An Algorithm for Fast Edit Distance Computation on GPUs

Reza Farivar, Harshit Kharbanda
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {farivar2,kharban2}@illinois.edu

Shivaram Venkataraman
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Email: shivaram@eecs.berkeley.edu

Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: rhc@illinois.edu

ABSTRACT

The problem of finding the edit distance between two sequences (and its closely related problem of longest common subsequence) are important problems with applications in many domains like virus scanners, security kernels, natural language translation and genome sequence alignment. The traditional dynamic-programming based algorithm is hard to parallelize on SIMD processors as the algorithm is memory intensive and has many divergent control paths. In this paper we introduce a new algorithm which modifies the dynamic programming method to reduce its amount of data storage and eliminate control flow divergences. Our algorithm divides the problem into independent 'quadrants' and makes efficient use of shared memory and registers available in GPUs to store data between different phases of the algorithm. Further, we eliminate any control flow divergences by embedding condition variables in the program logic to ensure all the threads execute the same instructions even though they work on different data items. We present an implementation of this algorithm on an NVIDIA GeForce GTX 275 GPU and compare against an optimized multi-threaded implementation on an Intel Core i7-920 quad core CPU with hyper-threading support. Our results show that our GPU implementation is up to 8x faster when operating on a large number of sequences.

1. INTRODUCTION

Given two strings a and b, the edit distance problem is used to find the fewest operations required to transform a to b. The operations usually allowed are insertion of a character, deletion of a character or substitution of one character for another. This distance is also called the Levenshtein distance and finding the edit distance between two sequences is closely related to finding the longest common subsequence(LCS). Edit distance calculation has applications in many domains like spell checkers, virus scanners [6], security kernels [15], optical character recognition [4] and genome sequence alignment [1].

The traditional method used for accurate global sequence alignment is the Needleman-Wunsch algorithm [12]. This is an example of a dynamic-programming algorithm in which the first phase of the algorithm fills a matrix of 'scores' based on the edit distance. The traditional Needleman-Wunsch algorithm is memory intensive and for sequences of length m and n, the memory required is $O(m.n)$. In the second phase, the algorithm traces the score matrix back to find the optimal alignment. This phase of the algorithm takes different control flow paths based on the contents of the input data it is working on. The combination of these two problems makes it hard to efficiently program them on SIMD (Single instruction Multiple Data) processors, such as GPUs (Graphics Processing Units), with limited memory availability and lack of tolerance for code divergence.

In this paper, we introduce a new algorithm as a variation of the dynamic programming method [3, 12] to greatly reduce its memory requirements and control flow divergence. We use the Needleman-Wunsch alignment algorithm as our baseline, and introduce two major modifications to make the algorithm run faster on GPUs. First, our algorithm divides the problem into independent *quadrants* and makes efficient use of shared memory and registers available in GPUs to store data in between different phases of the algorithm. Secondly, we eliminate any control flow divergence by embedding condition variables in the program logic to ensure all threads execute the same instructions even though they work on different data items.

We present an implementation of this algorithm on an NVIDIA GeForce GTX 275 GPU and compare it against an optimized multi-threaded implementation on an Intel Core i7-920 quad core CPU with hyper-threading support. Our results show that our GPU implementation is up to 9.3 times faster when operating on a large number of sequences. Moreover, our GPU implementation is up to 4 times faster than state of the art implementations [1] (refer to section 5).

The rest of the paper is organized as follows. Section 2 motivates our work by demonstrating the need for a fast pairwise alignment tool and the challenges of implementing it. Section 3 describes the proposed algorithm in detail. Section 4 evaluates the proposed algorithm and compares it to baseline implementations on a multi-core CPU. Related efforts are discussed in section 5, and section 6 concludes the paper.

2. MOTIVATION

Calculation of edit distances between two sequences and aligning the two sequences based on their edit distances is a very compute intensive process. This problem becomes n-fold when the number of sequences on which these operations have to be done are huge. Many fields have hence refrained from using edit-distance as a possible solution to solve problems. This section contains an explanation of the use of this algorithm in a larger context in which this algorithm plays a part, and then describes the challenges in implementing the Needleman-Wunsch algorithm on SIMD architectures, specifically GPUs.

2.1 Pairwise alignment in a larger context

Pairwise alignment can be used to compute the edit distance between two sequences and then align these sequences with an allowed fixed number of insertions, deletions and mismatches. Edit distance computation between two sequences also finds its applications in other domains. [15] uses an edit-distance algorithm to detect correlated attacks in distributed systems.[6] Uses the edit distance technique to identify the type of intrusion. A very common use of edit distance is to find how close two strings are to each other and auto-check the spelling of the word accordingly. Similar approaches could also be used to suggest search strings in search engines. Edit distance is used in speech [4] and evaluating optical character recognition[17]. All of these approaches involve calculation of edit distances between millions of sequences and then the alignment of these sequences.

An interesting application of edit distance and alignment of two sequences is in Bio-Computation. The machines which generate the DNA data have progressed at a much rapid pace than the techniques to analyze this data[13]. This is a profound problem in the bio-computation domain. Today, biologists have terabytes of data but do not have enough computational power and techniques to work on this data. In many ways this problem is similar to the Big-Data problem that the computer industry has been facing for sometime now. There is a need for techniques which could be used to analyze this data rapidly. Accurately aligning the genes against each other allows the comparison of DNAs and gives the researchers ability to draw inferences from these alignments. Our tool allows fast alignment of a large number of short reads quickly.

2.2 A short description of the Needleman Wunsch global alignment algorithm

The Needleman-Wunsch global alignment algorithm [12] is a dynamic programming algorithm that is used for global alignment, meaning that it can be used to find the best alignment possible between two different strings. This is in comparison to local alignment algorithms (for example the Smith-Waterman) algorithm, where the best alignment is among smaller segments of the two strings. In [12] to find the alignment of two strings x and y, a two-dimensional m by n "score matrix" is allocated, where m and n are the lengths of the two strings x and y. In the first phase of the algorithm, the (i, j)'th entry of the matrix contains the optimal score for the alignment of the first i characters in x and the first j characters in y.

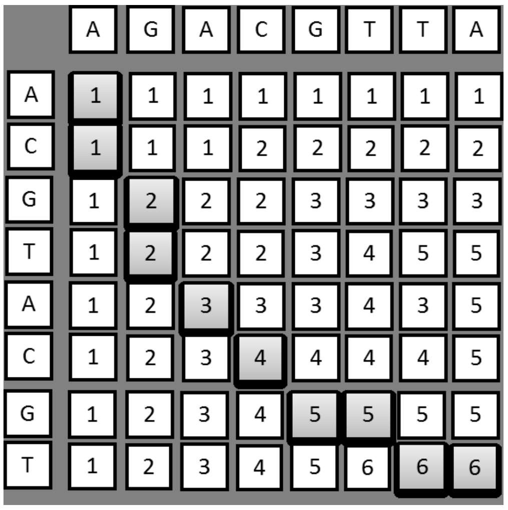

Figure 1: Global alignment using Needleman-Wunsch algorithm

The contents of each cell $a_{i,j}$ is computed based on the i'th character in x, the j'th character in y and the contents of the cells on its left ($a_{i,j-1}$), top ($a_{i-1,j}$) and top-left ($a_{i-1,j-1}$) that are already computed and stored in the matrix. The value of the cell $a_{i,j}$ is computed using the following expression:
$a_{i,j} = Max\{a_{i,j-1} + g, a_{i-1,j} + g, a_{i-1,j-1} + S[x[i], y[j]]\}$
where g is the gap penalty value and $S[x[i], y[j]]$ represents a similarity matrix (a look-up table that represent the penalty of changing one character to another). Figure 1 depicts the contents of the matrix a.

Once the matrix is computed and stored in memory ($O(m.n)$ memory requirement and $O(m.n)$ time), the algorithm starts the backtracing phase from the last location of the matrix ($a_{m,n}$) backwards. At each step, the algorithm picks the cell whose values was used in the previous phase of the algorithm, and goes backwards to the first cell $a_{0,0}$. Each movement upwards means an insertion in string x, and a movement leftwards means an insertion in string y (or a deletion in string x). A diagonal movement would mean either a match or a mismatch. The backtracing phase requires the whole matrix in memory ($O(m.n)$), and takes $O(m+n)$ time to execute.

2.3 Challenges of implementing the Needleman-Wunsch algorithm on GPUs

Implementing the Needleman-Wunsch algorithm on SIMD machines, and more specifically GPUs in this research, is challenging for two main reasons, discussed in the next subsections.

2.3.1 High Memory Consumption

The first problem when implementing the Needleman-Wunsch algorithm on GPUs is the high memory usage of the algorithm, which is of the order of $O(m.n)$. The high memory usage forces the use of slow global memory to store the score matrix. Alternatively, in order to keep the score matrix in the fast memories of the GPU (shared memory or registers), we would need to keep the number of parallel threads running in the GPU limited, which would result in reduced performance.

To further describe these problems, we present a concrete example. Assume that the input strings are 32 long each, necessitating 1024 bytes to store the score matrix. An NVIDIA Tesla GPU of compute capability 1.3 allows for 16KB of shared memory per each streaming multiprocessor (SM). This would translate to 16 threads per SM, which is clearly not enough parallelism to mask even the pipeline latencies of the GPU processors (at least 24 threads required) or completely utilize the parallel processors (at least 32 threads required), let alone memory access latencies. Increasing the input size to 128 long strings would require 16KB to store the matrix, which would mean only 1 thread per SM and deny any parallelism. The situation in the Fermi GPUs is not much better either. Each SM allows access to up to 48KB of shared memory, therefore in our first example we can have 48 simultaneous threads running in each SM. However, the Fermi architecture requires a 4 to 8 times larger SM utilization compared to Tesla architecture to hide the same latencies, therefore the memory pressure in Fermi is even more than the Tesla architecture.

The other possibility is to store the score matrix in the slower global memory. However, the backtracing phase of the Needleman-Wunsch algorithm necessitates non-coalesced memory accesses, since the elements of the score matrix that are read depend on the contents of the data, making the memory access patterns data-dependent. With non-coalesced memory accesses, the threads running in the GPU are serialized and the performance will be hard hit.

As we will describe in the next section, we solve these problems by modifying the algorithm and reducing the memory requirements of the algorithm considerably.

2.3.2 Diverging SIMD flows

As mentioned in the last discussion, the second phase of the Needleman-Wunsch algorithm is inherently data-dependent. In the previous section, we described how this data-dependence affects the memory access patterns if the score matrix is stored in global memory. Another manifestation of this problem is in diverging code flows. Unlike the first phase of the algorithm, in which the same code flow takes place regardless of the data contents, the backtracing phase is completely data-dependent. The back tracing might take anywhere between $max(m, n)$ (when the backtracing always takes the diagonal route) to $m + n$ steps (for example when backtracing first completely moves upwards in m steps and then takes n steps left).

This is a serious problem for a SIMD machine such as a GPU. The whole reason for the massive parallelism potential of SIMD machines is that they sacrifice independent control logic circuity for more computational units. Diverging code flows would make use of the parallelism potential of GPUs impossible. This challenge is one of the main reasons that some of the previous attempts to implement either the Needleman-Wunsch or the Smith-Waterman algorithm in GPUs have reported sub-optimal performance figures.

We describe an algorithmic technique in the next section to solve this problem.

3. DESCRIPTION OF THE ALGORITHM

In this section, we describe an algorithm for global alignment, which solves both of the problems presented in the previous section. Our algorithm is based on the classic Needleman-Wunsch algorithm [12] , but modifies it in a way to reduce memory requirement significantly, trading it off with more computations. Needleman-Wunsch algorithm was the first application of dynamic programming to biological sequence comparison. The runtime and memory requirement of [12] are both $\Theta(m \cdot n)$. This rather high memory usage makes it unsuitable for the SIMD cores of a GPU, since the amount of fast shared memory per computing core is quite limited. A variations of the Needleman-Wunsch algorithm based on the longest common subsequence (LCS) implementation is presented in [5], where they try to reduce the memory usage. In [5], the regular dynamic programming LCS problem is mixed with a divide and conquer strategy, based on the principle of optimality, to reduce the memory footprint of the algorithm to $O(min(m, n))$ while the runtime stays in the same order of $O(m \cdot n)$. In practice this algorithm requires about an order of magnitude more computation, however the algorithmic order remains the same.

A variation of [5] has been proposed in a slightly different way in [3], and our work is based on this algorithm.

3.1 Algorithm Design

The main idea behind our algorithm is that if one divides the Needleman-Wunsch score matrix of two strings in an arbitrary grid of quadrants, the contents of each quadrant can be readily recomputed with only access to the score matrix values on the quadrant's top and left boundary as well the corresponding subsets of the original strings, as depicted in figure 2. In our algorithm, the score matrix is first divided into a virtual grid of quadrants. Then the algorithm will perform a regular Needleman-Wunsch first pass to fill the score table. However, it does not store the contents of any cell other than cells that coincide with the quadrant boundaries. Note that as long as the boundaries are the only goal of this phase, they can be computed using a simplified form of [12] that only keeps two consecutive rows of the matrix for storage space, using $\Theta(2 \cdot m)$ memory space. To store the boundary elements themselves, we need an additional $\Theta(2 \cdot (\frac{m}{k} \cdot n))$ memory, where k is the length of the quadrants. For example, if the two strings are 32 long each, and we set the quadrant sizes to be 8 by 8, then 256 bytes are required to finish the first phase of the algorithm, and another 64 bytes are used to store the values on the quadrant boundaries. The size of the grid is selected such that each quadrant can successfully fit in a small memory that is available in the fast shared memory of GPUs. This means the score matrix of each quadrant can fit in less than 128 bytes of memory, which in turn translates to a high number of parallel threads running at the same time.

The next phase of the algorithm is back-tracing. Remember that the back-tracing phase of the Needleman-Wunsch algorithm follows the 'directions' from the last cell (bottom-right) to the first cell (top-left). In our algorithm we start from the last quadrant ($(3, 3)$ in the previous example). We first load the score matrix values which are located on the boundaries of this quadrant (that were computed and stored in the previous phase). Using the boundary values and the proper segments of the two input strings, we recompute the score values for the rest of the cells of this quadrant with the same forward pass algorithm. With the score values of this quadrant in hand, we start the trace-back phase within this quadrant. The entrance point of the trace-back route is set

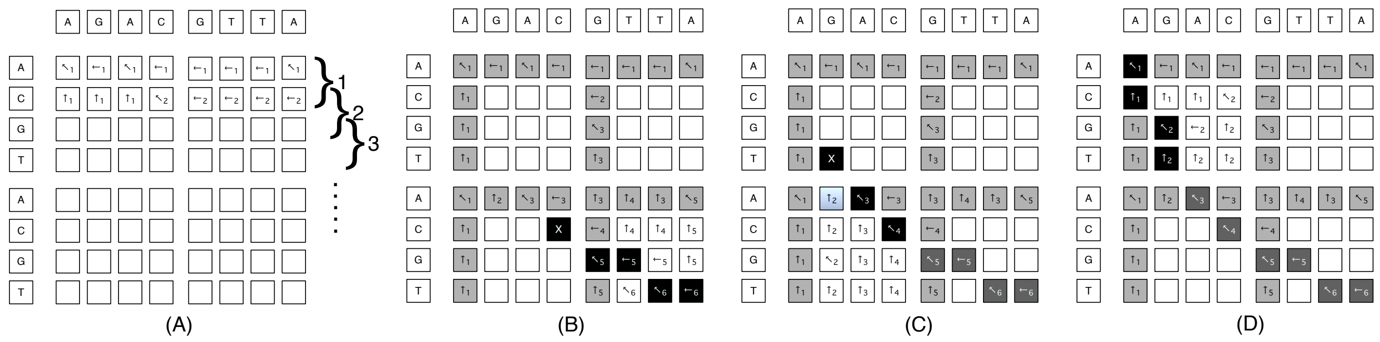

Figure 2: Accurate alignment computation in the GPU. A) The first pass of the algorithm keeps only two active rows of the alignment matrix while scanning it from top to bottom. During this scanning pass, it computes the boundary values of the smaller trivial quadrants for later access by the second pass of the algorithm, shown as shadowed cells in (B). B) The second pass of the algorithm relies on the boundary values calculated in the previous pass. Having these values ready for each quadrant, we can start from the last quadrant and compute the inner values using a simple Needleman-Wunsch dynamic programming variant. We then start tracking back from the last element of the matrix and follow the directions to find the exit cell, denoted by letter 'X'. (C) Keeping a record of the trace-back so far, it is continued in a new quadrant using the exit value of the previous quadrant. (D) The algorithm finally exits the larger alignment matrix through a quadrant either on the left edge or top edge of the alignment matrix.

to the last element of this quadrant, which is also the last element of the larger alignment matrix. From here, we commence the trace-back and follow the trace until it reaches either the left or the top boundary of the quadrant. Note that the exit point from one quadrant is the entry point to the next quadrant. This new quadrant is either on the left, top or top left of the previous quadrant, as the alignment trace-back route is monotonically decreasing from bottom right of the alignment matrix to its top left. Using this exit point, the algorithm loads the boundary values for the next quadrant that contains the traceback route, and the same sequence of steps is repeated for the newly loaded quadrant. The algorithm repeats these steps until it reaches the first cell of the score matrix, at which point the global alignment is computed. This pass of the algorithm is depicted in parts $(B), (C)$ and (D) of figure 2 (shown for a simplified case). The memory requirement of the trace-back phase of our algorithm is $\Theta(k \cdot k)$.

The differences in our algorithm and [3] is as follows. The first difference is that in our algorithm we store the values of the score matrix that are located on every quadrant boundary, where [3] chooses to recalculate them from scratch by performing the first phase of the Needleman-Wunsch algorithm for each quadrant. This results in more re-computation than is really necessary when implementing the algorithm in GPUs, as it does not utilize the available fast shared memory.

In comparison, our algorithm performs the first pass of Needleman-Wunsch only once, and stores all the top and left boundaries of all the quadrants in memory. The reason for this design choice is that even though GPUs have limited amounts of fast shared memory, it would be detrimental to overall performance if they are not properly utilized. In other words, for our target architecture, [3] spends too much processing cycles computing the score matrix over and over, while the boundary values can be successfully stored in memory.

Our implementation is hand-tuned for best performance on GPUs. Starting with an analysis on the available resources of the GPU to maximize its utilization, we find the size of a 'trivial' quadrant, which in our current implementation is set to 8 by 8. As such, we divide the large matrix required for global alignment into 16 quadrants (in case of 32-long strings), numbered from $(0, 0)$ to $(3, 3)$ based on their location in the alignment matrix. The trivial quadrant problem is solved with a simple Needleman-Wunsch algorithm in $\Theta(\frac{m}{4} \cdot \frac{n}{4})$ run time and $\Theta(\frac{m}{4} \cdot \frac{n}{4})$ memory.

3.2 Memory storage for the boundary values

As mentioned in the previous sub-section, our algorithm stores the score matrix values which are located on quadrant boundaries. Therefore it is important to ensure that the storing scheme be fast enough so that it is not a bottleneck for the rest of the algorithm.

The algorithm uses two different storing schemes, based on the size of the strings. In the general case and for longer strings, our algorithm stores the boundary values in the global memory. Since the traceback path is different for each input pair of strings, different quadrants should be loaded, which results in non-coalesced accesses. To alleviate this situation, we devise a technique according to which the algorithm reads all the quadrant boundary rows and columns, one row or column at a time, and stores them temporarily in the shared memory. During this phase, the algorithm stores the required part of the loaded rows and columns in another part of the shared memory. This results in more instructions and memory accesses than is required, but it ensures no code flow divergence and completely coalesced memory accesses. In practice, doing this much more work still results in better performance than just reading the required boundary values for a specific quadrant in each thread, that results in non-coalesced accesses. The size of the strings and hence the score matrix impacts this decision, and at some point the overhead becomes so high that the non-coalesced accesses will perform better, however in our experiments we did not reach this saturation point.

If the strings are small enough (For example $32 \sim 36$ long), we store the whole boundary values in another fast memory storage: registers. There is four times more fast memory available as registers in each SM than shared memory, and our algorithm itself does not utilize all the available registers. For example in a 1.3 compute capability GPU, if there are 192 threads per SM, each can access 84 registers, while our program uses almost 35 registers, and leaves the rest of the registers unused. The problem with using registers is that they cannot be used as an array, only specific variables. Trying to read the specific portion of the boundary rows and columns therefore requires a programmatic structure similar to a switch case, which will result in diverging code flows. The technique we use to solve this problem is similar to the previous case, in which the algorithm reads all the rows and columns one by one from register-stored variables into shared memory, and stores the required parts in another area of shared memory and ignores the rest. Using this technique, we managed to speed up our algorithm even more (refer to section 4 for experimental evaluation).

3.3 Solving the Diverging flows problem

As mentioned earlier, diverging flows present a significant challenge to SIMD architectures. The first phase of the Needleman-Wunsch algorithm is mostly non-divergent, since each thread fills the score matrix with the exact same flow. However, the second phase is inherently data-specific. In some cases, the trace-back phase in a quadrant takes a strictly diagonal path and reaches the exit point in exactly k steps. In other case, the path might first go k steps upwards and then k steps leftwards, for a total of $2k - 1$ steps.

To solve this problem, the algorithm was modified. The algorithm is written such that the trace-back in each quarter takes exactly $2k-1$ traceback steps. However, the algorithm also computes condition variables that evaluate to the zero value once the traceback reaches either the left or top boundary of the quadrant. These condition variables are then embedded in the expressions that find the next cell in the traceback path. In effect, once the traceback reaches either of the left or top boundaries, it gets "stuck" there until the $2k - 1$ traceback steps are finished. Using this mechanism, the same set of instructions are executed in each thread, eliminating any flow divergence among different threads.

The same problem should also be tackled for loading different quadrants, and is addressed in a similar way. Each thread loads the boundaries for exactly $\frac{2 \cdot m}{k}$ quadrants and performs traceback in each of them. However, once a quadrant on the first row or column of the quadrants grid is loaded, a similar mechanism is utilized to make the computation "stuck", while every thread keeps executing similar instructions.

This solution ensures that any combination of input arguments take the "worst case path". However, we utilize a GPU specific mechanism to help the situation. The SIMD execution model of NVIDIA GPUs runs different threads in "warps" of 32 threads in each SM. Therefore, once all 32 threads of the same warp get "stuck", we can break the loop and move on to the next phase, instead of waiting for all of them to reach $2k - 1$ steps. Fortunately, NVIDIA GPUs of compute capability 1.3 or higher provide and intrinsic function *"__all(condition)"*, which evaluates to TRUE once all the individual *conditions* of the warp's threads become TRUE. This mechanism helps our algorithm run faster.

An easier diverging flow problem to solve is the issue of if/else conditions, which might end up in diverging code. To solve this issue each of the if/else conditions in the code is replaced by an expression that executes all the instruction on either side of the condition, and incorporates them in the same expression. For example, assume that we want to make the following if/else condition non-divergent:

if $(cond)$ then $a = f(x)$ else $a = g(x)$
We can replace it with the following expression:

$a = cond \cdot f(x) + !cond \cdot g(x)$
This transformation is feasible since in the C language conditions can be used as numerical values, and are evaluated to zero when condition is FALSE.

Using the above techniques, our algorithm runs completely divergence free, and not a single instruction is serialized during execution.

4. EXPERIMENTAL RESULTS

4.1 Experiment Setup

The experimental evaluations were carried out against several multi-core baseline testbeds. The main baseline system has an Intel(R) Core i7 920 quad-core CPU with a clock frequency of 2.67GHz, a 8MB L3 cache and hyperthreading support (it is represented with 8 virtual cores to the operating system). The GPU used was the NVIDIA(R) GeForce GTX 275 with 895MB of global memory. For strong scaling experiments we used two different 8-core setups. The first one is a dual-processor Mac Pro computer with two Intel(R) Xeon E5462 quad-core CPUs, each running at a clock frequency of 2.8GHZ, 12MB of L3 cache without hyperthreading and 8GB RAM. This allows the baseline to run on 8 separate real cores. The second one is a Linux based server with two Intel(R) Xeon X5355 CPUs, each running at a clock frequency of 2.66 GHz, 8MB of L2 cache and no hyperthreading support with 16GB RAM. The CUDA driver version was 4.10 and the CUDA API version was 4.0. The operating system used for the experiments was Ubuntu 10.10 running the Linux kernel 2.6.35-30 in the first baseline testbed (which hosted the GPU card too), Mac OS 10.6.8 on the second testbed. Our baseline program uses OpenMP to utilize multi-core CPUs, and in all of our experiments the baseline is compiled with *-O3* optimization level.

4.2 Speedup

The first experiment we conducted was between a parallel implementation of the Needleman Wunsch algorithm on multi core CPUs and a GPU implementation of our proposed algorithm with all the computations stored in the shared memory and the registers. For the parallel baseline implementation, we parallelized our implementation of the Needleman Wunsch algorithm using OpenMP. The experiment was carried against 4 Intel(R) Core(TM) i7 CPUs with the above configuration. It is worth mentioning that the operating system perceives 4 Intel(R) Core(TM) CPUs with hyperthreading support as 8 cores so two threads could be pinned to one core. A range of input data sizes was utilized in this experiment to show the impact of the number of sequences to be aligned on the speedup. The data line in Fig-

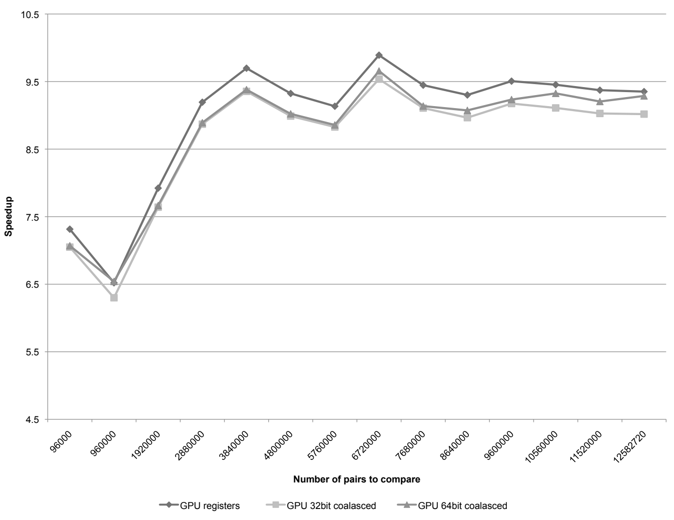

Figure 3: Speedup of our proposed algorithm implemented on a GPU compared to an optimized multi-threaded implementation on a quad-core CPU. Three versions of the GPU implementation are depicted above. At the largest input data set of over 12 million pairs, our shared memory algorithm runs 9.3 times faster.

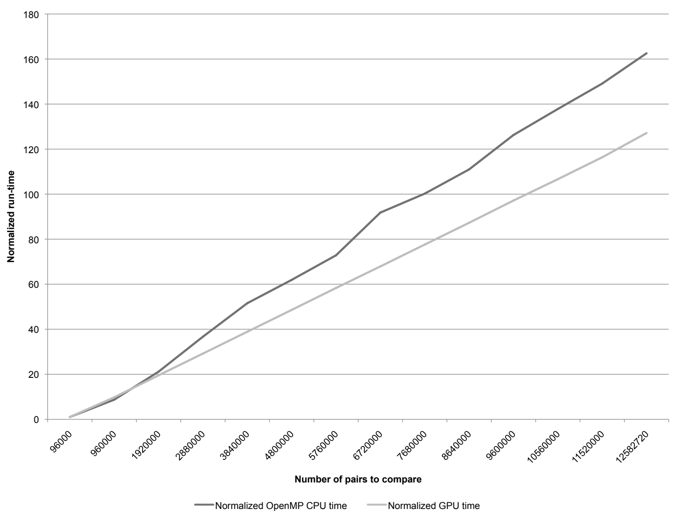

Figure 4: Normalized run-time of the multi threaded CPU and our proposed GPU algorithm. The GPU algorithm scales linearly with the input data size, while the CPU algorithm has variations from a linear run-time, which might be due to the impact of caches filled up and spilled into higher levels of memory hierarchy. The run-times are normalized since otherwise the CPU runtimes would be an order of magnitude larger than the GPU times.

ure 3 labeled as "GPU Registers" shows the speedup of this GPU algorithm compared to the baseline multi-threaded implementation running on the first testbed system (8 hyper threaded cores). The results show a speedup of up to **9.3 times** compared to the optimized (compiled with *-O3* option) multi-threaded implementation on a quad-core CPU for 12582720 input pairs. The multi threaded CPU implementation takes about 65 seconds to finish aligning the 12582720 input pairs, while our GPU implementation goes through them in only 6.95 seconds.

As can be seen from figure 3, the speedup of our GPU algorithm increases as the input size (number of pairs to be compared) increases. This can be attributed due to the fact that for smaller number of sequences, all the sequences fit in the cache of the CPU, whereas when the size of the sequences increases beyond a certain threshold, the CPU starts incurring cache misses and hence the performance decreases. The increase in the number of sequences does not affect the time taken by the GPU to process the genes which has a linear relationship with the number of sequences. This is because the GPU implementation keeps all the sequences to be aligned in the global memory and accesses them in a coalesced manner. The rest of the calculation (the Needleman Wunsch matrix calculation and the backtracing) are all done in the shared memory and the per-thread memory (registers). This calculation is independent of the number of sequences and hence does not affect the GPU performance like it does the CPU performance.

There are some variations in the amount of speedup as shown in figure 3. To investigate this further, we have presented the normalized run-times of the multi threaded CPU algorithm and our GPU algorithm in Figure 4. As we can see, the GPU algorithm scales almost linearly with the increase in the input data set, while the CPU runtime has fluctuations which result in speedup variation in figure 3. We suspect the two sudden drops in the speedup to be due to L2 and L3 caches filling up respectively.

The next experiment was carried out to see the affect of implementing the Needleman Wunsch algorithm partly in the global memory. To achieve this the algorithm described in the previous section was altered as follows:

1. Each thread computes its complete score matrix and stores the desired rows and columns in global memory. For our experiments (sequence length 32) the row and column number 0,8,16 and 24 formed the boundaries of different sub-quadrants of size 8 and hence these were the rows and columns which were saved in the global memory. Note that depending on the lengths of the two sequences and the size of the quadrant, the rows and columns which have to be stored in the global memory will differ. Our program can be easily adjusted to achieve this.

2. While the processing of the quadrant in the backtracing phase, all the rows and the columns were bought into the shared memory.

3. The required row and column were chosen and the others were overwritten. This step is similar to the one carried out in our algorithm explained earlier.

The only difference in the algorithm presented above and the algorithm we described earlier is that the global memory is used as the bulk storage for rows and columns in this case. In the earlier description the registers available per Streaming Multi-processor were used for the same purpose.

We tried the global memory accesses in the above mentioned algorithm in 2 ways:

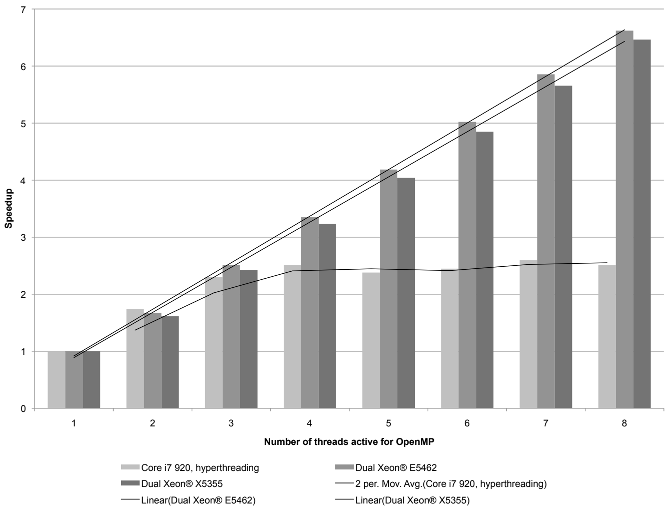

Figure 5: Strong scaling of the baseline Needleman-Wunsch implementation in three different multi-core platforms. A) hyperthreading seems to not have much impact on the speedup, B) the dual-socket testbeds scale much better. Note that if we were to include the GPU speedup in this graph, it would show a 22X speedup, since the speedups in this graph are compared to a single-core Needleman-Wunsch implementation.

1. Coalesced accesses to the global memory, 4 bytes (32 bits) at a time.

2. Coalesced accesses to the global memory, 8 bytes (64 bits) at a time.

Figure 3 shows the graph for Number of sequences vs. The time taken for the different algorithms mentioned as two other trend lines. As it can be seen from the figure, our original algorithm scales well in terms of the number of genes and maintains its speedup. The other 2 algorithms show slightly smaller speedups for small number of sequences. This is because the overhead of accessing the data from the global memory is larger for small number of sequences. Having said that, the difference is not dramatic, resulting in about 5% increase in speedup. This result reconfirms that the combination of the techniques we employed to make the algorithm compatible with SIMD architectures has enabled the system to successfully hide the memory access latencies, as we have provided the SMs with enough warps to be able to do so.

4.3 Strong Scaling

The last two experiments are performed on the baseline testbeds. In the first experiment, we ran our proposed algorithm on the CPU baseline testbed instead of a GPU, and compared its execution time to the regular Needleman-Wunsch on the multi-core. The results show that our algorithm is about **2.7 times slower** than the regular NW when it is executed on CPUs. This result clearly shows that an algorithm, which is the best candidate for SIMD architectures, might not be optimal for serial execution, and vice versa.

Figure 5 shows the results of our final experiment to determine the scaling behaviour of the baseline NW algorithm. We set this experiment to test the strong scaling feature of the baseline Needleman-Wunsch algorithm on three different multi-core test-beds. The speedups are computed by dividing the execution time of an OpenMP based multi-threaded implementation with the specified number of cores over the execution time of the NW algorithm with no OpenMP support. It should be noted that we experienced a 19% performance drop as a result of introducing OpenMP to our baseline code, meaning the OpenMP code with the number of threads set to 1 runs 19% slower.

As is shown in the figure 5, the speedup in the core i7 920 quad-core CPU is maxed out at 2.6 times which is achieved when the number of threads reaches 7. The impact of the hyperthreading support is a 3.3% increase compared to utilizing 4 threads (2.51 X), which is less than what one would expect from hyperthreading. (typically a 10 $\sim$ 15 percent performance improvement is reported in the literature). In comparison, the two dual socket machines perform much better with an almost linear scaling (shown with a linear regression line in figure 5 for the two Xeon testbeds that match the data set well), reaching in average 6.5 times speedup when all 8 cores are utilized. Please keep in mind that the GPU implementation is **22 times faster** than the single core NW implementation (results of figure 3 are speedups compared to when all 8 hyperthreading cores of the Core i7 920 machine are engaged using OpenMP). Therefore, even if the linear scaling of the multi-core algorithm remains above 8 cores (which we could not verify as we didn't have access to such a machine), we would need 24 to 32 CPU cores to achieve the same speedup. Moreover even our 8 core, dual-socket testbeds are prohibitively more expensive compared to the cost of adding the GeForce 275GTX GPU that we used to perform our experiments (which admittedly is not a top-of-the line GPU as of the date of writing this paper).

5. RELATED WORK

Our implementation of the Needleman-Wunsch algorithm can solve the problem of mapping large amount of short reads back to a reference genome quickly, which is one of the fundamental problems faced by the bio-informatics community today. There are a number of tools created for mapping reads against a reference genome. These tools map the read to a reference genome within a given edit distance. Some of these are MAQ [9], BWA [8], BOWtie [7], PASS, SHRiMP [14] and SOAP2 [10]. MAQ was one of the first tools for this task, its approach is to hash the reads in memory while traversing the genome. This results in a reduced memory footprint. It lacks support for the output of more than one matching position per read. BWA, Bowtie, and SOAP2 index the complete reference genome using a Burrows-Wheeler-Transformation (BWT) (Burrows and Wheeler, 1994) and process all the reads sequentially, resulting in a considerable speed-up of the mapping process. PASS and SHRiMP also perform an indexing of the reference sequence, but use a spaced seed approach (Califano and Rigoutsos, 2002) instead of BWT. All mentioned approaches except SHRiMP are heuristic and do not guarantee the mapping of all possible reads. Especially reads that show insertions or deletions compared to the reference sequence are often missed. The work done in [1] is closely related to our work. The authors do not mention any optimization strategies which they applied to the GPU implementation of the pair-wise alignment. Hence it is difficult to compare the speedup obtained for their implementation with ours. It is worth mentioning that the authors report that 23,040,503 probe/pattern pairs of length 36bp each took a total of 42 seconds with their implementation. The same number of pattern/probe pairs can be aligned using our implementation on the GPUs in 14 seconds.

There also have been many previous attempts to accelerate the Smith Waterman algorithm on the GPUs and other specialized hardware. As the Smith-Waterman algorithm is used for local alignments of genes these efforts and the optimizations used in these implementations cannot be applied to our implementation of the Needleman Wunsch algorithm which is used for global alignments. Most of these implementations target the search for similar proteins in DNA databases and hence the problem which they solve is fundamentally different from ours. We focus on aligning large amounts of short reads(probes) with their potential matches in the human DNA (patterns). The potential matches for the short reads are given using another algorithm, which acts as the first phase for this implementation. *The filter algorithm is not the focus of this paper.* Moreover, our GPU implementation can also be used to find the edit distances between millions of words in a few seconds. Hence the applicability is beyond the general bio-informatics application and our algorithm can be regarded as stand alone and independent. The differences in the optimizations that can be applied on the Smith Waterman algorithm when implemented on the GPUs can be gauged by CUDASW++ [11], which is one of the fastest implementations of the Smith Waterman algorithm on the GPUs.

In CUDASW++ the authors apply a number of optimizations for inter and intra task parallel implementations of the Smith-Waterman algorithm. The problem the authors are solving involves matching a gene sequence with gene sequences in a database which are within a given edit distance. This problem is different from ours and hence the optimizations used by the authors are different from the ones we can use. The Smith Waterman algorithm is used to find local alignments and hence the authors could split the sequences being matched into smaller subsequences and apply the SW algorithm on these subsequences. This improves the memory accesses and with coalesced accesses of global memory, this technique makes their implementation fast. This approach unfortunately cannot be used when finding global alignments. The authors only need to find an alignment score and hence no traceback phase is required. Due to this, they don't have to tackle with the problem of thread divergences, which is a huge problem in the traceback phase of the Needleman-Wunsch algorithm.

[2] Gives a performance comparison of the Needleman Wunsch algorithm on FPGAs, GPUs and multi cores. Their work is an analysis of three diverse applications-Gaussian Elimination, DES, and Needleman-Wunsch and is not focused on speedup, but rather on the diverse characteristics of their performance that allow speedup. In their parallel implementation of the Needleman Wunsch algorithm, they process the score matrix in parallel diagonal stripes from the top-left to the bottom right to find the maximum score path. The authors made no specific effort to tune their implementations to reduce cycle-counts and achieve speedup. No effort is also made to increase the GPU utilization and tune the application for the GPUs. The goal of their work is fundamentally different in the sense that we focus on accelerating the Needleman Wunsch algorithm using the GPUs and hence tune our implementation to a great extent to utilize the GPU well. [16] implements the NW algorithm on the GPUs using shared memory and global memory accesses. Their parallelization strategy is to calculate the score values on the minor diagonal in parallel with global memory accesses. In their shared memory implementation the authors divide the NW matrix into small blocks and apply their diagonal parallelization strategy on each of the small blocks. Due to the dependency between blocks, the authors had to explicitly synchronize the CUDA threads. Their work concentrates on aligning a pair of sequences and the GPU works on only one sequence at a time, comparatively in our work, the GPU handles all the pairs of probe and patterns at the same time. The authors achieve a speedup of 4.2X compared to their CPU implementations of the Needleman-Wunsch algorithm.

6. CONCLUSIONS

In this paper, we proposed a new algorithm to compute the edit distance between two sequences using a GPU and presented details of its design and implementation. By carefully managing the memory usage and control-flow divergence, our algorithm provides a 9.3x speedup over an efficient multi-threaded, CPU-based implementation. With growing importance of applications like genome sequence alignment, we believe that using optimized algorithms on GPUs can help in large scale data analysis. An efficient edit distance algorithm that can be used by many applications is the first step in this direction and we hope to study how end-to-end systems can be designed using this as a building block.

References

[1] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann. Exact and complete short read alignment to microbial genomes using gpu programming. *Bioinformatics*, 2011. doi: 10.1093/bioinformatics/btr151.

[2] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101 –107, june 2008. doi: 10.1109/SASP.2008.4570793.

[3] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 591–600, New York, NY, USA, 2006. ACM. ISBN 0-89871-605-5. doi: http://doi.acm.org/10.1145/1109557.1109622.

[4] J. Droppo and A. Acero. Context dependent phonetic string edit distance for automatic speech recognition. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 4358 – 4361, march 2010. doi: 10.1109/ICASSP.2010.5495652.

[5] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360825.360861.

[6] J.-M. Koo and S.-B. Cho. Effective intrusion type identification with edit distance for hmm-based anomaly detection system. In S. Pal, S. Bandyopadhyay, and S. Biswas, editors, *Pattern Recognition and Machine Intelligence*, volume 3776 of *Lecture Notes in Computer Science*, pages 222–228. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-30506-4.

[7] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10 (3):R25, 2009.

[8] H. Li and R. Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[9] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851, 2008.

[10] R. Li, C. Yu, Y. Li, T. Lam, S. Yiu, K. Kristiansen, and J. Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966, 2009.

[11] Y. Liu, D. Maskell, and B. Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009. ISSN 1756-0500. doi: 10.1186/1756-0500-2-73. URL `http://www.biomedcentral.com/1756-0500/2/73`.

[12] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[13] A. Pollack. Dna sequencing caught in deluge of data. *The New York Times*, 2011.

[14] S. Rumble, P. Lacroute, A. Dalca, M. Fiume, A. Sidow, and M. Brudno. Shrimp: accurate mapping of short color-space reads. *PLoS computational biology*, 5(5): e1000386, 2009.

[15] S. Simsek. An edit-distance algorithm to detect correlated attacks in distributed systems. *International Journal of Computer and Information Engineering*, 2, 2008.

[16] T. Siriwardena and D. Ranasinghe. Accelerating global sequence alignment using cuda compatible multi-core gpu. In *Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on*, pages 201 –206, dec. 2010. doi: 10.1109/ICIAFS.2010.5715660.

[17] I. Yalniz and R. Manmatha. A fast alignment scheme for automatic ocr evaluation of books. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 754 –758, sept. 2011. doi: 10.1109/ICDAR.2011.157.