Evaluating the use of GPUs in Liver Image Segmentation and HMMER Database Searches*

John Paul Walters, Vidyananth Balu, Suryaprakash Kompalli†, and Vipin Chaudhary
Department of Computer Science and Engineering
University at Buffalo, SUNY
Buffalo, NY
{waltersj, vbalu2, vipin}@buffalo.edu
† Hewlett-Packard Laboratories, Bangalore, India
kompalli@hp.com

Abstract

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, using GPUs. 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. We test multiple enhancements in our GPU kernels in order to demonstrate the effectiveness of each strategy. Our optimized MRF kernel achieves over 130x speedup, and our hmmsearch implementation achieves up to 38x speedup. We show that the differences in speedup between MRF and hmmsearch is due primarily to the frequency at which the hmmsearch must read from the GPU's DRAM.

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 or hundreds of traditional CPU cores at a fraction of the cost. As tools such as NVIDIA's CUDA [18] 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 exceeding 1 TFLOP/s for the latest NVIDIA GPUs, their attraction for general 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 within the life science domain: the HMMER sequence database search application (*hmmsearch*), and a medical imaging liver segmentation application based on Markov random fields (*MRF*). HMMER's *hmmsearch* tool is particularly well-suited for many-core architectures due to the embarrassingly parallel nature of sequence database searches. *MRF* can be made similarly parallel. However, the many-core architecture of the GPU also creates challenges to HMMER and *MRF* implementations. CUDA exposes multiple levels of memory to the programmer that must be managed. Further, memory layout and load balancing are critical performance barriers that must be overcome to maximally utilize the GPU. We examine these issues and 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

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

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
```

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. Our future work is presented in Section 7.

2. *MRF* Segmentation

Segmentation is the identification of non-overlapping objects of interest from images or volumes, and is a fundamental problem in image processing. In the 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, voxels 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 [21]. Algorithms 1 and 2 provide pseudocode for our *MRF* implementation.

2.1. Related Work

Liver segmentation methodologies include model-driven approaches [3, 8, 10, 11] 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 [16, 22]. Our methodology falls into the data-driven approach, where we use a user-input seed point in combination with the Markov random field to obtain an initial liver boundary. We then 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. Masutani et al. McInerney and Terzopoulos, and Pham et al. provide surveys of the techniques [14, 15, 20]. 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 tool, HMMER, developed by Sean Eddy [5, 6]. 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++) {
    ...
 3   for (k = 1; k <= M; k++) {
       mc[k] = mpp[k-1]   + tpmm[k-1];
 5     if ((sc = ip[k-1]  + tpim[k-1]) >
        mc[k]) mc[k] = sc;
 7     if ((sc = dpp[k-1] + tpdm[k-1]) >
        mc[k]) mc[k] = sc;
 9     if ((sc = xmb  + bp[k])         >
        mc[k]) mc[k] = sc;
11     mc[k] += ms[k];
       if (mc[k] < -INFTY) mc[k] = -INFTY;
13
       dc[k] = dc[k-1] + tpdd[k-1];
15     if ((sc = mc[k-1] + tpmd[k-1]) >
        dc[k]) dc[k] = sc;
17     if (dc[k] < -INFTY) dc[k] = -INFTY;

19     if (k < M) {
         ic[k] = mpp[k] + tpmi[k];
21       if ((sc = ip[k] + tpii[k]) >
          ic[k]) ic[k] = sc;
23       ic[k] += is[k];
         if (ic[k] < -INFTY) ic[k] = -INFTY;
25     }
     }
27  ...
    }
29  ...
    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 3-26 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.

We note that the majority of the array references in Listing 1 are to one of three dynamic programming matrices, `mmx`, `imx`, and `dmx`. The matrices `mc` and `mpp`, for example, reference the current and previous rows of the `mmx` matrix. Similarly, the `ic` and `ip` matrices refer to the current and previous row of the `imx` matrix. Finally, the `dc` and `dpp` matrices refer to the current and previous rows of the `dmx` matrix. These array assignments occur between the `i` and `k` loops of Listing 1, but are not shown. Other matrices, such as the `tpXX` series contain the transition scores associated with the HMM, and are referenced repeatedly within the `k` loop. The digitized sequence is itself accounted for within both the `ms` and `is` arrays. This will become more important when we describe our memory optimizations in Section 5.2.

As is common of database search algorithms, *hmmsearch* is embarrassingly parallel over the database loop of Algorithm 3. After profiling, we found that over 97% of the runtime was spent in the *P7Viterbi* function, where approximately 50% of the run-time is spent in the portion of *P7Viterbi* displayed in Listing 1, lines 3-26. 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 [24] 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 [9]. It uses a hierarchical master model as well as improved data collection and load balancing strategies to alleviate the 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 [7]. 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 [13, 19, 23]. GPUs have also been used to accel-

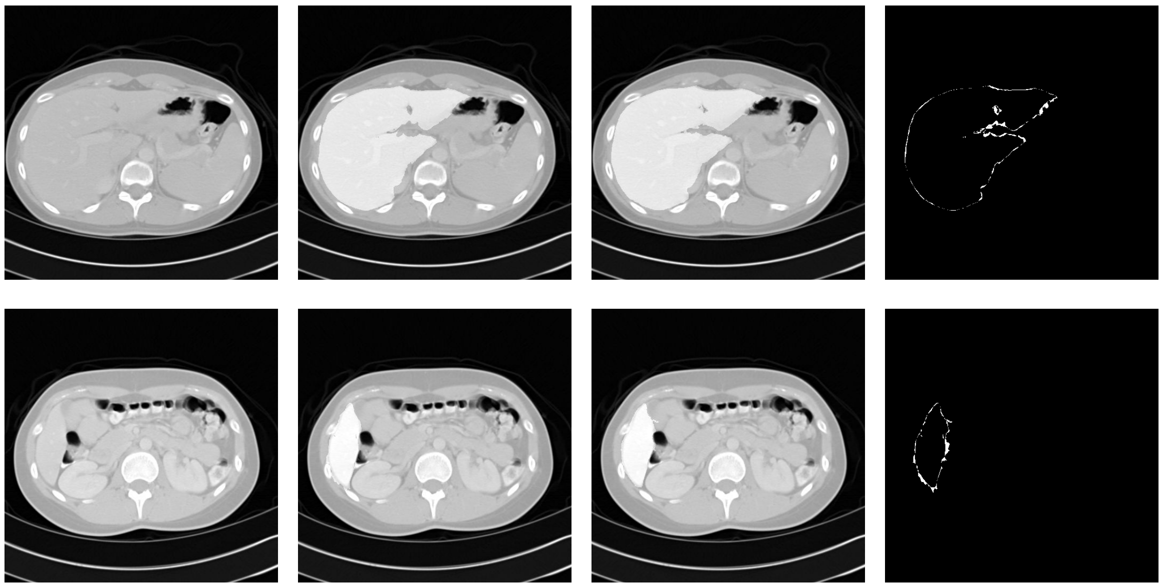

Figure 1. Initial boundary of the liver obtained from *MRF* segmentation, Difference in CUDA and serial results: top slice 3.86%, bottom slice 11.07%. Left to right: Image of the abdominal area, CUDA output, serial output, difference between CUDA and serial outputs.

erate other sequence analysis algorithms, including the Smith-Waterman algorithm [12].

Beyond HMMER, hidden Markov models have been applied to a variety of compute intensive problems. Chong et al. describe an HMM-based Viterbi algorithm for speech recognition applications. Their GPU implementation achieved approximately 9x speedup using the NVIDIA 8800 GTX [4].

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 NVIDIA 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 is programmed using NVIDIA's CUDA programming model [18]. Each multiprocessor can manage 768 active 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 and 64 KB constant 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. We discuss these optimizations in more detail in Section 5.

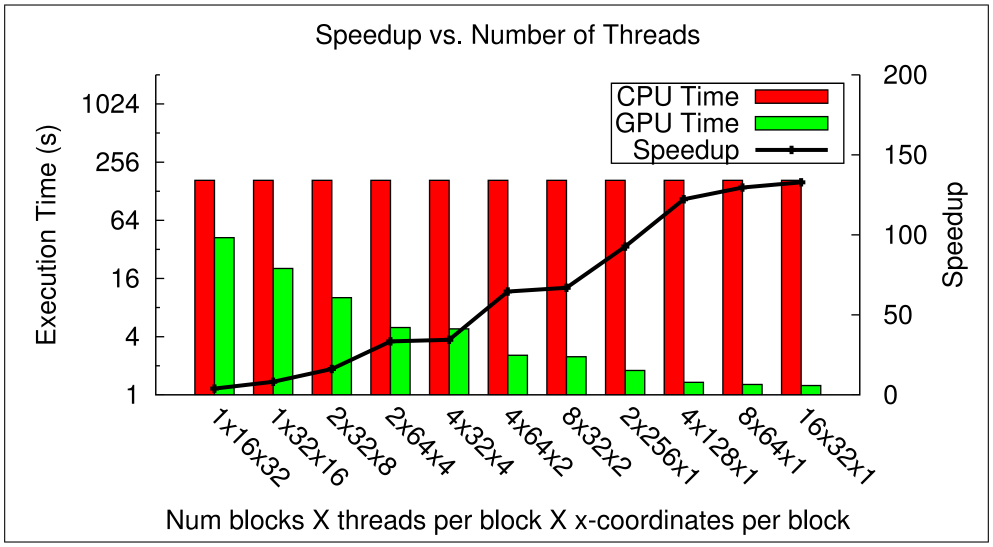

Figure 2. Speedup of *MRF* as a function of the number of threads; size of CT volume: 512 x 512 x 77

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 and serial tests were performed on a machine consisting of a 2.2 GHz AMD Athlon 275 processor with 8 GB memory and 2 NVIDIA 8800 GTX Ultra GPUs. A single GPU was used in our 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 the 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 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. Both the serial and CUDA versions of *MRF* segmentation have been tested on a publicly-available dataset [1]. Our results indicate that segmentation from both the serial and CUDA versions differ only with respect to outlier points. The difference in segmentation boundaries is computed by taking dissimilar areas from the serial and CUDA results, and computing the ratio of this area with the CUDA segmentation area. The difference ratio ranges from a minimum of 2.86% to a maximum of 22.76%, with an average of 9.35%. Large differences ($>$16%) occur at extremities of the liver region. The liver has a small size at the extremities, and even a small variation in segmentation accounts for a large percent error. Figure 1 shows two results from the serial and CUDA implementations. The *MRF* segmentation results serve as inputs to a snake-based method that refines these initial estimates of the liver boundary.

A primary optimization in our implementation is memory coalescing. Coalescing is a technique used 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. On the 8800 GTX, Consecutive 32-bit

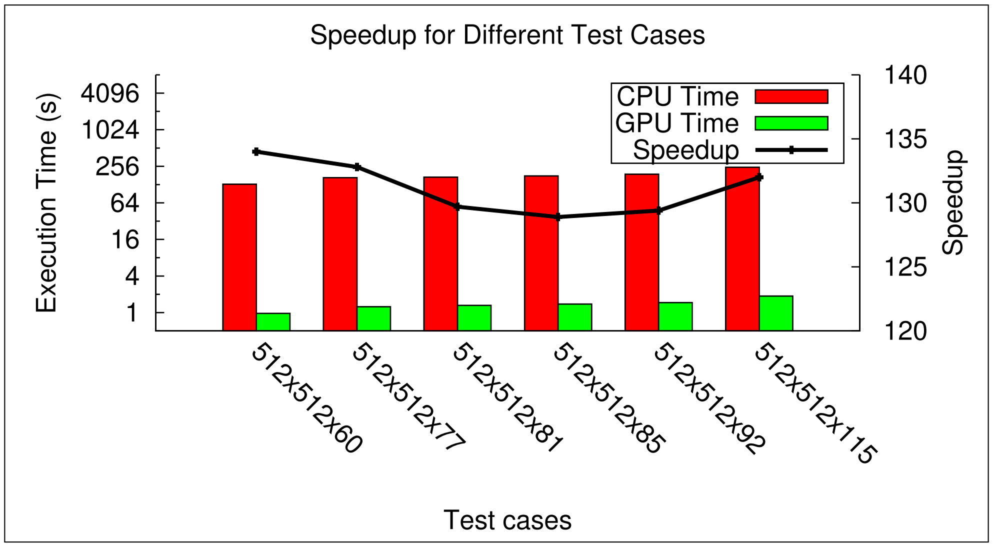

Figure 3. Speedup of *MRF* for multiple test cases.

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 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.

Figure 2 presents speedup results for different configurations. A configuration in this context refers to, "number of blocks $\times$ threads per block $\times$ number of x coordinates per thread." It appears that speedup increases with an increase in the block size as well as an increase in the total number of threads. However, when the number of blocks is held constant with increasing thread counts, 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 less than when multiple blocks are used without increasing thread count. For example, configuration 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.

In Figure 3 we present the results of our *MRF* kernel with increasing numbers of CT slices. As we show, the smallest volume demonstrates the highest speedup of 134x for the 60 image CT volume. The 77 image CT volume used in Figure 2 achieved nearly 133x speedup. None of the volumes achieved a speedup of less than 129x, when compared to a single core of an Athlon 275 processor.

5.2. *P7Viterbi* Kernel

The C code of the *P7Viterbi* algorithm was ported to CUDA with a variety of performance optimizations. The kernel operates on multiple sequences simultaneously, with each thread operating on a unique sequence. The number of threads that can be executed in parallel will be limited by two factors: (1) *GPU memory* will limit the number of sequences that can be stored, and (2) *The number of registers used by each thread* will limit the number of threads that can run in parallel. In our implementation, register use is the most prohibitive resource.

The *P7Viterbi* kernel in our initial implementation requires 32 registers per thread, allowing a maximum of 256 active threads per multiprocessor. The NVIDIA 8800 GTX Ultra has 16 multiprocessors, and can therefore maintain 4096 ($256 * 16$) active threads. This is accomplished by splitting the `k` loop from Listing 1 into three independent `mc`, `dc`, and `ic` loops. The advantage to this strategy is that fewer registers are required, resulting in higher GPU utilization. Further, splitting the loops provides an easy mechanism to employ loop unrolling. The disadvantage is that we lose most oppor-

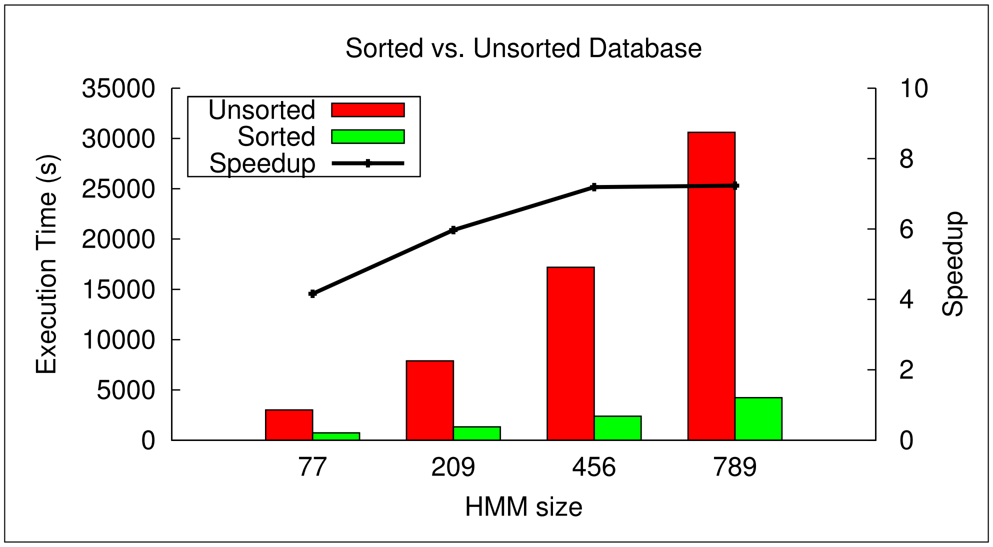

Figure 4. Speedup of *hmmsearch* with sorted database.

tunities for reusing itermediate data. We revisit point in our optimized kernels.

In the remainder of this section we describe the optimizations made to the GPU kernel. We consider a variety of optimizations in our implementation including database-level load balancing, memory layout and coalescing, loop unrolling, and shared/constant memory use. The results presented in this section show the impact of each of these optimizations (and several others) as we lead the reader through our implementation strategy, ending with our highest performing implementation. Results are shown with a variety of HMMs of increasing length.

We note that the average length of an HMM within the commonly available Pfam database is 209 states. Within the context of hidden Markov models, the *length* of an HMM corresponds to the number of match states encoded in a specific model/HMM. A match state (e.g. m_i), as opposed to an insert state or delete state, indicates that a target sequence aligns to the model at state m_i. More importantly (for the purpose of this paper), the length corresponds to the array bounds of the innermost M-loop from Listing 1. Thus, HMMs of greater length result in longer running computations.

We test HMMs of length 77, 209, 456, 789, and 1431 states. All HMMs except the 77 state HMM were taken directly from the Pfam database, while the 77 state HMM is distributed with the HMMER source. All tests are taken against the publicly available NCBI non-redundant database (NCBI `NR` [17]). The 3 GB `NR` database used in these tests consists of over 5.5 million sequences with sequence lengths varying from 6–37,000 amino acids.

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 initial implementation, 4096 threads are run in parallel on a single GPU, with each thread operating on its own sequence. A typical database is unordered, placing short sequences in close vicinity to long sequences. On a CUDA-enabled GPU this results in threads operating on the shorter sequences completing early, and being forced to wait for the thread computing 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. This has the advantage of being both effective and quite straightforward as we are able to achieve a nearly 7x performance improvement over the unsorted database without changing the GPU kernel in any way (see Figure 4). For the database used in these experiments, only 262.36 seconds were required for sorting. Further, the sorted database can be reused for the entire useful life of the database, making the one-time cost of sorting it negligible.

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 pro-

```
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. Loop unrolled four times

```
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

gramming, the use of additional registers may reduce the number of active threads, further reducing the overall GPU utilization. Listings 2 and 3 provide 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 (approximately 0.8x improvement).

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 and 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 quickly exhaust 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. This was accomplished by noting that the *P7Viterbi* algorithm shown in Listing 1 largely requires only the current and previous rows of the dynamic programming matrices `mmx`, `imx`, and `dmx` over the length of the inner-most loop, `M` (the number of HMM states). Thus, `mmx`, `imx`, and `dmx` contribute $6 * M$ array elements. The `xmb` and several temporary scoring elements contribute the remainder of the memory requirements for scoring.

Reducing the memory footprint means that we can no longer perform the trace back procedure on line 30 of Listing 1. Fortunately, the trace back is only needed when a database hit is made. In our tests fewer than 2% of the database results in hits, so we simply perform a full software *P7Viterbi* including traceback on all database hits. This is a common strategy in hardware accelerators, particularly FPGA [19].

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. As a reminder 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 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 non-coalesced reads.

We reorganize all `mmx` (as well as `dmx` and `imx`) 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

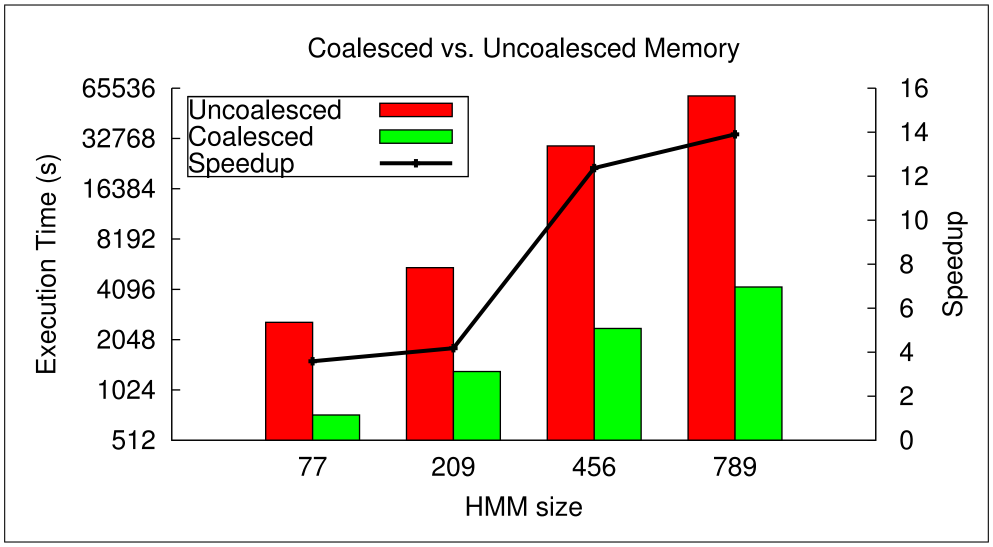

Figure 5. Performance improvements after applying memory coalescing.

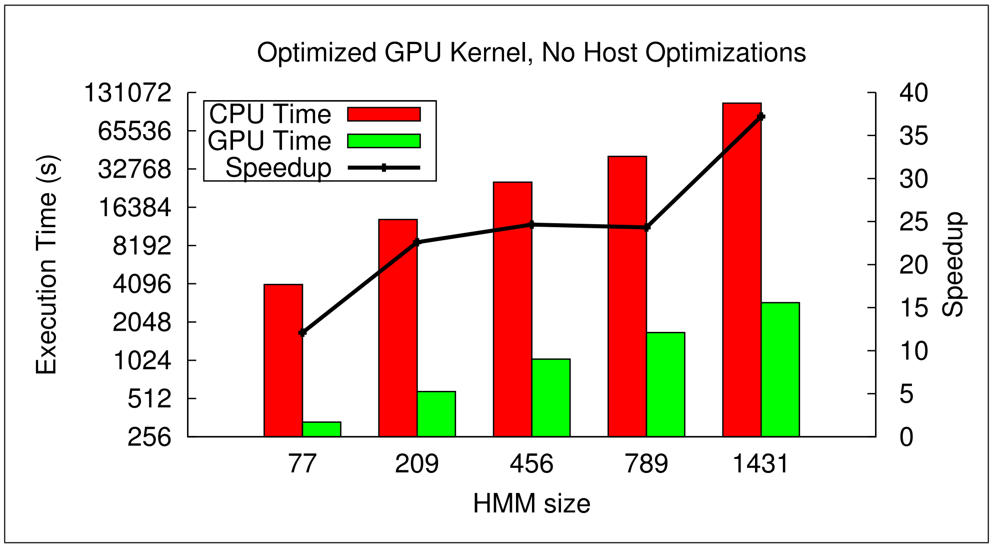

Figure 6. Performance of final GPU kernel.

texture memory. Figure 5 shows the results of applying memory coalescing to the *P7Viterbi* algorithm. This resulted in an improvement of more than $9x$ for larger HMMs.

We now consider our second and final *P7Viterbi* design. Rather than splitting the `k` loop into three independent `mc`, `ic`, and `dc` kernels, we express the *P7Viterbi* similarly to that shown in Listing 1. In doing so, we include the enhancements described above in addition to the use of constant and shared memory. The advantage to this strategy is that we are able to reuse several intermediate values that are computed within the `k`-loop of Listing 1. The disadvantage is that this implementation introduces twelve additional registers, bringing the total to 44. As a consequence we reduce the number of threads per block to 192 and push two registers into the global memory. In the following paragraphs we describe the optimizations that are responsible for our highest performing solution.

By examining Listing 1 we can see that there are three opportunities to reuse intermediate values throughout the `k`-loop. Specifically, the current and previous values of `mc`, `dc`, and `ip` can be maintained between loop iterations to reduce the number of global memory

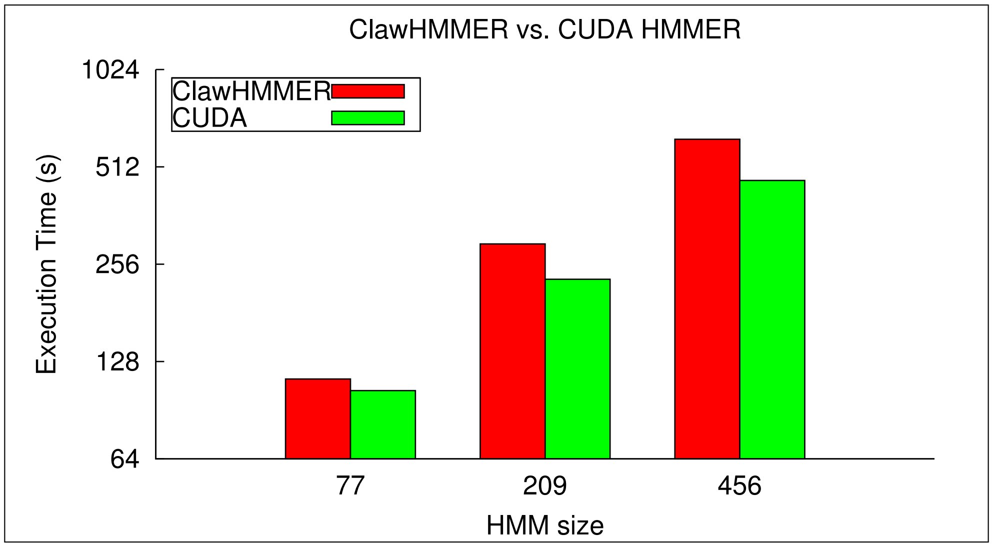

Figure 7. Performance comparison of our solution to the ClawHMMER implementation.

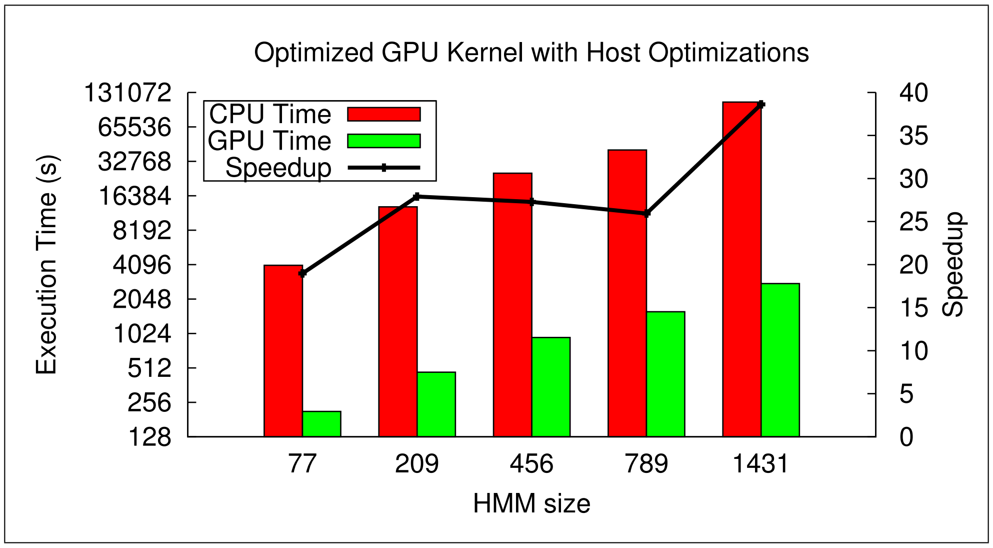

Figure 8. Performance evaluation of the final *hmmsearch* implementation including all host and GPU optimizations.

reads. This is crucial, as global memory reads represent the single greatest performance inhibitor. For the purposes of

In this final implementation we also include the use of the shared and constant memories. We note that the HMM stays constant throughout the entire computation and is used by each thread for each sequence. In most cases we can fit the entirety of the core transition matrices (`tpmm`, `tpim`, `tpdm`, etc.) as well as the `bp` matrix into the 64 KB constant memory. In cases where the size of the HMM exceeds the amount of constant memory, we utilize the full constant memory before switching over to texture memory for the remaining portions of the HMM.

Finally, we use shared memory to temporarily store our index into each thread's digitized sequence which is itself used as an index into the `ms` and `is` arrays from Listing 1. As a consequence, we are able to reduce the number of texture reads to two per iteration (4 if the loop is unrolled). In Figure 6 we present the results of our final GPU kernel. As Figure 6 shows, we are able to achieve between 12x and 37x speedup, depending on the size of the HMM. We note that the largest HMM (size 1431) runs for over one day before the completion (serial time). This results in a much higher speedup as the vast majority of the CUDA runtime is spent on the GPU. For the same reason, the 77 state HMM results in much lower speedup as more of its time is spent reading from the sequence database and post-processing. In fact, between post-processing, database reading, and DMA transfers to the GPU, the 77 state HMM spends twice as much time outside of the GPU kernel as within the GPU kernel.

To compensate for such overhead, particularly for small or average-sized HMMs, we performed several host-side optimizations. In particular, the serial overhead of reading the database and post-processing the database hits between GPU kernel invocations must be addressed. To do so we create two threads: the first for database reading, and the second to post-process database hits. We note that our GPU did not permit us to overlap DMA operations to the GPU during kernel executions. However, more recent hardware is capable of such optimizations.

In Figure 7 we compare our final implementation to the existing ClawHMMER GPU implementation by Horn et al. [7]. Because ClawHMMER runs within Windows XP, we were forced to use a smaller version of the `NR` database as well as smaller HMMs in order to stay within the Windows XP 2GB memory limit. Nevertheless, as we show, our implementation substantially outperforms the ClawHMMER implementation for every HMM. Moreover, as the size of the HMM increases, the performance of our CUDA implementation increases relative to the ClawHMMER implementation.

In Figure 8 we present our final performance results for the full `NR` database and a variety of HMMs of increasing size. Here the benefits of the host-side optimizations become clear - the 77 state HMM, for example, now achieves a performance of 19x, compared to the 12x previously achieved. Further, both the 209 state and 456 state HMMs also improved in performance, from 22.5x and 24.6x to 28x and 27x for the 209 and 456 state HMMs, respectively. The 789 and 1431 state HMMs improve slightly, from 24x to 26x, and from 37x to 38.6x. Their improvement was less dramatic as their runtimes dwarfed the serial portions of the computation.

6. Discussion

A comparison of the speedup from our GPU implementations of *MRF* (Figure 3) and *P7Viterbi/hmmsearch* (Figure 8) 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. While *hmmsearch* also reads global/texture memory a minimum of twice per iteration, it also writes 3 values to global memory for each iteration. 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 and the use of constant memory 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 more 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 exceptionally well-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 because its inner-loop is relatively free of branches with each thread operating on the same set of images simultaneously. While HMMER demonstrated exceptional speedup as well, it by necessity includes more branching than the *MRF* code and requires more I/O from global memory. This ultimately prevents the level of speedup achieved by *MRF*.

7. Conclusion

We have presented the performance of two statistics-based life science applications, *MRF*-based liver segmentation, and HMMER's *hmmsearch* database searching tool. Both applications demonstrated excellent performance improvement on the GPU, with *MRF* exhibiting a speedup of over **130x** compared to serial execution and *hmmsearch* outperforming all known HMMER GPU implementations with a speedup of **38.6x**. 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, proper shared and constant memory usage, 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] J. Chong, Y. Yi, A. Faria, N. Satish, and K. Keutzer. Data-Parallel Large Vocabulary Continuous Speech Recognition on Graphics Processors. In *Proceedings of the 1st Annual Workshop on Emerging Applications and Many Core Architecture (EAMA)*, pages 23–35, June 2008.

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

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

[7] 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*. IEEE Computer Society, 2005.

[8] 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.

[9] 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.

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

[11] 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.

[12] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-Sequence Database Scanning on a GPU. In *In 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006) (HICOMB Workshop)*. IEEE Computer Society, 2006.

[13] 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 (ICS 2006)*, pages 287–296. ACM, 2006.

[14] 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.

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

[16] 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.

[17] NCBI. The NR (non-redundant) database. `ftp://ftp.ncbi.nih.gov/blast/db/FASTA/nr.gz`, 2009.

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

[19] 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.

[20] 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*, volume 6512 of *(SPIE) Conference*, 2007.

[21] 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.

[22] 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. Springer-Verlag, 2001.

[23] TimeLogic BioComputing Solutions. DecypherHMM. `http://www.timelogic.com/`, 2009.

[24] 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.