Eurographics Symposium on Parallel Graphics and Visualization (2008)
J. Favre, K. - L. Ma, and D. Weiskopf (Editors)

Parallel Longest Common Subsequence using Graphics Hardware

J. Kloetzli [1] B. Strege[1] J. Decker[2] M. Olano[1]

[1]UMBC ({jk3,bstreg1,olano}@umbc.edu)
[2]US Naval Research Lab (decker@ait.nrl.navy.mil)

Abstract

We present an algorithm for solving the Longest Common Subsequence problem using graphics hardware acceleration. We identify a parallel memory access pattern which enables us to run efficiently on multiple layers of parallel hardware by matching each layer to the best sub-algorithm, which is determined using a mix of theoretical and experimental data including knowledge of the specific hardware and memory structure of each layer. We implement a linear-space, cache-coherent algorithm on the CPU, using a two-level algorithm on the GPU to compute subproblems quickly. The combination of all three running on a CPU/GPU pair is a fast, flexible and scalable solution to the Longest Common Subsequence problem. Our design method is applicable to other algorithms in the Gaussian Elimination Paradigm, and can be generalized to more levels of parallel computation such as GPU clusters.

1. Introduction

Dynamic programming (DP) algorithms solve a vast set of optimization problems in computer science. This class of algorithms is based upon the principle of the time/space trade-off, and typically leans towards using more space in order to reduce the asymptotic complexity of an algorithm. The fundamental observation about DP algorithms is that they recursively break the problem up into overlapping subproblems, storing the answer to the subproblems for later reference. If the subproblems overlap enough, then the time complexity can be reduced drastically, typically from exponential to polynomial. Some of the most efficient of these algorithms for single processors belong to the cache-oblivious model [CR06], in which algorithms do not need knowledge of cache and memory sizes to perform efficiently.

Computer graphics hardware, specifically the consumer available Graphics Processing Unit (GPU), is a massively parallel processor used in conjunction with the CPU which provides substantial computing power. In addition, the cost for GPU processors is significantly lower than comparable CPU clusters. These advantages have lead to the popularity of General-Purpose GPU (GPGPU) applications, or those which apply the power of the GPU to solve non-graphics problems. NVIDIA has recently released a software development kit called CUDA specifically to support GPGPU applications [NVI07]. Research in this area consists of devising parallel algorithms specifically for GPU architecture [OLG*05] in an attempt to minimize the constant factors in running time.

Although specific DP algorithms have been implemented in graphics hardware, little or no work has been done to describe a general mapping of DP algorithms (or a significant subset therof) onto GPU-style architectures. Our concurrent goals in this paper are to describe our Longest Common Subsequence algorithm as an efficient GPGPU solution to a specific problem on NVIDIA hardware, and also to present an algorithmic design pattern for creating GPGPU solutions for a class of DP algorithms called the Gaussian Elimination Paradigm on any graphics hardware.

2. Related Work

Dynamic programming is a powerful theoretical technique for solving a certain class of optimization problems in polynomial time. However, if greater practical efficiency is desired, acceleration can be achieved by creating parallel DP solutions. While such solutions do not improve asymptotic performance, reductions in algorithmic constant factors can lead to nontrivial performance improvements. Galil and Park designed parallel versions of four algorithms for a CREW (Concurrent Read Exclusive Write) PRAM (Parallel Random Access Machine) model of parallel computation [GP91], reporting optimal sublinear-time for three of the four resulting algorithms. Canto et al. implemented parallel dynamic programming algorithms on a network of workstations enabled with message passing capability [CdMB05]. They reported that their system was able to scale to 12-15 processors, but noted that unavoidable load balancing issues hindered system performance.

There has also been work in developing GPU-accelerated versions of specific dynamic programming algorithms. Liu and Schmidt [LSVMW07] compare a large number of very short sequences efficiently on the GPU by computing multiple sequences in parallel. Because they have multiple unrelated tasks to perform, they are able to achieve very high utilization of the GPU processors and report speedup of over an order of magnitude. Since they are only interested in finding similar sequences, they only reconstruct the actual matching subsequences for a small number of the comparisons. Schatz et al. [STDV07] presented a GPU accelerated sequence alignment software package called MUMmerGPU. They pre-process a single long sequence millions of base pairs (bp) long into a suffix-tree data structure which allows them to process many small sequence (800 bp or less) alignments in parallel. They achieve a 3-4x speedup for their entire application runtime.

Another important research direction for dynamic programming is the development of linear-space DP algorithms. Solving for the LCS of large sequences, such as genetic data which can consist of millions of characters, is not practical given the $O(n^2)$ memory requirement of the naïve DP algorithm. Hirschberg [Hir75] proposed the first linear space dynamic programming algorithm for LCS, making genomic-length comparisons possible. However, Hirschberg's linear method was susceptible to cache thrashing, resulting in sub-optimal performance. Chowdhury and Ramachandran presented a cache-oblivious framework for algorithms within a subset of DP algorithms they termed the Gaussian Elimination Paradigm (GEP). This paradigm describes all algorithms with similar construction to the method for solving Gaussian Elimination without pivoting [CR06]. The GEP includes dynamic programming algorithms such as Edit Distance, LCS and Matrix Multiplication. By efficiently using memory cache, Chowdhury and Ramachandran were able to achieve faster performance on LCS than Hirschberg.

In addition, Chowdhury et al. have presented a parallel implementation of their cache-oblivious GEP framework. They define the maximum number of processors which can be used efficiently in parallel as a function of the input size. Unfortunately, this function scales poorly with sequence length, and thus the maximum amount of parallelization is fairly low, even for large sequences. Given this, it would not be reasonable to implement their method on the GPU without modification, as the massively parallel GPU architecture would not be used effectively. Instead, we implement a highly parallelizable algorithm for solving subproblems of their framework. This enables us to retain their high performance on the CPU while fully exploiting the highly parallel GPU hardware.

The GPU architecture has unique benefits and challenges. First, the GPU was designed to fit a very specific parallel application (rasterized rendering of computer graphics images) and so does not fit exactly into one of the common parallel architecture categories. Recently, GPU manufacturers have released programming APIs which expose the underlying processors of the GPU without having to work through a graphics-oriented interface. Our work is based on one of these frameworks, called the Compute Unified Device Architecture (CUDA), from NVIDIA [NVI07]. We use the documentation released with the CUDA API heavily to determine how to efficiently program on their hardware.

3. Background

In the following section, we provide some explanation about the concepts and problems behind our work.

3.1. Longest Common Subsequence

Longest Common Subsequence is a problem that has applications in a number of fields. Given two sequences X and Y of lengths n and m respectively, the solution is the longest ordered series of elements that X and Y have in common. Note that this definition does not require that elements be contiguous in either sequence, simply that they be in order. The recursive definition of LCS is:

$$\mathrm{LCS}(X_{1\dots i}, Y_{1\cdots j}) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \mathrm{LCS}(X_{1\dots i-1}, Y_{1\cdots j-1}) + 1 & \text{if } x_i = y_i \\ \mathrm{MAX}\begin{pmatrix} \mathrm{LCS}(X_{1\dots i-1}, Y_{1\cdots j}), \\ \mathrm{LCS}(X_{1\dots i}, Y_{1\cdots j-1}) \end{pmatrix} & \text{otherwise} \end{cases} \tag{1}$$

LCS is sometimes used in biology to determine similarity between genetic sequences. Our main focus is to use LCS as a representative problem from the Gaussian Elimination Paradigm in order to have a concrete problem to study. In theory, many problems from the GEP could be accelerated by our method, although the amount of acceleration may vary. We choose to solve this particular problem because it is simple to implement, easy to understand, and has practical applications.

The Cache-Oblivious Model of computation is similar to the Turing model, except it introduces a finite-size cache tape between the processor and the turing tape. The function of this cache is similar to a cache in the RAM model which most computers follow, acting as a buffer between the turing tape and the machine. Cache-Oblivious algorithms are designed to work efficiently without knowledge of the cache tape length, removing the need to tweak parameters based upon the specific architecture on which code is being executed. Although the Cache-Oblivious model assumes fully-associative cache, which is generally not implemented in actual hardware, and perfect cache replacement policy, which is impossible, Cache-Oblivious algorithms are still efficient on real processors.

3.2. The NVIDIA G80 Architecture

NVIDIA recently released a new parallel graphics hardware architecture called the G80, along with a framework for General Purpose GPU (GPGPU) programming called the Compute Unified Device Architecture (CUDA). The G80 we used has 16 processing units, which each have 8 cores clocked at 1.3GHz, and has 768MB of onboard DDR3 RAM. In order to meet the memory requirement for each processor, GPU hardware has several memory spaces with different attributes. Our fundamental design strategy involves determining algorithms to run efficiently in each of these memory spaces, so understanding their performance characteristics is crucial to our design.

The *Global* memory space is read-write, allocated from the onboard RAM, and accessible from any processor at any time. In order to avoid conflicts global memory is not cached, making it the slowest type of memory to access. *Constant* memory is read-only, resides in a special memory block which can be read by any processor at any time, and is cached in each processor independently. This memory must be set by the CPU before executing GPU code, and has very limited size (around 64kb). The final type, called *Shared* memory, is very fast read-write memory located on each multiprocessor chip and only accessible by the cores of that particular multiprocessor. The purpose of shared memory is to hold frequently used data close to the multiprocessor for as long as possible before writing back out to the global memory.

4. Algorithm Description

The method which we developed accelerates the linear space method by Chowdhury et al. by solving fixed-sized subproblems very quickly on the GPU. One advantage this approach has over parallelizing their algorithm directly is that we can use any algorithm which maps well onto the GPU instead of being limited to a specific algorithm. As long as the maximum size subproblems solved is a constant we still achieve linear space with potentially significant acceleration. This gives us the freedom to find a design which maximizes the potential of graphics hardware architectures.

4.1. Basic Structure

One possible approach to solving the LCS recurrence from Equation 1 is to view a *logical matrix* for all the solutions of the recurrence, where each element of the matrix is referenced by the values i and j on the left hand side of the equation. The solution to the problem then requires two steps: a *memoization* phase which loops over every element of the matrix to fill in the solutions, and a *reconstruction* phase which uses the values in the matrix to trace out the longest subsequence. The memoization stage gives the length of the LCS, but reconstruction must be performed in order to obtain the actual subsequence values. Formulated in this way the entire logical matrix would have to be stored in memory at the same time, which requires $O(n^2)$ space. This method is very fast for small sequences, and more advanced algorithms (including the Chowdhury algorithm) use it as a base case for small subproblems. The pseudocode for solving the LCS problem in this way is given in the SIMPLE-LCS pseudocode as follows:

```
SIMPLE-LCS (subsequences $a_{ax...ay}$, $b_{bx...by}$, input boundary $B$)
1  Store input boundary in physical matrix $m$
2  $r \leftarrow ay - ax$
3  for $i \leftarrow 1$ to $r$
4      do for $j \leftarrow 1$ to $r$
5             do execute Equation 1 on $m[i][j]$
6                return output boundary of physical matrix $m$
```

The Chowdhury et al. algorithm, which is represented by the pseudocode CH-LCS , is the current fastest sequential linear-space LCS algorithm. The main idea is to divide the logical matrix into quadrants in a specific way to allow determination of the longest subsequence path while only retaining a linear number of cells in memory at a given time.

```
CH-LCS (subsequences $a_{ax...ay}$, $b_{bx...by}$, input boundary $B$)
1  $r \leftarrow ay - ax$
2  if $r \leq CUTOFF$
3     then run SIMPLE-LCS on input subsequences
4     else CH-LCS on input subsequences in TOP LEFT
5          CH-LCS on input subsequences in TOP RIGHT
6          CH-LCS on input subsequences in BOTTOM LEFT
7          CH-LCS on input subsequences in BOTTOM RIGHT
8          return output boundary from quadrant output sections
```

Note the calls to SIMPLE-LCS to solve subproblems which fall below a certain cutoff size, which they determined experimentally to be between 2^8 and 2^{10} depending on the specific architecture used. Our accelerated version of their algorithm is shown in GPU-LCS as:

```
GPU-LCS (subsequences $a_{ax...ay}$, $b_{bx...by}$, input boundary $B$)
1  $r \leftarrow ay - ax$
```

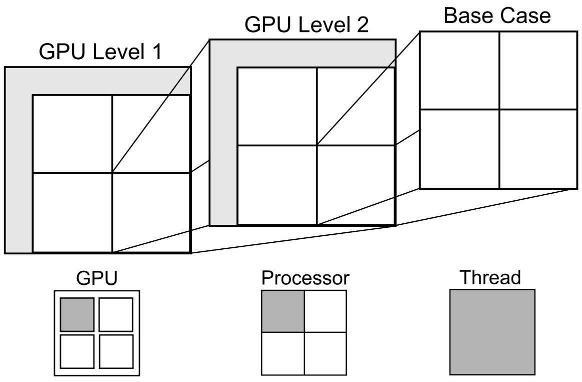

Figure 1: *The three levels of parallelism present on the GPU, and levels of the logical LCS matrix mapped to each one. GPU Level 1 parallelism is stored in the global memory of the GPU according to a quadratic-space matrix folding technique. GPU Level 2 parallelism is stored in the shared memory of one multiprocessor on the GPU, and is computed with a linear space method. Each base case is computed by one of the threads (cores) of the multiprocessor.*

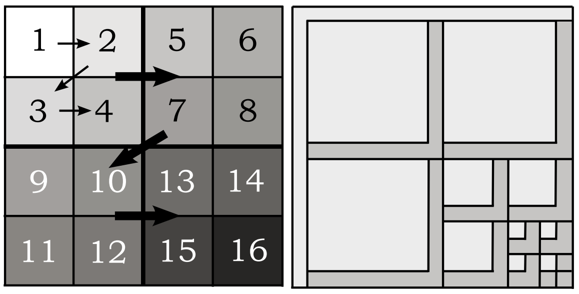

Figure 2: *On the left, order of element computation in a 4x4 logical submatrix for the output boundary problem. At every level (i.e. 2x2, 1x1), first the upper left quadrant is fully computed, then the upper right, lower left, and lower right. On the right, an image of the required storage to find the LCS for a 16x16 logical matrix. The input boundary (top and left edges) has a space complexity of* $\Theta(2n+1)$. *The dark shaded regions represent the required intermediate output boundaries that must be stored, with space complexity of* $\Theta(6n-6-3\lg(n))$. *Total storage required for the computation of an LCS is* $\Theta(8n-5-3\lg(n))$, *which is linear.*

```
 2  if $r \leq CUTOFF$
 3    then run SIMPLE-LCS on input subsequences
 4    else  if $r \leq GPUCUTOFF$
 5            then GPU-LEVEL-ONE on input subsequences
 6            else  GPU-LCS if in TOP LEFT
 7                  GPU-LCS if in TOP RIGHT
 8                  GPU-LCS if in BOTTOM LEFT
 9                  GPU-LCS if in BOTTOM RIGHT
10                  return output boundary from quadrant output
```

The difference between our algorithm and theirs at this level of abstraction is that we have introduced a second way to solve subproblems below a certain size very quickly on the GPU. The maximum size problem which we can currently solve this way is 2^{16}, which is large enough to let us offload significant portions of the algorithm. If the algorithm needs to solve a very small problem, however, it is still more efficient to use the CPU with the original SIMPLE-LCS algorithm.

In the following subsections we discuss our CPU-optimized linear space algorithm, followed by the GPU algorithms and how they map onto NVIDIA hardware to maximize performance.

4.2. CH: CPU Linear Space DP

Our linear space implementation for calculating the LCS of a pair of sequences on the CPU was based on the pseudocode in [CR06] describing their Cache-Oblivious LCS algorithm. They use 4-way partitioning to break what is called the "output boundary" problem down into smaller and smaller pieces until the base case size or smaller, and memoizes enough data along the way to reconstruct the sequence once the output boundaries have been computed.

4.2.1. Output Boundary

Computing the output boundary for particular subsequences is essentially the computation of the leading edges of the logical LCS matrix. Since the storage of intermediate LCS matrix data is not needed for our reconstruction algorithm, the output boundary can be computed instead from the entire submatrix. Our algorithm for computing the output boundary requires only linear space, and follows from the method described in [CR06] as it uses 4-way partitioning. Computing the output boundary for certain subsequences requires the input boundary – the trailing edges – of the logical LCS matrix. As can be seen in the first image of Figure 3, the input boundary can easily be stored in linear space. From this point, a sequence of "pushing" the values of the input boundary in the order shown on the left side of Figure 2 leads us to the output boundary, and this never increases the size of the physical array used to store these boundaries. At each point that must be pushed, the generic algorithm for computing values in the logical LCS matrix is applied. Figure 3 shows a few snapshots that an input boundary will take before becoming an output boundary with our algorithm.

4.2.2. Reconstruction

The output boundaries initially required to reconstruct the LCS are shown in Figure 2. Once these output boundaries have been calculated and memoized, the reconstruction of the LCS can begin. This works by retracing the optimal path

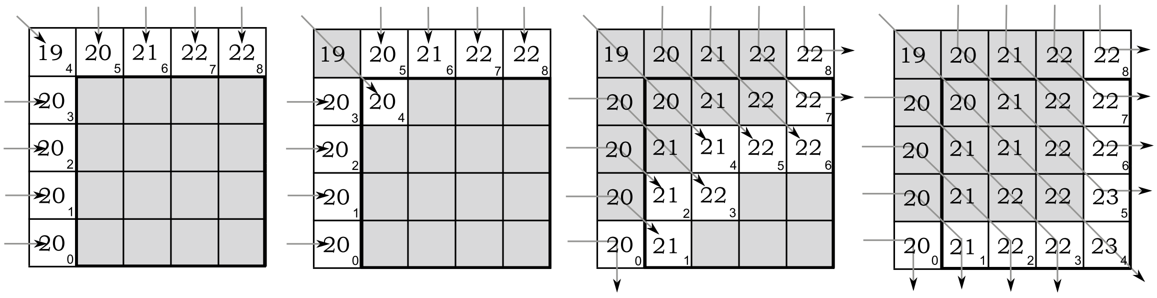

Figure 3: *Montage of four stages in computing the output boundary of a 4x4 logical submatrix with input boundary shown. The lightened blocks show what is currently stored in physical memory, with the subscripts corresponding to their respective array positions. The first image shows the initial state when the output boundary computation function is called. The second image shows the first "push," where the new value is calculated based upon the standard LCS formula for the corresponding subsequences (not shown). The third image shows an intermediate state of the output boundary computation. The fourth and final image shows the result of computation of the output boundary, where the output boundary is comprised of the highlighted blocks within the original 4x4 matrix space.*

back through the logical LCS matrix, and since we do not have all of the values in the matrix recorded – which would require quadratic space – the intermediate values must be recomputed as the optimal path is found. Much like the computation of the output boundary, our method uses 4-way partitioning to accomplish this retracting. We follow the algorithm as described in [CR06] where the reconstruction works by computing a small section of the optimal path, then finding which adjacent quadrant this path intersects and moving on to that quadrant. Following this method, quadrants that do not contain a piece of the optimal path are ignored while retracing.

The amount of space required for the initial output boundaries has been computed to be $\Theta(6n-6-3\lg(n))$. Combined with $\Theta(2n+1)$ space which is required for the initial input boundary, this algorithm initially requires $\Theta(8n-5-3\lg(n))$ space. As the retracing moves back through the logical LCS matrix, we will never need to increase this amount of space since we can store the newly computed output boundaries in the same memory locations as the ones that we can now discard. Therefore this algorithm also requires only linear space, and since all of the space ever needed by the output boundary problem is given to it by this algorithm, our entire process of computing the LCS for two sequences requires only linear space.

5. GPU Algorithm Description

Once the 4.2 algorithm has broken the logical matrix into subproblems below a constant threshold our GPU optimized method is used to determine the output boundary very quickly. The GPU is essentially a hierarchical processor, since it contains processors which consist of cores. Thus, we again break the subproblem down further, creating an algorithm optimized for each specific layer of hardware. This section provides an overview of the parallel memory access pattern our method is based upon, followed by a discussion of the specific optimizations performed.

5.1. Parallel Memory Access in LCS

One of the main bottlenecks of any parallel algorithm is data access. In order to perform parallel LCS computations efficiently, we have identified a specific parallel data access method for the logical matrix which the GPU relies upon to avoid memory contention. Consider a $n \times n$ block inside the logical matrix divided up into $m \times m$ pieces, where n is divisible by m, forming a grid of subproblems. In order to determine how the subproblems can be solved in parallel, it is important to notice that each block (x,y) depends on $(x-1,y)$, $(x,y-1)$, and $(x-1,y-1)$. This leads us to the observation that we can compute all blocks on one *diagonal* of the grid in parallel, assuming that all earlier diagonals have been computed. The bottom right of Figure 4 shows the subproblem matrix with each grid shaded according to the diagonal it belongs to. One interesting feature of this memory pattern is that it contains "ramp up" and "ramp down" periods at either end, which are determined by diagonals which have fewer blocks than available processors. In order to maintain efficiency, these areas of under-utilization must be small when compared to the entire job, which puts an effective minimum size subproblem which is efficient. We determine these limits experimentally, as described in later sections.

5.2. GPU Level 1: Quadratic Space DP

The first level of parallelism we use is a quadratic-space DP algorithm which is run on the entire GPU. This level is called by CH-LCS as soon as the problem is large enough to be efficiently solved on the GPU but small enough to fit onto graphics hardware, which has been empirically determined

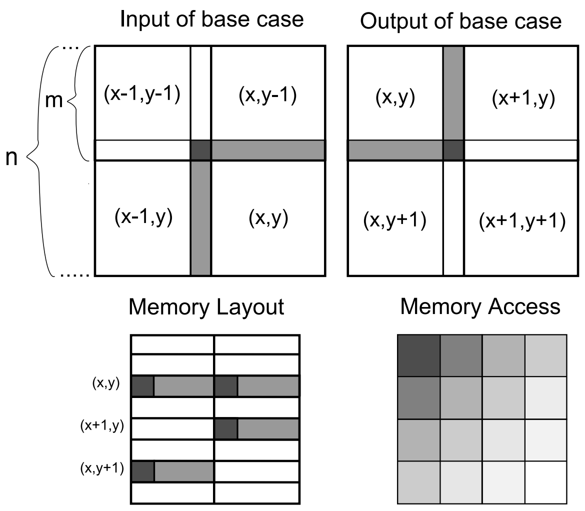

Figure 4: *Top left: Block (x,y) requires the three shaded areas as input boundaries. Top right: The same block has two output boundaries. Bottom left: Each block only needs to store its output boundary, which we fold to minimize memory usage and memory contention. Bottom right: Depiction of the parallel memory access pattern. All blocks on the same diagonal (depicted as the same shade) can be computed in parallel, given that all previous diagonals (darker shades) have been computed.*

to be sequences between 2056 and 32,256 for our hardware. Since this algorithm is a base case for the recursive *output boundary* function of CH-LCS described above, it only needs to compute the output boundary of a submatrix within the logical matrix given the input boundary to that submatrix, and not the actual LCS solution. In addition, since CH-LCS only needs to solve square subproblems with sequence lengths which are a power of two, our implementation only works with sequences which satisfy these two requirements, allowing us to perform some small optimizations.

GPU Level 1 subproblems are solved in parallel by looking at the $n \times n$ logical matrix of the sequence and dividing it up into m blocks in each dimension, which are solved in parallel according to the memory access pattern from the previous section. We loop through each diagonal, concurrently launching one multiprocessor GPU job for each block in the current diagonal. The values for each grid block are filled in by the GPU Level 2 algorithm described below, which is referred to as thread-level parallelism. The pseudocode for this algorithm can be seen in GPU-LEVEL-ONE .

Although the GPU Level 1 method is quadratic space, it employs a memory addressing scheme pictured in the right of Figure 4 which reduces the absolute amount of space from $\Theta(n^2)$ to $\Theta(2(n^2)/m)$, where m is the size of each block. Instead of using an $n \times n$ matrix to store the results, we only store the output boundary for each grid cell. In order to compute the subproblem within each grid, we need to have access to the output boundaries of the blocks which it depends on. Since the optimal value for m was determined experimentally to be 512, this memory folding scheme significantly reduces the required memory and allows us to solve larger subproblems. Note that this method, pictured on the bottom left of Figure 4, stores two copies of the corner output element, allowing both requesting blocks in the next diagonal to be served in parallel. Given that the global memory in the GPU is not cached, performance could suffer if memory contention existed.

```
GPU-LEVEL-ONE (sequences $a$, $b$, input boundary $B$)
1  for $i \leftarrow 1$ to $2n-1$
2     do for $j \leftarrow 1$ to diagonal $i$ size, in parallel
3           do GPU-LEVEL-TWO with block j on diagonal i
4              Store the resulting output boundary
```

5.3. Level 2: GPU Linear Space DP

The next level is the linear space DP algorithm which we use to solve each grid from GPU Level 1. It uses the shared memory space in the GPU and computes the answer for the grid block by breaking the submatrix into blocks again. Each one consists of a submatrix 4×4 in size and is solved by one core of one of the multiprocessor in a sequential manner. Since the shared memory space is almost as fast as registers, we use a linear space algorithm which allows the largest subproblem possible to fit. Memory contention is not a big issue at this scale because reads and writes are very fast.

In order to maintain the correct solution, we have to enforce the same order property described in GPU Level 1. While the parallel data access pattern applies to this scale of computation, we cannot enforce the order with the CPU at this level. Instead, we use the built-in hardware synchronization of a multiprocessor; a hardware lock which only releases when all cores have requested the lock. We launch enough threads to compute each block of the largest diagonal in parallel, which is $2k+1$ where $k = n/m$. One way of thinking of this is to assign one thread to each slot of physical memory and have it compute the blocks which happen to fall into that slot of memory in the order enforced by the thread synchronization. This is visualized in Figure 3, where each arrow shows the computation of one core. The algorithm precedes much in the same way as in CH-LCS except that the pushing order will be determined by diagonal blocks instead of quadrants. The pseudocode for this algorithm is given by GPU-LEVEL-TWO below.

```
GPU-LEVEL-TWO (sequences $a$, $b$, input boundary $B$)
1  for $i \leftarrow 1$ to $2n-1$
2     do for $j \leftarrow 1$ to diagonal $i$ size, in parallel
3           do GPU-LEVEL-THREE with block j on diagonal i
4              Store the resulting output boundary
```

5.4. Level 3: GPU Serial Linear Space DP

Each of the blocks in GPU Level 2 is assigned to a single thread to compute in a serial manner. Since we are trying to maximize the size problem which will fit onto each core, the shared memory space containing the input boundary for each block is the only storage space used. This is accomplished by following the same linear space algorithm as GPU Level 2 without the parallel computation.

5.5. Analysis

This section provides basic analysis of our algorithms. We believe that showing that both GPU algorithms are work-efficient when compared to SIMPLE-LCS is enough to demonstrate that they are reasonable, although a more complete analysis, which we leave for future work, would also analyze the number of processors which can be effectively used based upon input size. The pseudocode for GPU-LEVEL-ONE contains two loops, one for each diagonal in the grid matrix and one for each element in the diagonal. Given that sets of blocks which appear in the diagonal do not intersect, this loop is going to call each diagonal in the block exactly once. This is exactly what the SIMPLE-LCS algorithm does, with the added constant factor overhead of diagonal indexing instead of column/row indexing. Therefore, GPU-LEVEL-ONE is work-efficient when compared to SIMPLE-LCS . The proof for GPU-LEVEL-TWO is exactly the same.

The parallel running time for both GPU-LEVEL-ONE and GPU-LEVEL-TWO is $O(n)$ given $2k-1$ processors at each stage, where $k = n/m$. This can be seen by the inner loop in the pseudocode, which collapses to a constant amount of time given k processors. In this case, only the outer loops will be executed in serial, and, since there are a linear $(2k-1)$ number of diagonals, this will be linear time.

6. Results and Analysis

This section presents the results we obtained from our implementation, along with the parameters we found to be optimal. Block size for GPU Level 2 was fastest at 4, while the optimal grid width for GPU Level 1 is 512, and the cutoff for calling SIMPLE-LCS and GPU-LCS is between 1024 and 2048. These numbers were determined experimentally after running test applications with all possible combinations, and should hold for most G80-based systems.

The performance of our new algorithm can be seen in Figure 5, along with the performance of the Hirschberg [Hir75] and Chowdhury et al. methods, as experimentally determined by Chowdhury et al. [CR06]. On average, we find our method to provide a factor of 5 to 6 times the performance of the CH-LCS algorithm alone. We ran our experiments running sequentially on an AMD Athlon 64 with a single G80 GTX GPU.

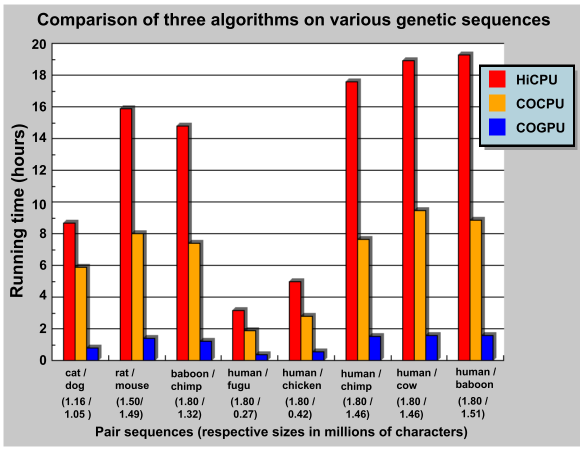

Figure 5: *Comparison of the running time of our approach (COGPU) against the linear space LCS algorithms proposed by Hirschberg (HiCPU) [Hir75] and Chowdhury et al. (COCPU) [CR06].*

6.1. Comparison to CPU parallelization

It should be noted that, even though we are comparing our performance with the performance of the linear CH-LCS algorithm, our algorithm is compatible with the parallel version of CH-LCS . In fact, it is trivial to integrate the two algorithms because GPU-LCS can be inserted into the recurrence along with SIMPLE-LCS as a separate base case without requiring any further work. Their parallel algorithm divides the work for the main CH-LCS algorithm between n processors, each of which is a standard desktop or server CPU. In order to give a theoretical bound on the number of processors which can be effectively used, they determine the following function:

$$p(n) = \frac{n^{\log_2\left(\frac{4}{3}\right)}}{2\,(4+log_2 n)} \tag{2}$$

This equation gives an exact upper bound on the number of processors which can be effectively used with their method as a function of input size. Unfortunately, $p(n)$ scales very poorly. Since $p(2^{21}) \approx 8.41$, we can conclude that sequences which are over two million elements long (which is slightly longer than any of the results presented above) will only be able to use eight processors effectively. They present results for a range of algorithms which are used in bio-informatics, achieving a maximum of a factor of six speedup with their method running on eight server processors. Given the definition of $p(n)$ above, we conclude that on the sequence sizes tested in this paper, they have achieved the maximum performance possible with their technique.

Unfortunately, data for the exact speedup for their method applied to the LCS problem is not available, so we must assume a speedup similar to the factor of six which was achieved on the algorithms which were implemented. Given this assumption, coupled with the fact that they have achieved the theoretical maximum parallelization with an eight-processor system, we conclude that our factor of five speedup using a CPU + GPU architecture has a lower cost/performance ratio. The graphics hardware which we use costs less than $500 on the current market and require no special support, while the eight processor cores required to give comparable performance cost at least as much and require special motherboard support. Finally, although we did not perform any formal analysis, it is highly likely to have an analogous $p(n)$ function which scales much better than the one for their method. This can be attributed to our focus on solving fixed-size subproblems in the most efficient way possible, instead of creating a parallel algorithm which attempts to maintain linear space usage at all levels. In this way our method is able to drastically increase performance by taking advantage of the massive parallelism on the GPU.

7. Conclusion

We have developed a framework for solving dynamic programming problems on contemporary graphics hardware. Graphics hardware has several levels of parallelism with differing characteristics. We focus on using each level of parallelization in the graphics hardware efficiently by solving subproblems of the dynamic programming recurrence differently depending on the scale of the problem and characteristics of that level of the graphics architecture. We use theoretical analysis to maintain the asymptotic time and space complexity of previous algorithms while designing each part of our algorithm to have the best performance at each level of graphics hardware. We also use empirical data to fine-tune boundaries between the different levels of subproblem computation. We believe this is a powerful design strategy for creating new and efficient GPU and GPU-cluster based algorithms.

Utilizing multiple levels of parallelism available on the GPU, we report over five-fold speedup of our CPU + GPU Longest Common Subsequence algorithm over the cache-oblivious single processor method presented by Chowdhury and Ramachandran [CR06]. We maintain reasonable memory usage and a high degree of scalability to future hardware based upon the NVIDIA CUDA architecture.

8. Future Work

We have only implemented LCS within our parallel framework for the results in this paper. It should be possible to extend this to other DP algorithms in the GEP, and to more levels of parallelization. We hope that our hybrid dynamic programming algorithm design technique based upon hardware design, theoretical analysis, and empirical data will be further investigated. In particular, we would like to see our algorithm extended by adding another level of parallelism in GPU cluster applications.

It would also be worthwhile to explore combining our method of parallelism with the method described by Chowdhury et al. to achieve even better performance. It is possible with current hardware to run such an algorithm on multi-CPU and multi-GPU architecture, which should provide substantially increased performance over our single CPU and single GPU approach.

9. Acknowledgements

We'd like to thank Rezaul Alam Chowdhury for his support and Dr. Mike Brudno for providing us with the CFTR DNA sequence data.

References

[CdMB05] CANTO S. D., DE MADRID A. P., BENCOMO S. D.: Parallel dynamic programming on clusters of workstations. *IEEE Trans. Parallel Distrib. Syst. 16*, 9 (2005), 785–798.

[CR06] CHOWDHURY R. A., RAMACHANDRAN V.: Cache-oblivious dynamic programming. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm* (New York, NY, USA, 2006), ACM Press, pp. 591–600.

[GP91] GALIL Z., PARK K.: *Parallel Dynamic Programming*. Tech. Rep. CUCS-040-91, Columbia University, 1991.

[Hir75] HIRSCHBERG D. S.: A linear space algorithm for computing maximal common subsequences. *Commun. ACM 18*, 6 (1975), 341–343.

[LSVMW07] LIU W., SCHMIDT B., VOSS G., MÜLLER-WITTIG W.: Streaming algorithms for biological sequence alignment on gpus. *IEEE Transactions on Parallel and Distributed System 18*, 9 (2007), 1270–1281.

[NVI07] NVIDIA CORPORATION: *NVIDIA CUDA Compute Unified Device Architecture : Programming Guide*. Whitepaper, NVIDIA Corporation, 2007.

[OLG*05] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports* (Aug. 2005), pp. 21–51.

[STDV07] SCHATZ M., TRAPNELL C., DELCHER A., VARSHNEY A.: High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics 8*, 1 (2007).