Evaluating the Use of GPUs for Life Science Applications[*]

Technical Report # 2008-10

Vidyananth Balu, John Paul Walters, Suryaprakash Kompalli, and Vipin Chaudhary
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
Buffalo, NY 14260
{vbalu2, waltersj, kompalli, vipin}@buffalo.edu

May 22, 2008
Technical Report # 2008-10

Abstract

As the peak performance of GPUs continues to outpace general-purpose processors, the use of GPUs for application acceleration is becoming more widespread. However, not all algorithms benefit equally from their use. In this paper we present the results of parallelizing two life sciences applications, Markov Random Fields-based (MRF) liver segmentation and HMMER's Viterbi algorithm. We relate our experiences in porting both applications to the GPU as well as the techniques and optimizations that are most beneficial. The unique characteristics of both algorithms are demonstrated by implementations on an NVIDIA 8800 GTX Ultra using the CUDA programming environment. The features of the MRF algorithm and the reasons for its exceptional speedups are discussed in detail. We also characterize HMMER's Viterbi algorithm and note the features that facilitate speedup.

1 Introduction

In recent years graphics processing units (GPUs) have become increasingly attractive for general purpose parallel computation. Parallel code that traditionally required expensive computational clusters to achieve reasonable speedup may now port to GPUs yielding results equivalent to tens of traditional CPUs at a fraction of the cost. As tools such as NVIDIA's CUDA [15] continue to mature, the burden of GPU programming continues to decrease allowing for expression of traditional parallel codes in the familiar "C" language.

With peak computing power of 345-518 GFLOPS for the latest NVIDIA GPUs compared to only 32 GFLOPS for quad-core general purpose processors, the attraction of GPUs for general

[*]This research was supported in part by NSF IGERT grant 9987598, MEDC/Michigan Life Science Corridor, and NYSTAR.

purpose computation is clear. However, graphics processors are not suited for all types of computations. In this paper we evaluate the use of the NVIDIA 8800 GTX Ultra GPU for two classes of statistical problems: the HMMER sequence database search application (*hmmsearch*, and a medical imaging liver segmentation application based on Markov Random Fields (*MRF*). We demonstrate a variety of optimization strategies that are useful for different classes of GPU-based applications. We make the following contributions:

- Implement liver segmentation using Markov Random Fields on the GPU.
- Implement a GPU-based implementation of HMMER's *hmmsearch* tool.
- Discuss and analyze the advantages and limitations of GPU hardware for general purpose HPC.

The remainder of this paper is organized as follows: In Section 2 we introduce the liver segmentation algorithm using *MRF*, while in Section 3 we briefly describe HMMER and the parallelization strategy commonly used. We present an overview of GPU computing in Section 4. In Section 5 we present the results of our *MRF* and *hmmsearch* implementations, and in Section 6 we analyze the differences of each algorithm and their suitability for GPU acceleration. Future work is presented in Section 6.1.

2 *MRF* Segmentation

Algorithm 1 *MRF*algorithm (class label 3d-array (CL), data 3d-array (D), mean (μ), variance (σ))

```
1:  Initialize number of iteration ($K$) to zero.
2:  Initialize Current Index ($I_{curr}$) to start of volume.
3:  while $I_{curr} <$ size of volume do
4:      Current label $l_{curr}$= $CL[I_{curr}]$
5:      Current value $v_{curr}$ = $D[I_{curr}]$
6:      $r$ = Random class label from $air$, $bone$, $liver$;
7:      $Energy_{curr}$ = $GaussianPrior(v_{curr}, l_{curr})$ + $CliquePotential(I_{curr}, l_{curr})$
8:      $Energy_{new}$ = $GaussianPrior(v_{curr}, r)$ + $CliquePotential(I_{curr}, r)$
9:      $\Delta Energy$ = $Energy_{curr} - Energy_{new}$
10:     if $\Delta Energy >$ kszi then
11:         $CL[I_{curr}]$ = $r$;
12:         summaDelta + = DiffEnergy
13:     end if
14:     Increment $I_{curr}$
15: end while
16: Update mean $\mu$ and variance $\sigma$ using the new class labels.
17: Check if the energy change in the volume is minimal (summaDelta). If not continue, steps 2
    to 16 until the energy change is minimal.
18: Increment K
```

Algorithm 2 CliquePotential(Class label 3d-array (CL), Index I, Label L)

```
1: Initialize energy, $E$ as zero
2: for all $I_{neighbor} \in$ Neighborhood of $I$ do
3:     $l_{curr} = CL[I_{neighbor}]$
4:     if $l_{curr} = l$ then
5:         $E = E - betaInClass$
6:     else
7:         $E = E - betaOutClass$
8:     end if
9: end for
```

Segmentation is the identification of non-overlapping objects of interest from images or volumes, and is a fundamental problem in image processing. In case of the liver, segmentation is critical in several diagnostic and surgical procedures. We use a Markov Random Field (*MRF*) to obtain an initial estimate of the liver boundary. *MRF*s condition the property associated with each pixel (or voxel) on its immediate neighborhood. A sample set (sample set can be an image or a volume) S is said to be an *MRF* if: $\forall s \in S, p(Y_s|Y_r, r \neq s) = p(Y_s|Y_{\delta_s})$, where s and r are individual data points (pixels in 2D, volxels in 3D), and δ_s is a neighborhood of s. An *MRF* can be modeled by taking Y as a specific property, in our case a class label assignment. In other implementations, Y may represent features being extracted from an image [18]. Algorithms 1 and 2 provide pseudocode for our *MRF* implementation.

2.1 Related Work

Liver segmentation methodologies include model-driven approaches [10, 3, 7, 9] that use a model to limit the segmentation algorithm to certain image areas, and data-driven approaches that do not use a model to restrict the image being processed [14, 19]. Our methodology falls into the data-driven approach, where we use a user-input seed point in combination with Markov Random Field to obtain an initial liver boundary and refine the boundary using an Active Contour. A 2D, non-parallel version of the algorithm has previously been published [1].

There are several 2D and 3D algorithms available for liver segmentation, and surveys are presented in [17, 13, 12]. However, few algorithms have been analyzed with respect to speedup, and fewer still have been adapted to high-speed architectures like graphics processing units.

3 HMMER Background

Protein sequence analysis tools to predict homology, structure and function of particular peptide sequences exist in abundance. Some of the most commonly used tools are part of the profile hidden markov model search algorithm, HMMER, developed by Sean Eddy [5, 4]. These tools construct hidden markov models (HMMs) of a set of aligned protein sequences with known similar function and homology, and provide database search functionality to compare input HMMs to sequence databases (as well as input sequences to HMM databases).

HMMER is composed of two search functions, *hmmsearch* and *hmmpfam*. *hmmsearch* searches an input HMM against a sequence database, while *hmmpfam* searches one or more input sequences against a database of HMMs. Both *hmmsearch* and *hmmpfam* rely on the same core algorithm for their scoring function, *P7Viterbi*. We focus our GPU implementation on *hmmsearch* as it is the more compute-intensive of the two search applications.

Algorithm 3 Pseudocode for HMMER's *hmmsearch* tool.

```
1: Input: A profile HMM, $H$ and a sequence database $S$
2: for all $i \in S$ do
3:     $score$ = P7Viterbi($H$, $S\_i$)
4:     if $score$ is significant then
5:         $PostprocessSignificantHit(S_i, H, score)$
6:     end if
7: end for
```

```
 1 for (i = 1; i <= L; i++) {
     for (k = 1; k <= M; k++) {
 3     mc[k] = mpp[k-1]   + tpmm[k-1];
       if ((sc = ip[k-1]  + tpim[k-1]) > mc[k]) mc[k] = sc;
 5     if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
       if ((sc = xmb  + bp[k])         > mc[k]) mc[k] = sc;
 7     mc[k] += ms[k];
       if (mc[k] < -INFTY) mc[k] = -INFTY;
 9
       dc[k] = dc[k-1] + tpdd[k-1];
11     if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
       if (dc[k] < -INFTY) dc[k] = -INFTY;
13
       if (k < M) {
15       ic[k] = mpp[k] + tpmi[k];
         if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
17       ic[k] += is[k];
         if (ic[k] < -INFTY) ic[k] = -INFTY;
19     }
     }
21 }
   ...
23 P7ViterbiTrace(hmm, dsq, L, mx, &tr);
```

Listing 1: The most time consuming portion of the *P7Viterbi* algorithm.

At the core of the HMMER search is the Viterbi algorithm, used to compute the most probable path through a given state model. Algorithm 3 shows the pseudocode for a typical HMMER database search, and listing 1 provides a code snippet of the most time consuming portion of the *P7Viterbi* algorithm. Line 1 from Listing 1 represents the sequence loop, while lines 2-20 represent the HMM loop. The *P7Viterbi* algorithm is sensitive to both the length of sequences in a sequence database and the length of input HMM.

As is common of database search algorithms, *hmmsearch* is embarrassingly parallel over the database loop of Algorithm 3. This results in a parallel region of over 97%, where approximately 50% of the run-time is spent in the portion of *P7Viterbi* displayed in Listing 1, lines 2-20. Therefore, the key to parallelizing a HMMER search is to offload the *P7Viterbi* function to multiple computing elements, while also ensuring that the code fragment shown in Listing 1 is as efficient as possible.

3.1 Related Work

HMMER includes a PVM (Parallel Virtual Machine) implementation of the searching algorithms. However, due to its reliance on PVM and its non-optimized messaging, its scalability is limited. MPI (Message Passing Interface) implementations are the most common parallel HMMER techniques. MPI-HMMER [21] is a well-known and commonly used implementation. In MPI-HMMER, worker nodes are assigned multiple database chunks to compute in parallel. A single master node is used to collect the results. This results in near linear speedup for small to mid-sized computational clusters (64 nodes or less).

A second Bluegene-based MPI implementation has been demonstrated to scale through 1024 nodes [8]. It uses a hierarchical master model as well as improved data collection and load balancing strategies to alleviate single master bottleneck present in MPI-HMMER. However, its reliance on a Bluegene supercomputer limits its widespread adoption.

ClawHMMer was the first GPU-enabled *hmmsearch* implementation and is capable of efficiently utilizing multiple GPUs in the form of a rendering cluster [6]. Unlike our implementation, ClawHMMer is based on the BrookGPU stream programming language [2]. Other optimizations, including several FPGA implementations, have been demonstrated in the literature [20, 16, 11]. FPGAs can achieve excellent performance, at the cost of exceptionally long development times.

4 Computing With GPUs

Computing with GPUs presents unique challenges and limitations that must be addressed in order to achieve high performance. In this section we describe the NIVIDIA 8800-based GPU that is used in our tests and also explain the unique features of the GPU that make programming them a challenge.

The graphics processors used in our tests are NVIDIA 8800 GTX Ultra GPUs with 768 MB RAM. The 8800 GTX Ultra is composed of 16 stream multiprocessors, each of which is itself composed of 8 stream processors for a total of 128 stream processors. Each multiprocessor has 8192 32-bit registers, which in practice limits the number of threads (and therefore, performance) of the GPU kernel. The GPU itself is programmed using NVIDIA's CUDA [15]. Each multiprocessor can execute 768 concurrent threads. Threads are partitioned into thread blocks of up to 512 threads each, and thread blocks are further partitioned into warps of 32 threads. Each warp is executed by a single multiprocessor. Warps are not user-controlled or assignable, but rather are automatically partitioned from user-defined blocks. At any given clock cycle, an individual multiprocessor (and its stream processors) executes the same instruction on all threads of a warp. Consequently, each multiprocessor should most accurately be thought of as a SIMD processor.

Programming the GPU is not a matter of simply mapping a single thread to a single stream processor. Rather, with 8192 registers per multiprocessor, hundreds of threads per multiprocessor and thousands of threads per board should be used in order to fully utilize the GPU. Memory access patterns, in particular, must be carefully studied in order to minimize the number of global memory reads. Where possible, an application should make use of the 16 KB of shared memory per multiprocessor, as well as the texture memory, in order to minimize GPU kernel access to global memory. When global memory must be accessed, it is essential that memory be both properly aligned, and laid out such that each SIMD thread accesses consecutive array elements in order to combine memory reads into larger 384-bit reads.

5 GPU Implementations and Results

In this section we describe the GPU implementations of *MRF* liver segmentation and the *P7Viterbi* algorithm. We provide details and performance results of optimizations for both GPU kernels. All GPU tests were performed on a machine consisting of a 3.0 Ghz AMD Athlon 642 processor with 8 GB memory and 2 NVIDIA 8800 GTX Ultra GPUs. Only a single GPU was used in our tests. The *MRF* serial tests were also taken on the AMD Athlon machine. All *hmmsearch* serial tests were performed on machines consisting of a single quad-core 2.0 Ghz Intel Xeon processor with 4 GB RAM. Only a single core was used for the serial tests.

5.1 *MRF* Liver Segmentation Kernel

Algorithms 1 and 2 outline the major steps in the *MRF* computation that provides an approximate liver boundary from the 3D CT volume. The CT volume is a stack of multiple 512 X 512 images, for example a 512 X 512 X 60 volume has 60 CT images in it; x and y values of the volume will range from 0 to 511 and the z co-ordinate will range from 0 to 59. In our GPU approach, lines 3 through 15 of algorithm 1 are implemented on the GPU, and multiple threads are used to iterate through the volume. Each thread is assigned a particular x coordinate, or a range of x-coordinates according to its thread Id; the y and z co-ordinates will iterate through 0-511 and 0-N respectively, where N is number of images in the volume.

In our GPU implementation, multiple threads process the volume in parallel and update class label values simultaneously. The GPU architecture does not permit these updates to be synchronized across threads. Hence, calls to $CliquePotential$ in lines 7 and 8 of algorithm 1 (Algorithm 2) do not have guaranteed access to updated class labels along the x-coordinate. In the sequential implementation on CPU, updated class label values of the entire volume are available. Since class labels are not available across threads, the GPU implementation is a departure from the *MRF* model. Empirical analysis shows that the effect on segmentation result is insignificant in practice.

A primary optimization in our implementation is memory coalescing. Coalescing is a technique to combine non-sequential and small reads from global memory, into the more efficient sequential and large global memory reads. This minimizes the penalty of reading from memory. Reads by consecutive threads in a warp are combined by hardware into several, wider memory reads of up to 384 bits each. Consecutive 32-bit reads that are issued simultaneously are automatically merged into multiple 384-bit reads in order to efficiently saturate the memory bus. For the GPU to be able to coalesce memory reads, we have modified the implementation such that threads within a warp read memory sequentially. The entire class label values of the 3D volume are laid out in a single dimensional array. The neighboring x coordinates lie close together and threads operate on this x coordinate in order, leading to coalesced reads for nearly every access to the global memory. If multiple GPU threads are reading from the same array beginning at offset n then thread 0 should read (assuming 32-bit array elements) `array[n]`, thread 1 should read `array[n+1]`, etc.

MRF code occupies 32 registers on the GPU. Each multiprocessor on the NVIDIA 8800 Ultra has 8,192 register, limiting the maximum number of MRF threads per multiprocessor to 256. Various configurations of ”number of blocks X threads per block X registers per thread” have been tried and the best speedup has been obtained for 16 X 32 X 32 (Table 1). Higher register count limits the number of threads per multiprocessor, and reduces the occupancy count (Ratio of number of threads per multiprocessor to 768, the maximum possible threads per multiprocessor. For a register count of 32, occupancy is 33%, and with 12 registers, the occupancy count is 67%. Nevertheless, occupancy count is not the best indicator of speedup, as seen in Table 1.

Figure 1 presents speedup results for different configurations. It appears that speedup increases with an increase in the block size as well as total number of threads. However, when the number of blocks is constant, but the number of threads are increased, speedup changes more significantly than with change in number of blocks alone. Significant difference is seen in the speedup of configurations with different thread counts; examples include the configurations 2x32x8 versus 2x256x1, 4x32x4 versus 4x128x1, and 8x32x2 versus 8x64x1. Also, the increase in speedup is comparatively lesser when multiple blocks are used without increasing thread count; for example configurations 2x64x4 versus 4x32x4 and configuration 4x64x2 versus 8x32x2. Overall, our results show that the total number of threads executed has a more significant effect than block count. This is also attested in Table 1 where relatively less increase in speedup is achieved while going from 4 blocks to 16 blocks.

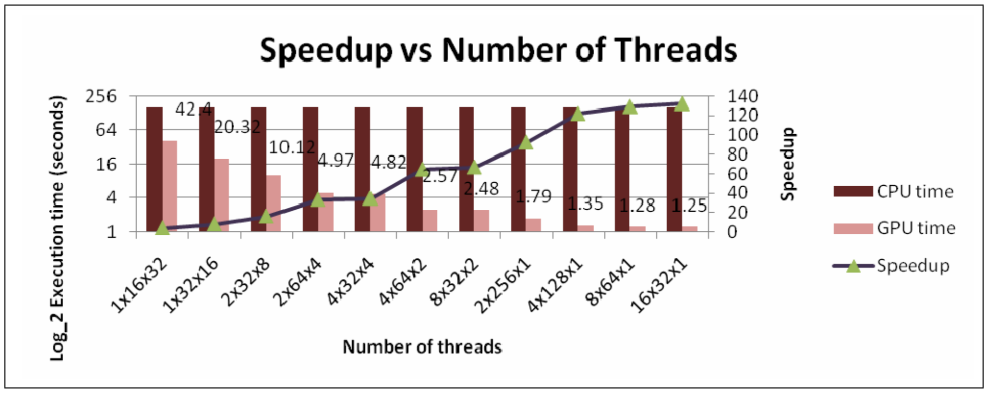

Figure 1: Speedup of *MRF* as a function of the number of threads; size of CT volume: 512 x 512 x 77

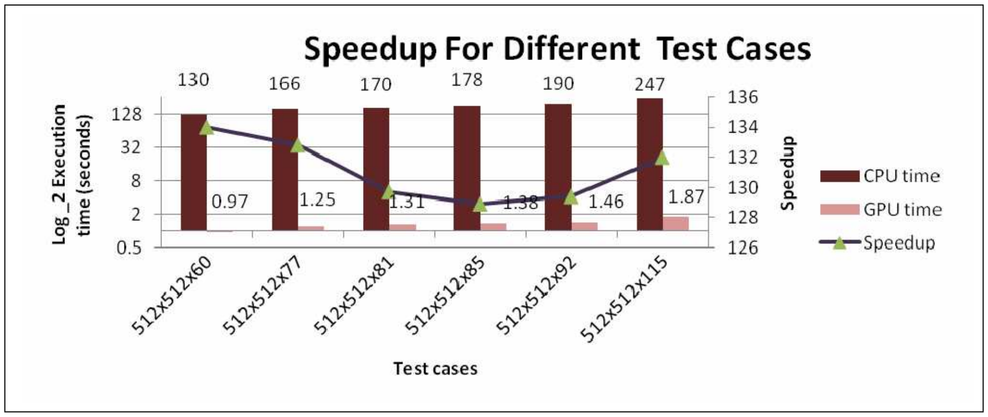

Figure 2: Speedup of *MRF* for multiple test cases.

Table 1: Speedup for varying block count and register usage for MRF; size of CT volume: 512 x 512 x 81

Blocks	Threads per block	Registers per thread	Occupancy count	Execution Time	Speedup
1	512	12	67%	3.92	43
2	256	32	33%	2.05	83
4	128	32	33%	1.45	117
8	64	32	33%	1.34	127
16	32	32	17%	1.31	130

5.2 *P7Viterbi* Kernel

C code of the *P7Viterbi* algorithm was ported to CUDA with performance optimizations. The kernel works on multiple sequences simultaneously, with each thread working on a unique sequence. The number of threads that can be executed in parallel will be limited by two factors: (i) *GPU memory* will limit the number of sequences that can be stored, and (ii) *The number of registers used by each thread* will limit the number of threads that can run in parallel. Typically, register use is the most prohibitive resource.

The *P7Viterbi* kernel in our implementation requires 32 registers per thread, allowing a maximum of 256 active threads per multiprocessor. NVIDIA 8800 GTX Ultra has 16 multiprocessors, and each can run 4096 ($256*16$) threads in parallel. In the remainder of this section we describe the optimizations made to the GPU kernel. We consider three primary optimizations in our *P7Viterbi* kernel: database-level load balancing, memory layout and coalescing, and loop unrolling.

As we described in Section 3, HMMER's *P7Viterbi* function is sensitive to both the length of the query HMM as well as the length of an individual sequence. CUDA provides limited support for thread synchronization; a barrier synchronization function is provided that returns only when all threads have finished execution ($cudaThreadSynchronize()$). In our implementation, 4096 threads are run in parallel on a single multi-processor, with each thread operating on its own se-

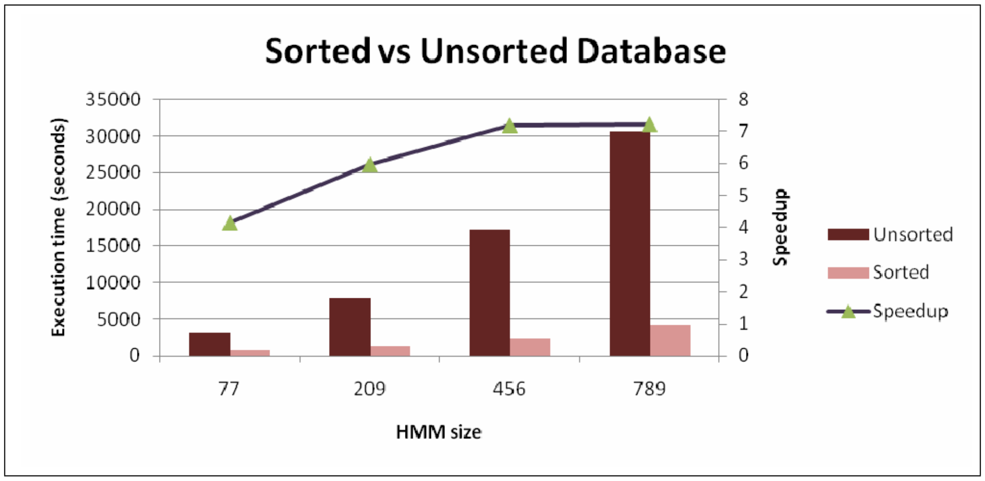

Figure 3: Speedup of *hmmsearch* with sorted database.

quence. A typical sequence database is unordered, placing short sequences in a close vicinity to long sequences. On a CUDA-enabled GPU this results in shorter sequences completing early, and being forced to wait for the longest sequence in the current batch before the barrier synchronization completes. The solution is to presort the sequence database by length, thereby balancing a similar load over all 4,096 threads participating in the computation.

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1] + tpmm[k-1];
}
```

Listing 2: Original loop

```
for (k = 1; k <= M; k+=4) {
  mc[k] = mpp[k-1] + tpmm[k-1];
  mc[k+1] = mpp[k] + tpmm[k];
  mc[k+2] = mpp[k+1] + tpmm[k+1];
  mc[k+3] = mpp[k+2] + tpmm[k+2];
}
```

Listing 3: Unrolled loop

Loop unrolling is a classic loop optimization strategy designed to reduce the overhead of inefficient looping. The idea is to replicate the loop's inner contents such that the ratio of useful computation to loop bounds computation increases. The same principles apply to GPU computation, with the caveat that loop unrolling may introduce additional register pressure. In GPU programming, the use of additional registers may reduce the number of active threads, further reducing the overall GPU utilization. Listings 2 and 3 provides an example of the loop unrolling transformation for a portion of the k-loop of Listing 1. We have experimentally determined an unrolling factor of 2 to provide modest performance improvements for most cases. However, due to space considerations we must omit this data.

The most effective optimization to the *P7Viterbi* is from optimizing memory layout and usage patterns within the Viterbi algorithm. Because the CUDA environment does not allow threads to dynamically allocate GPU memory, all memory allocations (even those allocating the GPU's on-board memory) must be performed by the host system and copied to the GPU before instantiating the kernel. By default, the *P7Viterbi* function requires integer arrays of size $3*M*L+5*L$, where

M, L are the length of the sequence and HMM, respectively. For large HMMs and large sequences, this can easily result in several megabytes of data per thread. With only 768 MB memory for 4096 threads, this can exhaust of the GPU's memory. Through careful optimization we are able to reduce the memory requirements of the *P7Viterbi* scoring computation to $6 * M + 10$ integer array elements.

Reducing the memory footprint means that we can no longer perform the trace back procedure on line 23 of Listing 1. Fortunately, the trace back is only needed when a database hit has been made. In our tests less than 2% of the database results in hits, so we simply perform a full software *P7Viterbi* include track back on all database hits. This is a common strategy in hardware accelerators, particularly FGPAs [16].

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1] + tpmm[k-1];
}
```

Listing 4: Non-coalesced memory

```
for (k = 1; k <= M; k++) {

  mc[k*CHUNK+idx] =
   mpp[(k-1)*CHUNK+idx] +
   tex1Dfetch(tscTex, TMM*M + k-1);
}
```

Listing 5: Coalesced memory with texture

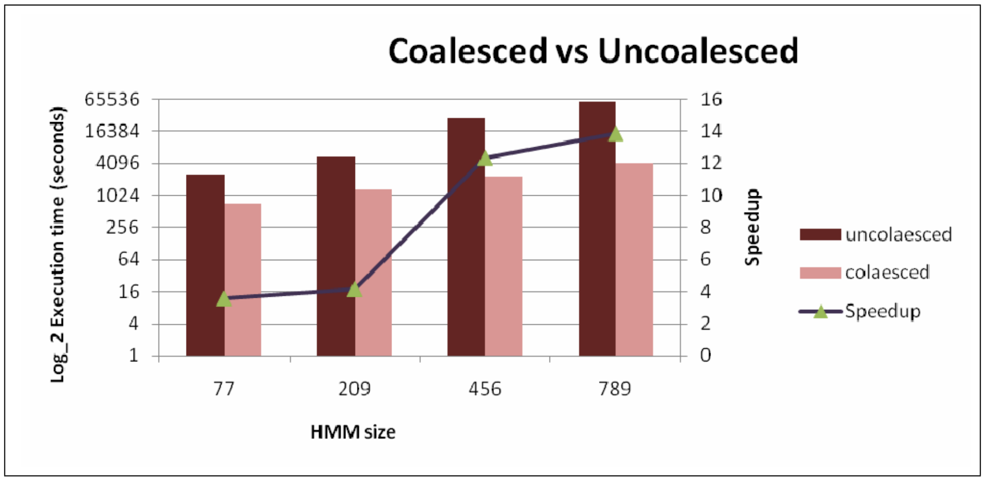

Figure 4: Performance improvements after applying memory coalescing.

We also make use of high speed texture memory to store both the current sequence batch as well as the HMM. Because the HMM is static through the search, it is well suited to read-only texture memory. Similarly, the sequence data itself is read-only, and each batch of sequences can be bound to texture memory prior to a GPU kernel invocation. Memory coalescing has also significantly improved *hmmsearch*'s overall speedup. In Listing 5 we provide an example of the changes needed to improve memory reads. The `mc` and `mpp` arrays are both coalesced while the `tpmm` array is stored in texture memory. Both `mc` and `mpp` point to different rows of the same array, `mmx`. By default each thread uses its own copy of the `mmx` array, reading each element

Table 2: *P7Viterbi* occupancy data (Threads per Block: 256, Registers per Thread: 32)

Active Threads per Multiprocessor	256
Active Warps per Multiprocessor	8
Active Thread Blocks per Multiprocessor	1
Occupancy of each Multiprocessor	33%
Maximum Simultaneous Blocks per GPU	16

starting from `mpp[0]` and proceeding through `mpp[M-1]`. In a GPU, however, this is inefficient as such an access pattern will result in multiple 32-bit reads.

We reorganize all `mmx` arrays into a single array, and reorganize the read pattern such that the first 4,096 elements (for 4,096 threads) correspond to `mpp[0]` in respective threads. In Listing 5 the variable `idx` corresponds to a thread ID and `CHUNK` denotes the number of threads. Thus, `idx = 0` will read `mpp[0]`, `idx = 1` reads `mpp[1]`, etc. All threads access identical elements of the HMM, so the `tscTex` array is stored as a single dimensional array in texture memory. Figure 4 shows the results of applying memory coalescing to the *P7Viterbi* algorithm. This provided the greatest performance increase of all optimizations, resulting in a speedup of more than 9 for larger HMMs. In Table 2 we present the occupancy of our *P7Viterbi* kernel. Due to the high register pressure, the utilization is limited to 33%.

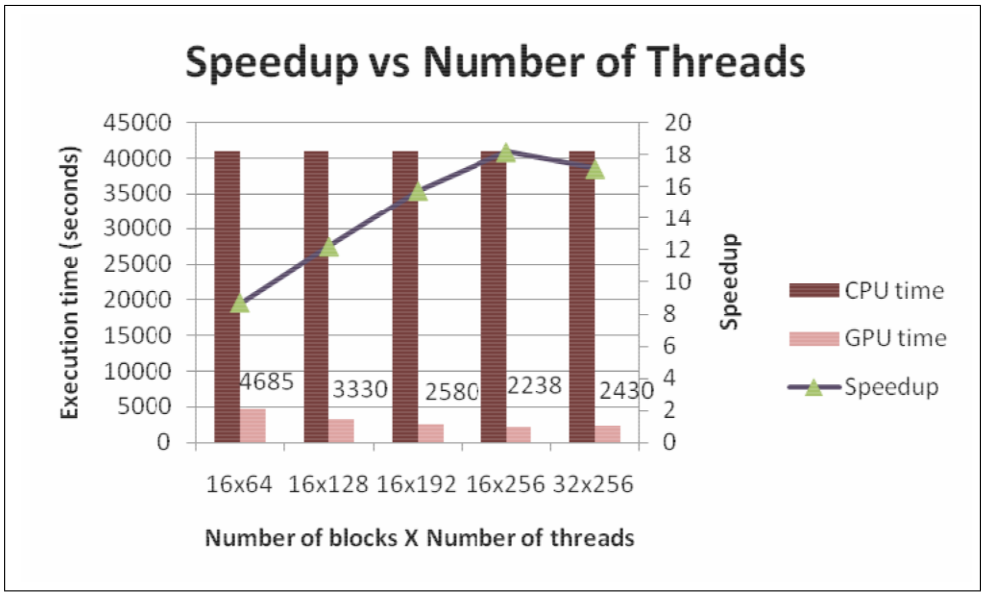

Figure 5: *hmmsearch* speedup as a function of the number of threads.

In Figure 5 and 6 we present the overall performance improvement results, including all optimizations. Figure 5 shows that the GPU offers best performance with 4,096 threads. This is to be expected, given that our *P7Viterbi* requires 32 registers per thread. With 8192 registers for each of the 16 multistream processors, we have perfectly consumed all registers within the GPU. By doubling the number of threads to 8192, we force the GPU to spend more time in context switches and

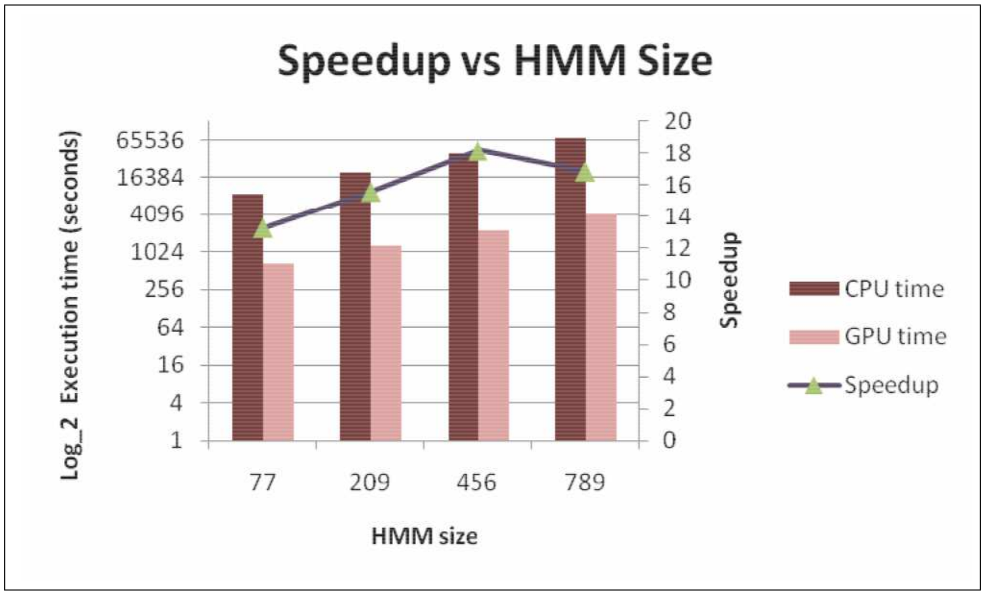

Figure 6: Speedup as a function of the size of the HMM, 4096 threads.

spilling registers to memory, resulting in reduced performance. In Figure 6, we present the results of performing multiple searches with varying HMM sizes. The trend is toward greater speedup for increasing HMM sizes. This is expected due to the k-loop of the *P7Viterbi* relying on the length of the HMM (Listing 1). Nevertheless, this does not always hold as evidenced by the slightly lower performance of the 789 state HMM in Figure 6. This effect has been previously shown in the literature, and we are actively searching for its cause [6].

6 Discussion

A comparison of the speedup from our GPU implementations of MRF (Figure 2) and *P7Viterbi* / *hmmsearch* (Figure 6) shows that significantly higher speedup is achieved in the MRF implementation. In this section we consider the major factors that resulted in different speedups. MRF's major advantage over *hmmsearch* is that the former makes very few reads from the GPU's global memory; at each iteration, MRF accesses global memory only twice. However, *hmmsearch* is forced to repeatedly access global memory within the inner-most loop of Listing 1. Since this loop is repeated over the entire length of the sequence, the *P7Viterbi* and, consequently, *hmmsearch* spend a large portion of their run-time accessing global memory. Because of this repeated global memory access in *hmmsearch*, memory coalescing proved more effective in HMMER than in our MRF implementation. This was unsurprising, considering MRF's limited use of global memory.

Loop unrolling proves more effective in *P7Viterbi* than in MRF. The Viterbi algorithm has a limited number of variables needed in the core loop, and lends itself nicely to unrolling the inner loop contents. However, MRF requires the use of comparatively larger number of variables in its inner-loop. Unrolling these iterations results in increased register usage for temporary variables, leading to reduced performance.

The MRF code ultimately proved to be better suited for acceleration on GPUs. Due to the architectural requirements of the NVIDIA GPU, any thread participating in a warp will, by definition execute the same instructions simultaneously. This essentially turns the GPU into a large SIMD processor. MRF is a natural fit for such architectures as its inner-loop is relatively free of branches with each thread operating on the same set of images simultaneously.

6.1 Conclusion

We have presented the performance of two statistics-based life science applications, *MRF*-based segmentation, and HMMER's *hmmsearch* database searching tool. Both applications demonstrated reasonable performance improvement on the GPU, with *MRF* exhibiting a speedup of over **130x** compared to serial execution. As we have shown, significant effort is required in order to properly leverage a GPU for general purpose computing. Moreover, we have demonstrated that algorithms must properly target the GPU in order to achieve performance improvements. This includes attention to the occupancy of the GPU kernel, loop unrolling, and most importantly memory coalescing. Looking forward, our next goal is to leverage multiple GPUs within a single workstation, and ultimately GPU-based workstation clusters in order to further optimize the performance of our applications.

References

[1] R. S. Alomari, S. Kompalli, S. T. Lau, and V. Chaudhary. Design of a Benchmark Dataset, Similarity Metrics, and Tools for Liver Segmentation. In *Proceedings of the 2008 SPIE Medical Imaging Conference* , 2008.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[3] E. Chen, P. Chung, C. Chen, H. Tsai, and C. Chang. An Automatic Diagnostic System for CT Liver Image Classification. *IEEE Transactions on Biomedical Engineering*, 45:783–794, 1998.

[4] R. Durbin, S. Eddy, A. Krogh, and A. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[5] S. R. Eddy. Profile Hidden Markov Models. *Bioinformatics*, 14(9), 1998.

[6] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A Streaming HMMer-Search Implementation. In *In proceedings of SC '05: The International Conference on High Performance Computing, Networking and Storage*, 2005.

[7] S. Huang, B. Wang, and X. Huang. Using GVF Snake to Segment Liver from CT Images. In *Proceedings of 3rd IEEE/EMBS International Summer School on Medical Devices and Biosensors, 2006*, pages 145–148, 2006.

[8] K. Jiang, O. Thorsen, A. Peters, B. Smith, and C. P. Sosa. An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence Search on a Massively Parallel System. *Transactions on Parallel and Distrbuted Systems*, 19(1):15–23, 2008.

[9] C. Krishnamurthy, J. J. Rodriguez R. J., and Gillies. Snake-based Liver Lesion Segmentation. In *Southwest04*, pages 187–191, 2004.

[10] F. Liu, B. Zhao, P. K. Kijewski, L. Wang, and L. H. Schwartz. Liver Segmentation for CT Images Using GVF Snake. *Medical Physics*, 32(12):3699–3706, December 2005.

[11] R. P. Maddimsetty, J. Buhler, R. Chamberlain, M. Franklin, and B. Harris. Accelerator Design for Protein Sequence HMM Search. In *Proc. of the 20th ACM International Conference on Supercomputing (ICS06)*, pages 287–296. ACM, 2006.

[12] Y. Masutani, K. Uozumi, M. Akahane, and K. Ohtomo. Liver CT Image Processing: A Short Introduction of the Technical Elements. *European Journal of Radiology*, 58:246–251, may 2006.

[13] T. McInerney and D. Terzopoulos. Deformable Models in Medical Images Analysis: A Survey. *Medical Image Analysis*, 1(2), 1996.

[14] Y. Nakayama, Q. Li, S. Katsuragawa, R. Ikeda, Y. Hiai, K. Awai, S. Kusunoki, Y. Yamashita, H. Okajima, Y. Inomata, and K. Doi. Automated Hepatic Volumetry for Living Related Liver Transplantation At Multisection CT1. *Radiology*, 240(3), September 2006.

[15] NVIDIA. *Compute Unified Device Architecture (CUDA) Programming Guide*. NVIDIA, 1.0 edition, 2007.

[16] T. F. Oliver, B. Schmidt, J. Yanto, and D. L. Maskell. Acclerating the Viterbi Algorithm for Profile Hidden Markov Models using Reconfigurable Hardware. *Lecture Notes in Computer Science*, 3991:522–529, 2006.

[17] M. Pham, R. Susomboon, T. Disney, D. Raicu, and J. Furst. A Comparison of Texture Models for Automatic Liver Segmentation. In *Medical Imaging 2007: Image Processing. Edited by Pluim, Josien P. W.; Reinhardt, Joseph M.. Proceedings of the SPIE, Volume 6512, pp. 65124E (2007).*, volume 6512 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, mar 2007.

[18] C. Philips, R. Susomboon, R. Mokhtar, D. Raicu, and J. Furst. Segmentation of Soft Tissue Using Texture Features and Gradient Snakes. Technical Report TR07-011, CTI DePaul, 2007.

[19] P. Regina and K. D. Toennies. A New Approach for Model-Based Adaptive Region Growing in Medical Image Analysis. In *CAIP '01: Proceedings of the 9th International Conference on Computer Analysis of Images and Patterns*, pages 238–246, Otto-von-Guericke University Magdeburg,, Department of Simulation and Graphics, London, UK, Regina@isg.cs.uni-magdeburg.de,Klaus@isg.cs.uni-magdeburg.de, 2001. Springer-Verlag.

[20] TimeLogic BioComputing Solutions. DecypherHMM. `http://www.timelogic.com/`, 2006.

[21] J. P. Walters, B. Qudah, and V. Chaudhary. Accelerating the HMMER Sequence Analysis Suite Using Conventional Processors. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 289–294, Washington, DC, USA, 2006. IEEE Computer Society.