ClawHMMER: A Streaming HMMer-Search Implementation

Daniel Reiter Horn Mike Houston Pat Hanrahan

Stanford University

Abstract

The proliferation of biological sequence data has motivated the need for an extremely fast probabilistic sequence search. One method for performing this search involves evaluating the Viterbi probability of a hidden Markov model (HMM) of a desired sequence family for each sequence in a protein database. However, one of the difficulties with current implementations is the time required to search large databases.

Many current and upcoming architectures offering large amounts of compute power are designed with data-parallel execution and streaming in mind. We present a streaming algorithm for evaluating an HMM's Viterbi probability and refine it for the specific HMM used in biological sequence search. We implement our streaming algorithm in the Brook language, allowing us to execute the algorithm on graphics processors. We demonstrate that this streaming algorithm on graphics processors can outperform available CPU implementations. We also demonstrate this implementation running on a 16 node graphics cluster.

Keywords: Bio Science, Data Parallel Computing, Stream Computing, Programmable Graphics Hardware, GPU Computing, Brook

1 Introduction

Biological sequence data are becoming both more plentiful and more accessible to researchers around the world. Specifically, rich databases of protein and DNA sequence data are available at no cost on the Internet. For instance, millions of proteins are available in the NCBI Non-redundant protein database [2005] and at the Universal Protein Resource [2005], and likewise the GenBank DNA database [NCB 2005] contains millions of genes. With this proliferation of data comes a large computational cost to query and reason about the relationships of a given family of sequences.

While a normal string compare is computationally simple, due to the randomness of evolution, proteins that share a purpose or structure may contain different amino-acid sequences, perhaps sharing a common sequence pattern. BLAST [Altschul et al. 1990] uses dynamic programming to perform a fuzzy string match between two proteins, penalizing gaps in the match to evaluate a sequence similarity score. However, in practice, BLAST queries must be run several times for an operator to identify suitable gap-penalty values and get an appropriate number of hits for the task at hand.

To mitigate the problem of choosing an ad-hoc gap penalty for a given BLAST search, Krogh et al. [1994] proposed bringing the probabilistic techniques of hidden Markov models(HMMs) to bear on the problem of fuzzy protein sequence matching. HMMer [Eddy 2003a] is an open source implementation of hidden Markov algorithms for use with protein databases. One of the more widely used algorithms, *hmmsearch*, works as follows: a user provides an HMM modeling a desired protein family and hmmsearch processes each protein sequence in a large database, evaluating the probability that the most likely path through the query HMM could generate that database protein sequence. This search requires a computationally intensive procedure, known as the Viterbi [1967; 1973] algorithm. The search could take hours or even days depending on the size of the database, query model, and the processor used.

However, even given the lengthy execution time required to search a database, *hmmsearch* is widely used in the biology research community. For instance, Narukawa and Kadowaki [2004] built a model of a typical trans-membrane section of a protein and used *hmmsearch* to search for proteins that contain this trans-membrane domain. Staub et al. [2001] trained a model with a few significant Spin/SSTY protein homologues that were repeated, hence important, in vertebrates and used this model to find similar amino-acid sequences in the non-redundant database. Clark and Berg [1998] leveraged their knowledge of the function of transcription factor protein TRA-1 as controlling sexual development in the *C-Elegans* worm and used hmmsearch to identify genes in the *C-Elegans* genome that potentially bind with TRA-1. Additional applications of *hmmsearch* to molecular and cell biology can be found in [Fawcett et al. 2000; Sánchez-Pulido et al. 2004; Bhaya et al. 2002].

The utility of HMMer in biological research and the unwieldy query run times in its normal usage make *hmmsearch* an important candidate for acceleration. There has been a great deal of work on optimizing HMMer for traditional CPUs [Lindahl 2005; Cofer and SGI. 2002]. However, there is a new class of *streaming* processors currently available and forthcoming that require different optimization strategies. For example, modern graphics processors, GPUs, have been shown to provide very attractive compute and bandwidth resources [Buck 2005].

In this paper we present ClawHMMer, a streaming implementation of *hmmsearch*, running on commodity graphics processors and parallel rendering clusters. We discuss the transformation of the traditional algorithm into a streaming algorithm and explore optimizing the algorithm for GPUs and future streaming processors. On the latest GPUs, our streaming implementation is on average three times as fast as a heavily optimized PowerPC G5 implementation and twenty-five times as fast as the standard Intel P4 implementation.

2 Streaming Architectures

Traditional CPU programming relies on optimizing sequential code and cache coherence for good performance. Traditional architectures rely on careful data layout and use caches to amplify bandwidth, take advantage of data locality, and reduce memory latency. The addition of 4-way SIMD instructions on CPUs, like SSE2 and AltiVec, have also allowed limited data-parallel operations, placing more requirements on caches for bandwidth and latency hiding to keep the compute units full.

In contrast, streaming architectures rely on large amounts of data parallelism to hide memory latency and provide compute performance. As a result, algorithms must be reformulated to be explicitly data-parallel. Data is expressed as *streams*, a collection of records requiring similar computation. Data parallel code is expressed as *kernels*, functions applied to each element of a stream, allowing for many compute units to run simultaneously. A streaming processor executes a kernel over all elements of an input stream, placing the results into an output stream. Instead of carefully blocking data into caches, data is *streamed* into the compute units. Data movement and dependencies are explicit in this model. As stream elements are being processed, the next elements are being fetched. This inherently changes the programming and optimization strategies to require explicit data parallelism.

There are many examples of stream processors. The Imagine stream processor [Kapasi et al. 2002] showcased streaming on a number of media processing applications. The Merrimac project [Dally et al. 2003] is researching the design of high-performance, low-power, streaming processors and systems containing large numbers of such processors. IBM recently announced the Cell processor, which has many of the attributes of a streaming processor [Flachs et al. 2005]. Merrimac, Cell, and other streaming processors are currently not available; however, we do have access to chips that resemble these processors: commodity programmable graphics processors. While normally relegated to application-specific multimedia processing, these processors can be treated as streaming processors and are readily available [Buck et al. 2004b].

Graphics hardware has been shown to provide good performance on streaming applications performing the same operation over large collections of data such as image segmentation and vector multiplication as well as applications that have sufficient parallelism and computational intensity to hide memory latency [Buck et al. 2004b]. Modern programmable graphics accelerators such as the ATI X800 series and the NVIDIA GeForce 7800 series [ATI 2004; NVIDIA 2005] feature programmable fragment processors and floating point support, making them available targets for streaming computation. Each fragment processor executes a short user-specified assembly-level shader program consisting of 4-way SIMD instructions over a dataset, enabling it to be used for generic processing [Lindholm et al. 2001].

Since graphics processors are widely available streaming processors with the capability to provide increased performance over traditional CPUs, this paper primarily concentrates on the design of our streaming HMMer algorithm and its implementation running on GPUs.

3 Algorithm

A hidden Markov model (HMM) is a statistical model of a Markov process, where the physical state of the system is unobservable. However, observable events may be produced by the system at a given physical state with a known probability distribution, and the system itself can transition between any two states with a known probability after each observable event. A tutorial on HMMs is provided by Rabiner [1989]. In this application of HMMs to structural biology, observations are limited to the sequence of amino acids in a protein string.

3.1 The Forward Algorithm

To explain the Viterbi algorithm we begin by describing the idea of the Forward algorithm, which, given an HMM with M states and a sequence of observations with length L, computes the probability that the HMM generates the given sequence via any path through it. The Forward algorithm starts with a vector of per-state probabilities indicating the probability that the model is at each state during any given observation. Before the first observation the per-state probabilities are set to zero except for the start state, which receives probability one. Then one observation, A_i, is read, and for each state s, the probability of that state generating A_i is multiplied by the sum of the chance of being at each possible predecessor, t, of s before having read A_i and then moving to s over a transition with likelihood $p_{trans}(t, s)$. The function from state to state of transition probabilities, p_{trans} can be represented as an MxM transition probability matrix.

$$p_{state_i}(s) = p_{emit}(s, A_i) \sum_{t \in states} p_{state_{i-1}}(t) \cdot p_{trans}(t, s)$$

This is repeated for each element in the string. To compute the probability of the HMM generating the input string and arriving at any of its states, a final sum must be performed over the probabilities at each of the HMM's states.

In pseudocode, the Forward algorithm appears as follows:

```
prob[0][0..length]=0, prob[0][startState]=1
for i = 1 to length(observ): //observation loop
    for s in states:         //state loop
        for t in states: //transition loop
            tmp=p_transition(t,s)·prob[i-1][t]
            prob[i][s]=prob[i][s]+tmp·p_emit(s, observ[i])
totalProbability=0
for s in states:
    totalProbability=totalProbability+prob[length(observ)][s]
```

This algorithm is exactly L repeated matrix-vector multiplications. The vectors are the per-state probabilities, and the matrix is the transition matrix with the rows individually being scaled by the value of a per-state emission probability for the current observation. Since each observation requires $(M+1)\cdot M$ multiplies and M^2 adds, the input string requires $O(M^2 \cdot L)$ operations to process.

3.2 The Viterbi Algorithm

In contrast to the forward algorithm, which computes the likelihood of a HMM generating an observation sequence, the Viterbi algorithm finds the most likely path through the HMM given an observation sequence and the probability of that path both occurring and producing that sequence. To compute this probability, the same steps are performed as the forward algorithm with a max instead of a sum over all elements.

$$p_{state_i}(s) = p_{emit}(s, A_i) \max_{t \in states} (p_{state_{i-1}}(t) \cdot p_{trans}(t, s))$$

The most likely final state in the HMM is the state with the highest probability after the final observation. This can be used to find the most likely predecessor by reexamining the predecessor state probabilities multiplied by their transitions and selecting the highest one. When performed all the way back through the observation sequence, this procedure is known as *traceback*, and it establishes the entire most likely path.

We illustrate the procedure below in pseudocode that computes the Viterbi probabilities of a given observation string and then performs a traceback to find the most likely path through it.

```
for i = 1 to length(observ): //observation loop
    for s in states:         //state loop
        for t in states: //transition loop
            tmp=p_transition(t,s)·prob[i-1][t]
            prob[i][s]=max(prob[i][s],tmp·p_emit(s, observ[i]))
call traceback(prob[][])
```

3.3 Streaming over an observation database

While CPU implementations focus on optimizing the cost of a single query, they do not take advantage of the data parallelism available when searching many observation sequences to optimize across queries. We rewrite the previous pseudocode using parallel_for semantics in order to explore the possibilities for exposing data parallelism:

```
parallel_for observ in database: //database loop
  for i = 1 to length(observ): //observation loop
    parallel_for s in states: //state loop
      for t in states:       //transition loop
        tmp=p_transition(t,s)·prob[i-1][t]
        prob[i][s]=max(prob[i][s],tmp·p_emit(s, observ[i]))
  call traceback(prob[][])
```

Instead of evaluating the search on one database protein at a time, we evaluate all database proteins at once, bringing the database loop within the state loop. This transforms the sequential algorithm into a data-parallel algorithm suited to an extremely wide SIMD processor.

```
allProb[]=per-state probabilities for each database entry
for i = 1 to length(longest observation): //observation loop
  parallel_for s in states:               //state loop
    parallel_for observ in database: //database loop
      for t in states:                 //transition loop
        prob=allProb[observ]
        tmp=p_transition(t,s)·prob[i-1][t]
        prob[i][s]=max(prob[i][s],tmp·p_emit(s, observ[i]))
for observ in database:
  call traceback(allProb[observ][][])
```

If the transition matrix is known to be sparse, computation can be avoided on transitions with no weight assigned to them. For instance if a transition matrix is compressed into an average of k nonzero elements per row, the asymptotic running time of Viterbi can be tightened to $O(k \cdot M \cdot L)$.

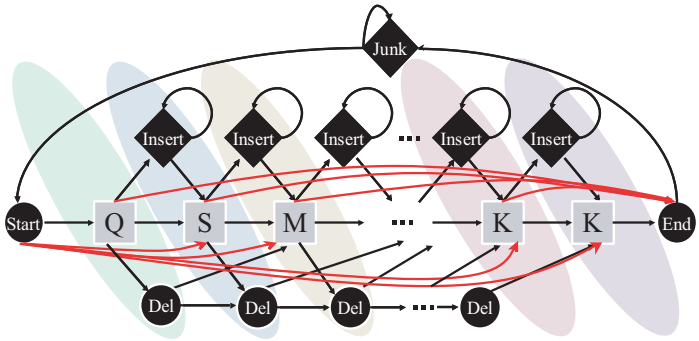


Figure 1: The special hidden Markov model used in HMMer. The gray states indicate important amino acids in the query model and are considered likely match states. Circular states emit no amino acids, and diamonds represent junk and insert states that emit unimportant amino acids in the query model. All insert states appear between two match states, but the junk state handles the large numbers of potentially unrelated amino acids between one complete match chain and another. Shaded ovals denote state triplets to be processed at once.

3.4 Utilizing the specifics of the HMMer HMM

In the HMMer application, a very specialized HMM is used to compute the Viterbi probability. This HMM models a protein family, so there is a core chain of states that represents important amino acids in this family. These states, known as *match states*, drive the HMM's progression from start to end states, making the HMM likely to produce proteins contained in the family. On the way through the chain of match states the HMM may generate meaningless amino acids by transitioning to an insert state.

To model a protein family, the HMM should be able to skip over match states. Normally this would entail a transition from any match state to any other match state further down the chain. However, this would introduce a number of transitions proportional to the square of the number of states. Instead HMMer introduces a new type of state not present in a general HMM, known as a delete state, which does not produce an observation. Each match state can link to a delete state chain which effectively allows the HMM to skip amino acids, by transitioning to the delete state chain.

This HMM has the property that the all states but one have four or fewer transitions into them, and only a single state, the end state, has a number of incoming transitions proportional to the number of states in the HMM, as illustrated in Figure 1. Thus, the transition matrix is not only sparse, but k can be bounded by four, setting the running time proportional to $O(M \cdot L)$. Likewise, it becomes possible to unroll the transition loop to only process incoming transitions with nonzero probability in the *max* step of the Viterbi algorithm.

Adding a new delete state type, however, is not without cost: it introduces a loop-carry dependency into the state loop. This occurs because the probability of being at a delete state depends on the probability of being at its predecessor delete state at the same index in the observation sequence.

prob[i][deleteState[j]] depends on prob[i][deleteState[j-1]]

The dependency eliminates an entire axis of parallelism that might have been exploited for a more general HMM;

however, because the database contains a large number of proteins, the algorithm remains data-parallel.

Our inner loop, which will become our kernel, unrolls the state loop 3 times, processing 3 states at once: a match state, its following delete state and its previous insert state, as illustrated in Figure 1. We denote this group of states a *state triplet*. The state loop should be unrolled because each state in a state triplet shares a number of predecessor states with the others. The probability for the end state, the only state with a number of transitions proportional to the HMM size, is also accumulated in the same loop.

Additionally *hmmsearch* only requires a traceback if the protein itself has a high-probability. Thus, for the majority of cases only the two most recent per-state probabilities vectors are needed. We utilize this property by bouncing between two per-state probability arrays, using the observation index mod 2 to calculate which array to use as output.

The new arrangement of the code, with the unrolled loop, is illustrated in the following pseudocode:

```
for i = 1 to length(longest protein):      //observation loop
  for s in greyMatchStates:                //state loop
    j = previousInsertState(s)
    q = previousDeleteState(s), d = nextDeleteState(s)
    parallel_for protein in database: //database loop
      curProb=allProb[protein][i mod 2]
      prevProb=allProb[protein][(i + 1) mod 2]
      //transition loop (unrolled)
      tmp_s=max(s's 4 incoming trans. in prevProb)
      curProb[s]=tmp_s
      curProb[d]=max(tmp_s · p_trans(),curProb[q]·p_trans())
      curProb[j]=max(j's 2 incoming trans. in prevProb)
      curProb[end]=max(prevProb[end],tmp_s·p_trans())
for protein in database:
  if (allProb[protein][last][endstate]>globalThreshold):
    call generalViterbi(protein) for traceback
```

4 Implementation

4.1 Brook

BrookGPU provides a language and runtime system to use GPUs as streaming processors [Buck et al. 2004b]. Data is represented as finite streams and the computation is expressed as kernels. Every element in a stream is assumed to be independent, so each data element can be operated upon in parallel. On the GPU, streams are stored in texture memory and kernels are compiled to fragment programs. Geometry is then sent to the GPU which generates fragments for each element in the output stream. Each fragment invokes the kernel, reading from the input streams and writing to the output streams. Although BrookGPU abstracts the GPU as a streaming architecture, stream size is limited to available texture memory, and kernels are limited by the number of instructions, registers, texture reads, and outputs allowed by the hardware.

4.2 HMMer on the GPU

Graphics hardware has a very fast memory subsystem of limited size. For large databases, we divide the database into batches upon which we execute our algorithm so that the batches will fit in graphics memory alongside all the temporary state probabilities. To simplify the database division, we pre-sort the database so similar length proteins reside together. This causes the sequences within a batch to require similar amounts of computation. We stream batches of database sequences onto the GPU as others are being processed. We save state probabilities in 4-channel textures, one texture per state triplet in the HMM, and we encode the running-max operation for the end state in the last channel. Thus, we must have one float4 texture per state triplet in the HMM, which enforces an upper bound on the database batch size we can work on while fitting in GPU memory.

Each kernel computes the unrolled transition loop of the Viterbi process as illustrated in Section 3.4, performing a max over all incoming transitions and accumulating the end state probability. The transition probabilities are loaded into constant registers which can be accessed quickly by each fragment processor. The emission probability table is stored as a 2x24 table in texture memory, representing the 24 amino acids in the match and insert states respectively.

Because the origin of the nonzero transitions for a given state triplet are known before the kernel is run, the appropriate textures needed for the shader to process a particular state triplet are located and are presented to the GPU as input through the texture units. The program also requires a texture of bytes, representing amino acids at the same location in respective proteins in the batch. The kernel uses this amino-acid texture to do lookups on the emission probability table for each of the match and insert state emissions. The state triplet probabilities, along with the current probability value of the end state, are output by the kernel as a 4-channel floating point texture.

The kernel is then run over the length of the database batch, resulting in the execution of the parallel_for in our HMMer-specialized Viterbi algorithm. This is repeated in series for each character in the longest protein for each state in the HMM. A pictorial example of the execution is provided in Figure 2.

4.3 Optimization

Lindahl [2005] illustrated the benefit of vectorizing the state loop to process 24 HMM states in SIMD fashion to produce 24 intermediate state probabilities in the form of 8 state triplets at once. On a CPU this reduces the number of math ops that must be performed on the values because the same number of instructions can operate on nearly four times as many elements. It also helps the inner loop hide latency from memory fetches by interleaving prefetch commands with math for loading values. On a GPU, memory latency is not as much of an issue because the hardware hides memory latency with computation on other elements. The entirety of Lindahl's approach is not possible because of the constraints of the GPUs memory system: we can only output a maximum of 16 floating point values at once, the memory subsystem only allows very constrained output patterns of four 4-vectors, and we only have access to a small number of registers to hold the intermediate state probabilities.

The concept of calculating multiple neighboring states triplets in a single kernel also reduces the bandwidth requirements of the algorithm. On the GPU, each kernel invocation has to read input values, operate on them and write output values. Since the delete state depends directly on the previously produced values and there exists no way to pass intermediate results directly to a subsequent kernel invocation, many memory reads could be avoided by directly using a previously computed delete state probability to calculate the next delete state probability within the same kernel. Likewise the insert and match state portions of neighboring

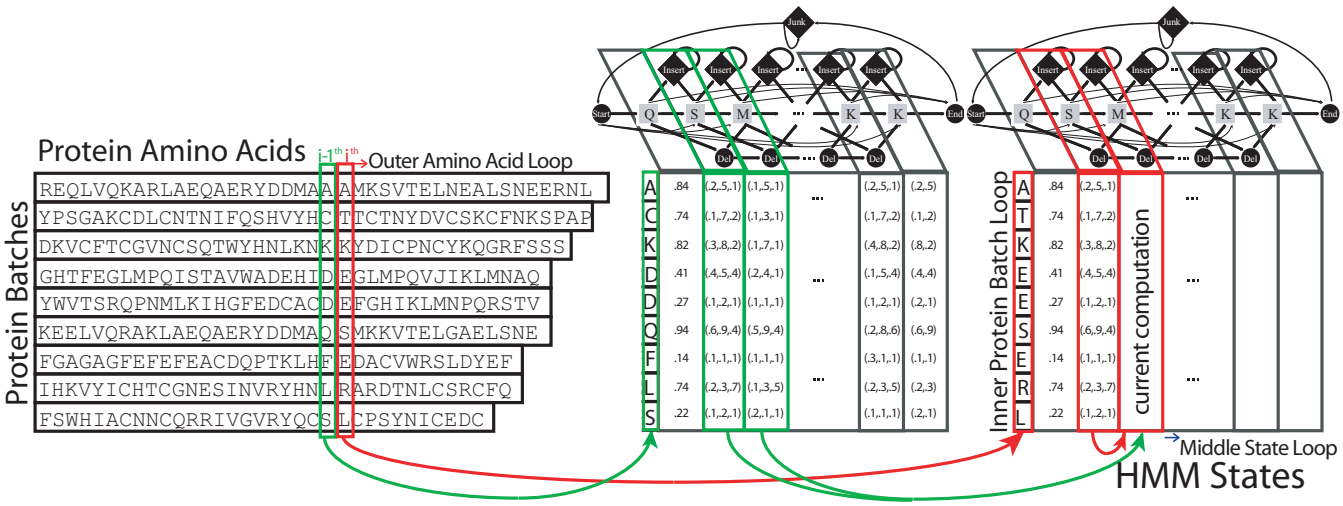


Figure 2: The outer loop of the GPU Viterbi process loops over one amino acid observation from each of many protein batches. The inner loop uses these single amino acids, and the previously computed probabilities from the current and previous state arrays, to compute probabilities of the most likely state at the current observation being the current state

Kernel	1-output	2-output	4-output
Input (bytes)	69	89	145
Output (bytes)	16	32	64
Ops (ARB/ps2b)	32	55	93
Calls (billions)	111.4	56.1	28.4
NVIDIA 6800 Ultra	943.3s	749.8s	643.1s
NVIDIA 7800GTX	475.8s	388.1s	361.4s
ATI X800XTPE	446.0s	348.6s	301.0s

Table 1: Cost of kernel calls in HMMer. Instructions are listed in in number of ARB/ps2b statements. Number of calls is measured from querying the Adenovirus model on the NCBI Non-redundant database.

state triplets share predecessors in the state graph and can reuse probabilities from the previous observation when iterating over the incoming transitions. Calculating many state triplets at once allows for some producer-consumer locality between the calculations to be exploited. However, because we are limited to sequential writes of 16 floats, we can only process four state triplets simultaneously on current GPUs.

Our original kernel, which processes one state triplet at a time, reads 69 bytes and writes 16, issuing 32 ARB/ps2b instructions. Our 2-output kernel processes 2 state triplets per invocation inputs 89 bytes and writes 32, while issuing 55 instructions. This kernel is executed half as many times as the original kernel, and hence does less than twice as much total work. Our 4-output kernel processes 4 state triplets at once, thus operating on 12 states simultaneously, half as many as in the AltiVec loop [Lindahl 2005]. We read 145 bytes and write the maximum allowable number on graphics hardware, 64. This kernel requires 93 instructions to execute, and is run a fourth as many times as the original kernel that only processed a single state triplet. This is summarized in Table 1, and we analyze actual performance results of each version in Section 5.

4.4 HMMer on a Cluster

Similar to SledgeHMMer [2004], we distribute the protein database among the nodes in the cluster and issue the

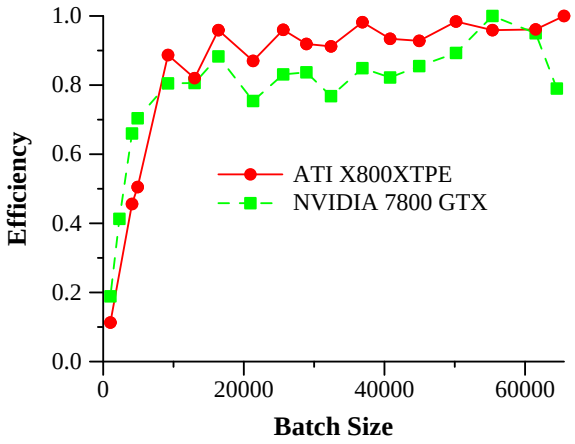


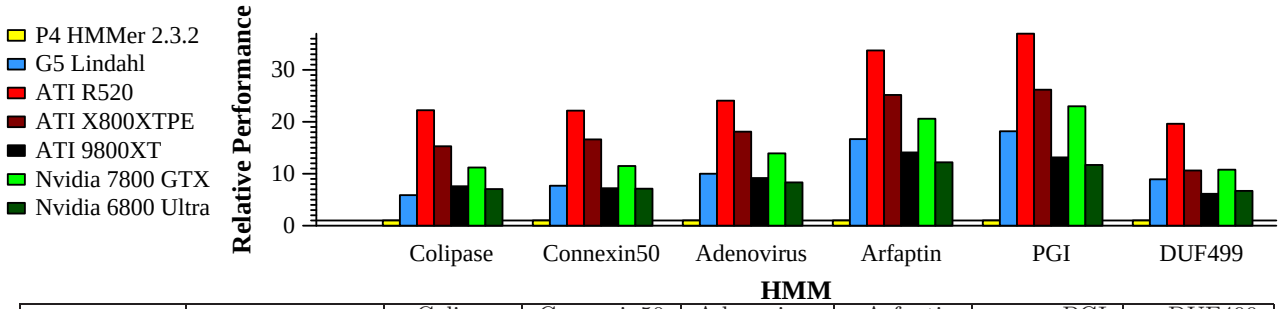
Figure 3: The efficiency of GPUs using different batch sizes

HMM query to each node. After initially sorting the protein database by protein length, we give each node approximately the same amount of total work. We first divide the database into partitions of the same total size in bytes, rounded to the nearest batch size, so that some partitions will receive many short sequences and other partitions will receive few long sequences. We choose sets of nodes to process each partition. Then for each set we interleave the partition between individual nodes.

We cannot maintain a perfect balance since we must respect large batch sizes, and interleaving the databases results in the batches becoming unbalanced towards the end of a query string, reducing effective batch size and hence performance. We investigate scalability more in Section 5.3.

5 Results

For our performance tests, we compare our streaming version of HMMer against the publicly available HMMer 2.3.2 running on a 2.8GHz Intel Pentium4 Xeon and a 2.5GHz PowerPC G5 (Altivec path), as well as a highly tuned Altivec



Architecture	Best Codepath	Colipase 139 states Time(s)	Connexin50 211 states Time(s)	Adenovirus 427 states Time(s)	Arfaptin 739 states Time(s)	PGI 1543 states Time(s)	DUF499 3271 states Time(s)
P4 2.8 GHz	HMMer 2.3.2	1589.1	2372.8	5030.6	12236.9	28423.6	29601.7
G5 2.5 GHz	HMMer 2.3.2	539.9	729.0	1300.5	2229.1	4547.9	11778.4
G5 2.5 GHz	Lindahl	279.4	320.8	546.0	836.5	1574.9	3341.6
R520	Brook 4-output	71.48	107.1	209.0	362.6	768.9	1508.4
X800XTPE	Brook 4-output	107.5	148.5	301.0	555.5	1088.9	2800.6
9800XT	Brook 2-output	217.2	324.2	594.4	988.7	2175.9	4861.3
7800 GTX	Brook 4-output	141.9	206.4	361.4	594.3	1236.1	2747.2
6800 Ultra	Brook 4-output	233.1	330.4	643.1	1142.8	2451.0	4461.0

Table 2: Performance results from scoring the 2.4 million proteins from the NCBI Non-redundant protein database against HMMs trained from the Colipase C-terminal domains, Connexin50 Gap junction α-8, Adenovirus GP19K, an Arfaptin-like domain, Phosphoglucose isomerase and Domain of Unknown Function #499 in the pfam HMM database.

version from Erik Lindahl. We demonstrate the algorithm running on a ATI 9800XT and X800XTPE and a NVIDIA 6800 Ultra and 7800GTX, each with 256MB of graphics memory, as well as a prerelease ATI R520 (600MHz core, 700MHz memory) with 512MB of graphics memory. Table 2 shows the performance of each architecture. As can be seen, the ATI X800XTPE and R520 and the NVIDIA 7800GTX outperform the standard versions of HMMer on both the P4 and G5 as well as the highly tuned Altivec version.

5.1 Batch Sizes

Because many GPUs have only 256MB of on board memory and each state of the HMM requires additional temporary storage, larger HMMs necessitate suboptimal batch sizes. For example with Domain of Unknown Function #499 (DUF 499), we must reduce the batch size from 36864(192x192) to 9216(96x96) to fit in 256MB of memory. This causes an overall decrease in performance since smaller batch sizes decrease the efficiency of the GPU as can be seen in Figure 5. On an ATI X800XT we obtain almost 90% of peak performance at a batch size of 9216 (96x96), but the X800XT loses performance rapidly below this mark. The NVIDIA 7800 GTX maintains 70% of peak performance for batches as small as 4900 (70x70) but requires batch sizes of at least 16384(128x128) to achieve 90% of peak performance. However, both ATI and NVIDIA offer 512MB boards which can support very long HMMs without decreasing the batch size below 16384. For instance, on an 512MB ATI R520, the DUF 499 HMM takes 23.2% less time than on the 256MB version of the same board, 1508.4s instead of 1962.3s.

5.2 Performance Model

GPUBench [Buck et al. 2004a] provides a way to model the performance characteristics of GPUs. We can run simple tests on each of our GPUs to determine the maximum bandwidth for different access patterns, the peak FLOPS rate, and information about latency hiding. For example, from GPUBench, we know that the theoretical streaming input bandwidth on an ATI X800XTPE is 17.5GB/s and it takes 7 kernel instructions to hide a float4 texture fetch. The NVIDIA 6800 Ultra can stream a maximum of 9.5GB/s while the NVIDIA 7800GTX can stream 19GB/s, but it takes 4 instructions, which cannot be hidden, to issue a float4 fetch. Output costs for both vendor's hardware seem to be hidden when large enough batch sizes are used.

Since the ATI X800XTPE can hide memory latency provided enough instructions, we expect the total cycle time of a kernel to be the maximum of the instruction count and the number of clock cycles required to access memory. For example, for the single-output version of the code, we read 69 bytes, output 16 bytes, and issue 32 instructions. Since we can issue one instruction per clock cycle, we expect the cycle count from instructions alone to be 32 as written, but the fragment program is optimized by the graphics driver to around 27 cycles. We base our estimates on the driver's ability to optimize the fragment program by replacing texture fetches with fetching values from constant registers, requiring no input bandwidth. Since all memory fetches are streaming fetches, we estimate that 31 cycles are required to access memory. Using the max of memory and instruction cycles, we expect each kernel invocation to take 31 cycles. From the time and the number of kernel invocations required for computation, shown in Table 1, each invocation requires 33 cycles. This give us 16.1GB/s of input bandwidth, or 92% of peak streaming input bandwidth. The 2 output version of the algorithm is invoked half as many times as the single output version, but does more work per invocation. The goal of the 2-output version is to reduce the total amount of data transfered. Bandwidth requirements have been reduced enough for the application to become limited by instruction issue. We expect 40 cycles of memory access, 55 cycles of instruction issue, optimized to around 48 instructions by the driver, and we observe the kernel requires 51 cycles per invocation. The board is achieving 94% of peak instruction issue

rate and a streaming input bandwidth of 13.1GB/s. The 4-output version continues this trend, and further reduces the bandwidth usage and is also able to take better advantage of the SIMD ALUs on the GPU. With this version, we expect 64 cycles of memory access, 93 instructions, optimized to around 80 by the driver, and we observe 87 cycles per invocation. Hence the hardware is operating at 92% of peak instruction issue rate and at a streaming input bandwidth of 12.9GB/s. Similar conclusions can be drawn from the next generation R520 and the previous generation 9800 hardware when clock rates and pipe counts are taken into account, except that the 9800 is limited to 64 ARB/ps2 instructions precluding the use of the 4-output implementation.

For NVIDIA hardware, all texture latency can be hidden, but it takes 4 cycles to fetch a float4. Including all texture fetches, we expect 49 cycles for a single-output kernel. Given the clock rate and execution time, the kernel actually requires 60 cycles to complete. Thus even though GPUBench predicts a maximum bandwidth of 9.5GB/s for the 6800 Ultra and 19GB/s for the 7800GTX, the application exhibits 7.59GB/s and 13.5GB/s respectively. However, unlike the ATI hardware, the NVIDIA hardware cannot directly use previous outputs as inputs for future iterations without a copy. When factoring in the extra data copy per iteration, the total bandwidth is 9.34GB/s for the 6800 Ultra and 18.5GB/s for the 7800GTX, or 97% the peak bandwidth on both boards. Using this analysis we obtain 8.44GB/s (88%) and 16.3GB/s (85%) for the 2-output kernel and 8.59GB/s (90%) and 15.3GB/s (80%) for the 4-output kernel.

5.3 Cluster Scalability

We demonstrate the scalability of ClawHMMer on SPIRE, a 16-node parallel rendering cluster with a Radeon 9800 Pro GPU in each machine. For the scalability test, the sorted 2.4 million, 1.06GB NCBI Non-redundant database was divided as evenly as possible between the active nodes. We first divide the database into even pieces, giving roughly equal amounts of total work, and then interleave the proteins among nodes, giving roughly equal length proteins to each node to keep SIMD efficiency. For instance, for the 8 node case, the database was partitioned into two sorted 544.5MB pieces which were each interleaved on a per-protein basis across 4 nodes. This maintained batches with proteins of similar length while assuring that the 2 partitions were not uneven due to batch sizes. For the 16 node case, the best performance was demonstrated when the data was partitioned into 4 nearly-even chunks which were each interleaved across four nodes.

Figure 5.3 illustrates the scalability of our algorithm on the cluster. With up to 10 nodes we are able to partition the databases well and maintain both uniform batches and balanced database sizes. At 10 nodes, the cost of interleaving the database between nodes becomes noticeable. It becomes more difficult to assign the same amount of work to each node, causing a performance imbalance. However, on 16 nodes, the overall efficiency is still greater than 95%.

6 Discussion

6.1 Costs of Traceback

When performing queries over B sequences at a time, the memory usage of saving the per-state probabilities for each of M HMM states is $O(B \cdot M)$. However, saving the information needed for a traceback over a length L query amino-acid

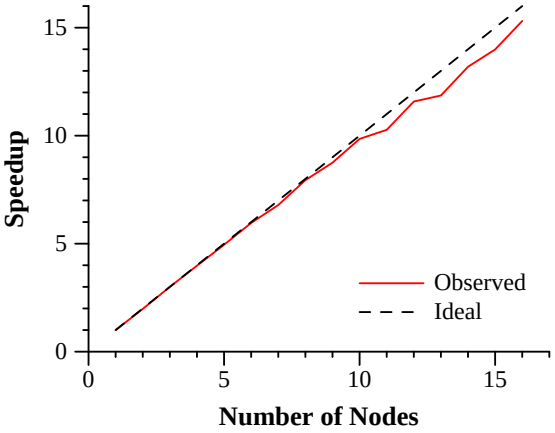


Figure 4: Scalability of distributing the NCBI Non-redundant database across a 16 node cluster with ATI 9800s. As expected, we scale almost linearly in performance with added nodes. As the number of nodes increases, it becomes difficult to evenly subdivide work among the nodes, which causes performance to decrease to 95% of ideal for 16 nodes.

requires $O(B \cdot M \cdot L)$ memory. This drastically decreases the maximum batch size we can use, lowering the efficiency of the processor. Thus, we can only efficiently apply the GPU as an early-reject phase in the *hmmsearch* utility, and we perform tracebacks for high probability matches on the CPU. This precludes efficiently running *hmmpfam*, the sister application to hmmsearch, because hmmpfam is often run with a special score correction step. The score correction requires a traceback to compensate for observations that are emitted by junk states as opposed to match states. Eddy [2003b] asserts that without score correction, pfam searches will have less specificity, yet more sensitivity to matches.

Future work includes investigating the possibility of limiting the traceback for score correction by saving only the list of characters that caused the start or end of an entire match state chain. This could be used to separate the Match or Insert states from the Junk state of the HMM and could potentially assist in approximating a score correction term.

6.2 Applying GPU's to General HMMs

Much of our analysis and implementation has relied on the specific HMM layout when performing a HMMer protein search to achieve good performance. However the Viterbi process and other techniques involving hidden Markov models are common in a variety of domains including speech recognition, voice printing, economics, and AI. These domains usually employ more general HMMs with fewer restrictions on transition layouts. For instance, the possibility of a transition from any HMM state to any other state obviates the need for the delete states which emit no observable characters, exposing an additional axis of parallelism over the states themselves. This parallelism reduces the number of batches needed for a GPU to run at peak efficiency, which in turn allows for much larger HMM's to fit into a GPU's onboard memory. The Viterbi algorithm for general HMMs is very similar to iterated matrix vector multiplication as explained in Section 3.

The performance of GPU's on this more general application will vary with HMM size. The transition matrix for small HMM's might completely fit into the cache of a CPU, allowing for very efficient use of producer-consumer local-

ity and resulting in compute-limited performance. On the GPU, we cannot take advantage of producer-consumer locality, so the performance will be limited by the memory system, which is currently competitive with the CPU cache. However, as GPUs increase compute performance, it may be difficult for the memory subsystems to keep pace. In contrast, large HMMs might not fit in a CPU cache forcing the CPU's main memory system to compete with the GPU's higher performance memory system. The GPU has memory which is designed for streaming multimedia applications, making Viterbi on large, general HMM's a potentially effective algorithm on a GPU.

6.3 Limited Data Reuse on the GPU

When the optimized HMMer AltiVec code computes the probabilities of a particular amino acid being emitted from each of the states, the previous probability array fits in the G5's 64GB/s cache. This gives fast access to more states, allowing the inner loop to be unrolled, which in turn increases the ability to use more SIMD math operations. Operating over more states at once can help exploit producer-consumer locality and instruction-level parallelism. On the GPU, loop unrolling is accomplished by fusing kernels together, creating a larger kernel representing multiple iterations of the loop. The amount of kernel fusion is limited by the amount of data that may be output. Furthermore, since all writes must commit to off-chip memory, we do not have fast access to previously computed values, limiting our ability to use producer-consumer locality across kernel invocations. However, even with the limited loop unrolling available to current GPUs, we can create instruction-issue bound applications on ATI hardware.

For many algorithms to achieve good performance on CPUs, clever blocking strategies are employed to ensure that the data to be reused resides in the cache for the majority of its use. The fundamental idea is to fit the working set within the fastest memory, often a high speed cache. This property allows both the HMM add-max inner-loop as well as the matrix-vector multiply-add inner-loop to be executed from a cached value to a cached value, and since the cache runs at or near clock rate, the math units remain busy, constantly processing new values from the cache.

The GPU, on the other hand, supports random read-only access from a much larger array of texture memory, compared to most CPU caches. The latency from the memory system is hidden by the sheer number of items being processed in parallel along with computation that can be overlapped with communication. Thus, as Fatahalian et al. [2004] argued, to run at peak efficiency, all the math units would need to have work to process during the entire duration of the memory fetch. For an algorithm like Matrix Multiply, which involves a single multiply-add per pair of memory reads, or even the more math-intensive Viterbi algorithm, which requires a trio of add and max operations, the application is entirely limited by the speed of the texture memory system. However the larger, but slower, GPU memory subsystem must compete with a much faster, but much smaller, CPU cache. While trends have shown that graphics hardware computes at rates that continue to increase at a rate that dwarfs competing hardware, this algorithm, like Matrix Multiply, will continue to be dominated by the texture memory performance on graphics processors unless caches or scratch-pads are added.

6.4 HMMsearch on future architectures

As GPUs get faster, it is unclear if the texture memory system will keep pace with the available compute performance. Without faster memory access, either from the texture memory system, caches, or scratch-pads, memory performance will begin to dominate the cost of many algorithms. However, architectures that provide fast local memory systems can benefit from the design considerations of a streaming formulation of HMMer. In order to maximize throughput, our streaming algorithm saves enough state probabilities to fill the closest memory system, which on the GPU is a large texture memory. For a larger HMM, like the 3271 state DUF499, to remain entirely in 256 MB of texture memory we decrease the batch size from 36,864 to 9,216, trading efficiency for a working set that avoids slow access to host memory.

Thus, on future parallel architectures with fast local stores like the Cell processor, a similar batching mechanism should be employed to fill the memory system and each 256KB local store [Dhong et al. 2005]. Yet, since the size of the local store is much smaller than the texture memory on the GPU, an optimized version of HMMer on Cell would most likely run on the order of 10's of protein strings per processing element instead of the much larger batches used on the GPU. However, because we do not have the output limitations as we do on the GPU, we can perform more kernel fusion in order to hide memory latency, exploit SIMD units more effectively, and reduce the total bandwidth requirements.

While batching the queries is a necessity on graphics hardware, it also mitigates the repeated lookup of numerous transition probabilities, favoring their storage in instruction code as constants until the amino-acid at the current position for the whole batch has been processed.

For these reasons, our streaming implementation on GPUs can be viewed as a prototype for the next generation of parallel architectures including the Cell processor, future GPUs, and Intel's projected many-core [Borkar et al. 2005] architectures.

7 Conclusion

We have presented a streaming version of the Viterbi algorithm and demonstrated its implementation on GPUs. Our implementation running on GPUs outperforms currently available CPU implementations, including heavily optimized, hand-tuned ones. Our algorithm performs well on current graphics hardware and remains viable on varying parallel and streaming architectures. We have performed an analysis that demonstrates the bottlenecks of our algorithm and the efficiency of our implementation on current hardware.

We have demonstrated a cluster version of hmmsearch with linear scalability. A cluster of GPUs running our implementation is now a competitive alternative to traditional CPU clusters running HMMer. As GPUs continue to get faster, using GPUs and clusters of GPUs for tasks such as protein sequence matching becomes even more attractive.

We hope that as new streaming architectures and multicore CPUs become widely available, we can explore the efficiency of our algorithm as well as the many other applications currently written and targeted for GPUs.

8 Acknowledgments

We would like to thank Erik Lindahl for his prerelease source code and helpful optimization hints, and Sean Eddy for his tireless work on HMMer and careful responses to our queries. We would like to thank Jeff Golds, Raja Koduri, Gianpaolo Tommasi and Kai Tai at ATI and Nick Triantos and Ian Buck at NVIDIA for hardware donations, driver support and immense patience. This work is supported by the US Department of Energy (contract B527299).

References

Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. 1990. Basic local alignment search tool. *Journal of Molecular Biology 215*, 3 (October), 403–410.

ATI, 2004. Radeon X800 product site. http://www.ati.com/products/radeonx800.

Bhaya, D., Dufresne, A., Vaulot, D., and Grossman, A. 2002. Analysis of the *hli* gene family in marine and freshwater cyanobacteria. *FEMS Microbiology Letters 215*, 209–219.

Borkar, S. Y., Dubey, P., Kahn, K. C., Kuck, D. J., Mulder, H., Pawlowski, S. S., and Rattner, J. R. 2005. Platform 2015: Intel processor and platform evolution for the next decade. *Technology@Intel Magazine 3*, 3 (April).

Buck, I., Fatahalian, K., and Hanrahan, P. 2004. Gpubench: Evaluating gpu performance for numerical and scientific applications. In *Poster Session at GP2 Workshop on General Purpose Computing on Graphics Processors*. http://gpubench.sourceforge.net.

Buck, I., Foley, T., Horn, D., Sugerman, J., Hanrahan, P., Houston, M., and Fatahalian, K. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. In *Proceedings of the ACM SIGGRAPH 2004*.

Buck, I. 2005. Taking the plunge into GPU computing. In *GPU Gems 2: Programming Techniques for High Performance Graphics and General Purpose Computation*, M. Pharr, Ed. Addison Wesley, 880.

Chukkapalli, G., Guda, C., and Subramaniam, S. 2004. SledgeHMMER: a web server for batch searching the pfam database. *Nucleic Acids Research 32* (July), W542–544.

Clarke, N., and Berg, J. M. 1998. Zinc fingers in caenorhabditis elegans: Finding families and probing pathways. *Science 282* (December), 2018–2022.

Cofer, H., and SGI., 2002. HMMER on Silicon Graphics. http://sgi.com/industries/sciences/chembio/resources/hmmer .

Dally, W. J., Hanrahan, P., Erez, M., Knight, T. J., Labonté, F., Ahn, J.-H., Jayasena, N., Kapasi, U. J., Das, A., Gummaraju, J., and Buck, I. 2003. Merrimac: Supercomputing with Streams. In *Proceedings of SC2003*, ACM Press.

Dhong, S. H., Takahashi, O., White, M., Asano, T., Nakazato, T., Silberman, J., Kawasumi, A., and Yoshihara, H. 2005. A 4.8 GHz fully pipelined embedded SRAM in the streaming processor of a CELL processor. In *Proceedings of IEEE International Solid-state Circuits Conference*, 486–487,612.

Eddy, S., 2003. HMMER: Profile HMMs for protein sequence analysis. http://hmmer.wustl.edu.

Eddy, S. 2003. HMMER user's guide. *Howard Hughes Medical Institute and Dept. of Genetics, Washington University School of Medicine.* (October).

European Bioinformatics Institute, Swiss Institute of Bioinformatics, and Georgetown University, 2005. Universal protein resource. http://www.uniprot.org.

Fatahalian, K., Sugerman, J., and Hanrahan, P. 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of Graphics Hardware*, Eurographics Association.

Fawcett, P., Eichenberger, P., Losick, R., and Youngman, P. 2000. The transcriptional profile of early to middle sporulation in *Bacillus subtilis*. *Proceedings of the National Academy of Sciences USA 97*, 14 (July), 8063–8068.

Flachs, B., Asano, S., Dhong, S. H., Hofstee, P., Gervais, G., Kim, R., Le, T., Liu, P., Leenstra, J., Liberty, J., Michael, B., Oh, H., Mueller, S. M., Takahashi, O., Hatakeyama, A., Watanabe, Y., and Yano, N. 2005. A streaming processor unit for a CELL processor. In *Proceedings of IEEE International Solid-state Circuits Conference*, 134–135.

Forney, G. D. 1973. The Viterbi algorithm. *Proc. IEEE 61* (Mar.), 268–78.

Kapasi, U., Dally, W. J., Rixner, S., Owens, J. D., and Khailany, B. 2002. The Imagine Stream Processor. *Proceedings of International Conference on Computer Design* (September).

Krogh, A., Brown, M., Mian, S., Sjolander, K., and Haussler, D. 1994. Hidden markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology 235*, 1501–1531.

Lindahl, E., 2005. Altivec HMMer, version 2. http://lindahl.sbc.su.se/software/altivec/altivec-hmmer,-version-2.html.

Lindholm, E., Kilgard, M. J., and Moreton, H. 2001. A user-programmable vertex engine. In *Proceedings of SIGGRAPH 2001*, ACM Press/Addison-Wesley Publishing Co., 149–158.

Narukawa, K., and Kadowaki, T. 2004. Transmembrane regions prediction for G-protein-coupled receptors by hidden markov model. In *Proceedings of the 15th International Conference on Genome Informatics*, Universal Academy Press.

2005. National Center for Biotechnology Information. ftp://ncbi.nlm.nih.gov.

NVIDIA, 2005. GeForce 7800: Product overview. http://nvidia.com/page/geforce_7800.html.

Rabiner, L. R. 1989. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE 77*, 2 (Februrary).

Sánchez-Pulido, L., Rojas, A., van Wely, K., Martinez-A, C., and Valencia, A. 2004. Spoc: A widely distributed domain associated with cancer, apoptosis and transcription. *BMC Bioinformatics 5*, 1, 91.

Staub, E., Mennerich, D., and Rosenthal, A. 2001. The Spin/Ssty repeat: a new motif identified in proteins involved in vertebrate development from gamete to embryo. *Genome Biology 3*, 1, research0003.1–research0003.6.

Viterbi, A. J. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. on Information Theory 13*, 2, 260–269.