. Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6(1):132–137, 1985.