2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

A Speculative HMMER Search Implementation on GPU

Xiaoqiang Li, Wenting Han, Gu Liu, Hong An, Mu Xu, Wei Zhou, Qi Li
School of Computer Science and Technology
University of Science and Technology of China
Hefei, China
Key Laboratory of Computer System and Architecture
Chinese Academy of Sciences
Beijing, China
lixq520@mail.ustc.edu.cn, han@ustc.edu.cn, {gliu, han, xumu, greatzv, liqi1982}@mail.ustc.edu.cn

***Abstract*—Due to the exponentially growing bioinformatics databases and rapidly popular of GPU for general purpose computing, it is promising to employ GPU techniques to accelerate the sequence search process. Hmmsearch from HMMER bioinformatics software package is a wildly used software tool for sensitive profile HMM (Hidden Markov Model) searches of biological sequence databases. In this paper, we implement a speculative hmmsearch implementation on NVIDIA Fermi GPU and apply various optimizations to it. We test the enhancements in our GPU implementation in order to demonstrate the effectiveness of optimization strategies. Result shows that our speculative hmmsearch implementation achieves up to 6.5x speedup over previous fast single-threaded SSE implementation.**

***Keywords*-HMMER 3.0; GPU; memory optimization; speculative;**

I. Introduction

In recent years, the graphics processing units (GPUs) have become popular not only in traditional scientific computing domain but in more general purpose computing domains because of the tremendous computing power compared with traditional CPU and the being more mature GPU development tools such as NVIDIA's Compute Unified Device Architecture (CUDA) [2]. With the peak single precision floating point computing power exceeding 1 Tflops for recent NVIDIA GPUs and a reasonable thermal design power (TDP), the Gflops/watt is much higher than traditional CPU cluster, that is, the GPU computing is much greener than CPU computing. And more and more modern computer system ranging from personal computer to high performance computing server are equipped with one or more GPUs. By leveraging the computing capacity of GPU and traditional multicore CPU, performance of many program increase extremely. However, not all applications are easily ported to GPU and get higher performance than CPU because there are some restrictions on the applications that can execute efficiently on GPUs. Parallelization of program onto GPU will reveal these restrictions and help to improve the GPU architecture. In this paper, we propose techniques to reduce the influence of these restrictions and apply them to the parallelization of the HMMER's hmmsearch sequence database search tool on NVIDIA Tesla C2050 GPU. Hmmsearch is a sequence search application that search a sequence database for matches to an profile Hidden Markov Model (HMM) and is particularly well suited for many-core architectures due to the embarrassingly parallel nature of sequence database searches.

We demonstrate a variety of optimization strategies for hmmsearch, which is also useful for other GPU applications. In this paper, We make the following contributions:

- We find the possibility to unroll the outer loop of hmmsearch kernel and execute the unrolled loop speculatively. Apply many optimizations to minimize the number of device memory access and produce a highly optimized implementation that is faster than current SSE implementation.
- Locate some restrictions of the GPU hardware that may harm performance of applications running on it.

This paper is organized as follows: section II introduces the backgrounds of GPU computing and HMMER search; section III is the related work; our implementation is described in section IV; section V gives the result and discussion. We also show the performance comparison between our implementation and latest SSE implementation; section VI are the conclusion.

II. Background

A. GPU Programming

In this section we describe the NVIDIA Tesla C2050 [1] GPU based on the new generation Fermi [3] architecture. With 1.03Tflops peak single precision floating point performance, C2050 has 3GB graphic memory which is also called device memory and 14 stream multiprocessors (SM). A new unified L2 cache is added into Fermi architecture for the device memory to improve performance of memory system. Each SM has 32 stream processors, 32KB registers and 64KB configurable memory. The 64KB configurable memory can be configured as 48KB shared memory and 16KB L1 cache or 16KB shared memory and 48KB L1 cache. Though there are 32KB registers, total amount of

978-0-7695-4676-6/12 $26.00 © 2012 Crown Copyright
DOI 10.1109/IPDPSW.2012.91

registers that a thread can used is 63, even if there are few threads.

CUDA using a Single Instruction Multiple Threads (SIMT) programming model to program NVIDIA GPU. The GPU hardware enables creating thousands of threads with little cost, and programmers need to create as many threads as possible to obtain higher performance. But number of threads supported on each SM is limited by the number of registers, which means that SM with more registers performs better. The doubled register number on a SM of Fermi architecture brings larger thread capability, and also brings improvement to kernels that is bounded by register number. Threads are partitioned into thread blocks that is mapped to and executed on SMs. Thread block is further partitioned into warps of 32 threads. There are two warp schedulers on each SM and each warp scheduler chooses a warp from a block to execute each cycle. Warps are dynamically partitioned from thread blocks by hardware and are transparent to programmer.

The memory hierarchy exposed to programmer by CUDA is complex. Many different kind of memory are exposed to programmer, including registers, shared memory, constant memory, texture memory and device memory. Registers are allocated by compiler and total number is limited. Shared memory is both readable and writable to threads mapped to its SM. The size of constant memory is 64KB and it is read only. The texture memory space is much larger than the constant memory. Both constant and texture memory space reside in device memory and have caches on chip. The latency of device memory is much larger than on chip memory. So a new unified 768KB L2 cache for device memory is added in Fermi architecture.

The memory access pattern to Device memory can significantly impact its bandwidth. If all memory access requests of different threads within a warp fit in 128 byte with start address being a multiple of 128, only one cache/memory transfer is needed. This situation with best performance is called fully coalesced memory access. If the distance between end address and start address is larger than 128 or the start address is not a multiple of 128, there will be more than one cache/memory accesses. In general, the more transactions are necessary, the more unused data are transferred, reducing the instruction throughput accordingly. So, we need to carefully arrange the input data to utilize this memory system property.

B. HMMER search

HMMER developed by Sean Eddy [18], [22] is a wildly used profile hidden Markov model (HMM) search tool. A HMM is a probabilistic finite state machine, and used to represent a sequence pattern. A group of such protein sequences is called a protein family, and a profile HMM, which is typically constructed from a multiple alignment of those sequences, is used to model a protein family.

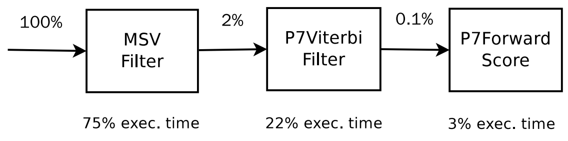

Figure 1. HMMER 3 execution pipeline, with profile data

The function of HMMER's hmmsearch tool is to search a sequence database for matches to an HMM.

The main procedure of hmmsearch is shown in algorithm 1, which is simple. The program reads a profile HMM and a sequence database as its inputs. It repeatedly reads one sequence from the sequence database and sents it to a process pipeline "p7_Pipeline" to process until all the sequences in database are processed.

The work flow of process pipeline "p7_Pipeline" is shown in figure 1: all sequences sent to the pipeline will processed by three filters. About 2% of all sequences will pass first filter and be sent to the next filter, and 0.1% sequences will reach the third filters. And profile result shows that first filter occupies 75% of total execution time and the second filter occupies 22%. Therefore, we concentrate on offloading the first filter of process pipeline onto GPU to accelerate the whole hmmsearch program.

Parallelism analysis: all sequences read from the input database are processed independently and can be processed parallel.

The pseudo-code of filter 1 is shown in listing 2, where L is the length of sequence and Q depends to the length of HMM. We can find some characteristics of MSVFilter algorithm:

- The working set is large and there is almost no data reuse. The data accessed is far from each other, which makes the cache system perform badly;
- Different iterations of outer loop depends on its former iterations, that is, the algorithm should only be executed serially.

```
dp[i][k] = rsc[k] + max(dp[i-1][k-1], xmb[i-1])
xme[i] = max(dp[i][k]) k=1...Q-1
xmj[i] = max(xmj[i], xme[i])
xmb[i] = max(arg2, xmj[i])
xmb[i] = max(0, xmb[i]-arg3)
```

Listing 1. Formulas implemented by MSVFilter

The dependency is explicitly shown in listing 1. The calculation of *dp*[i][k] depends on *xmb*[i-1], which depends on the maximum of all *dp*[i-1][k], where k ranges from 0 to Q-1.

Algorithm 1: pseudo-code of hmmsearch

```
Input: A profile HMM and a sequence DB
while((seq=ReadOneSequenceFromDB)!=0){
    p7_Pipeline(HMM, seq)
}
```

Algorithm 2: pseudo-code of p7_Pipeline

```
score1 = MSVFilter(HMM, seq)
if score1 > F1
    score2 = P7ViterbiFilter(HMM, seq)
if score2 > F2
    score3 = P7Forward(HMM, seq)
if score3 is significant
    seq is match
```

```
for (i = 1; i <= L; i++) {
    //prepare data for the for inner loop
    input[0] = dp[i-1][Q-4]
    input[1] = dp[i-1][Q-3]
    input[2] = dp[i-1][Q-2]
    input[3] = dp[i-1][Q-1]

    //read row i-1 of dynamic programming
        matrix to calculate row i
    for (k = 0; k <Q; k++) {
        //calculate one element
        temp = max(input[k%4], xmb)
        temp = min(255, temp+arg1)
        temp = max(temp-rsc[k], 0)
        xme = max(xme, temp)
        //prepare data for next loop
            iteration
        input[k%4] = dp[i-1][k]
        //write result to row i of dp matrix
        dp[i][k] = temp
    }
    ...
    //prepate data for next iteration. Note
        that xmb depends on xme which is the
        maximum of all elements in row i of
        dp matrix. This is the dependency
        between outer loop iterations
    xmj = max(xmj, xme)
    xmb = max(arg2, xmj)
    xmb = max(0, xmb-arg3)
    ...
}
```

Listing 2. Pseudo-code of MSVFilter

III. Related Work

HMMER3 [22] implements a new probabilistic model of local sequence alignment and a new heuristic acceleration algorithm. Combined with efficient vector-parallel implementations on modern processors, HMMER3 gains roughly 100-fold speedup relative to previous versions of HMMER.

There has been a great deal of work on optimizing HMMER 2 for traditional parallel computers [13]–[16]. J. P. Walters and etc. [15], [16] developed a MPI HMMER 2 implementation on cluster, which will get near liner speedup when the number of nodes is less than 64. There are many parallel implementation on novel hardware such as FPGA [6]–[9], GPUs [4], [5], [10], [23], network processor [11] and Cell/B.E. Processor [12]. The first GPU implementation of hmmsearch is ClawHMMER [4] by D. R. Horn. They implement on an ATI graphic card, which is different from ours. GPU-HMMER [5] is the first implementation on NVIDIA GPU. Their target GPU is NVIDIA 8800 GTX of G80 architecture and HMMER 2.3.2, while ours is a Fermi architecture GPU with many differences and newest HMMER 3.0. Their optimization mainly focus on memory system such as coalesced memory access and utilize faster texture and constant memory instead of device memory. However, we focus on to reduce memory access operation by making full use of registers, to cover data transfer time between host and GPU by calculation, to reduce pressure to memory system by speculatively executing the kernel. P. Yao [10] introduced a load balanced GPU implementation of hmmsearch. They concentrate on using idle CPU cores to offload part of calculation that the GPU is not good at and to presort the input sequences to balance load between different GPU threads. Not much effort is done to optimize their kernel implementation, while in this paper we mainly focus on optimizing the GPU kernel.

Both [9] and [23] are works that parallelizing HMMER 3.0. Work [9] is done on FPGA platform and [23] chooses GPU as their platform. The different between [23] and our work is that different computing division methods are used: they use one block to calculate the score of one sequence while we use one thread to do the same work. Though bandwidth is the key to achieve high performance, their method cannot reduce read or write operations to the device memory, while our speculative method can do, resulting a faster implementation.

IV. Our hmmsearch implementation and optimizations

A. Our hmmsearch implementation

In this section we describe our implementation of hmmsearch and optimizations to the first filter of p7_Pipeline. We provide details and performance results of various optimizations to hmmsearch version 3.0. All tests were performed on a machine consisting of a 2.13GHz Intel Xeon E5506 quad-core processor with 12 GB main memory and 1 NVIDIA

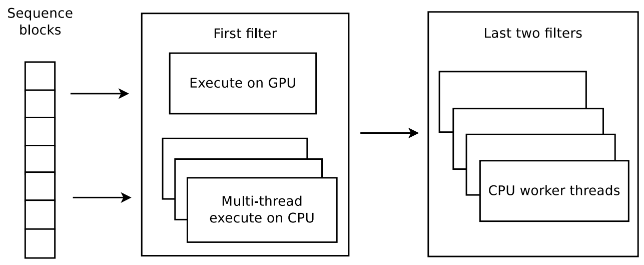

Figure 2. Work flow of our hmmsearch implementation

Tesla C2050 GPU. The CUDA version is 4.0. The host compiler is GCC version 4.4.5.

As shown in figure 2, our hmmsearch implementation works as follows: There are 1 data prepare thread, 1 GPU control thread and 3 worker threads. The data prepare thread reads sequence from database, does some preprocess such as digitize and puts the result sequences into blocks. All sequence blocks are stored in one global queue. The GPU control thread continuously gets a block from global queue, extracts necessary data into arrays, copies these arrays into device memory and then waits until GPU finishes computing the score of all sequences in the block. After that, it copies result to host memory and put the block into the input queue of worker thread. The responsibility of three worker thread is to fetch a sequence block from the global queue and calculate the score of sequences in these blocks. First filter of p7_Pipeline is calculated by both CPU and GPU, and then the last two filters of pipeline are calculated by the three CPU worker threads simultaneously. When the end of sequence database reached, data prepare thread will enqueue an empty block to signal the GPU control thread and 3 worker threads to return.

The GPU kernel speculatively computes the score of sequences. There will be wrong result when speculation is incorrect. All sequences that have incorrect score are marked as to be recalculated by GPU kernel. And worker threads will detect these sequences and recalculate their scores.

B. *Optimizations to our implementation*

The following optimizations are used to optimize the computing kernel running on GPU.

- A lot of ordinary optimizations such as coalesce access to *dp* array stored in global memory; put the *rsc* array into texture memory and adjust its arrangement to make full use of texture cache; asynchronously transfer data to device memory to cover data transfer time and kernel execution time. Because the host need only write device memory on GPU, we use write-combined flag to optimize the data transfer speed between host and GPU [19]. Asynchronously call the computing kernel to simultaneously do the calculation on GPU and calculation on host, which will hide the operation with shorter execution time and reduce the total execution time. Pre-sorting the input database to moderate the load imbalancing problem caused by the different sequence length processed by threads in a block. The longest sequence is put at the front of database to reduce execution time when there are not enough sequences in the last block. We put these wildly used optimizations together to evaluate them as one optimization.
- Use *int* type to substitute the *uchar4* type. To convert the SSE version of programs to CUDA kernel, the *uchar4* is an intuitive type to replace the 16-way vector type of SSE. However, the *uchar4* type is only a defined structure in CUDA and not natively supported by the underlying hardware, which result in an inefficient implementation. While the 32-bit *int* type is natively supported on NVIDIA Fermi architecture, we use *int* type as default to store intermediate result and almost all calculations are carried on *int* type.
- Unroll the outer loop of listing 2. There are dependencies between different iterations of outer loop in listing 2. Rows of dynamic programming matrix (*dp* in listing 2) are calculated by the loop one by one. One loop need the result of the former one to calculate a new *xmb* value which is needed to calculate its own row of dynamic programming matrix. So, unrolling of outer loop directly has no benefit except more registers are occupied, unless we can speculatively execute the second loop. We found that the *xmb* variable in listing 2 does not change frequently between two continuous loop. In fact, the *xmb* variable will not change in the calculation of almost all the sequences. In all our test cases, there are at most 1% sequences in the calculation of which the *xmb* variable changed. For this reason, we unroll the outer loop 2 times and execute the second iteration speculatively. The pseudo-code of the unrolled loop is shown in listing 3. The *xmb* variable will be stored in a separated register before execution and checked after execution. If *xmb* is changed by the first iteration, the sequence will be tagged as to recalculate, otherwise, the next two iterations will be executed. The host will copy the recalculation information when the calculation of one chunk of sequences have been finished. The recalculation information is then checked to choose sequences that need be recalculated and the recalculation is carried out in host CPU. As shown in listing 3, after unrolling, Q reads and Q writes operations to device memory are saved for every two loop.

```
for (i = 1; i <= L; i+=2) {
    input[0] = dp[i-1][Q-4]
    input[1] = dp[i-1][Q-3]
    input[2] = dp[i-1][Q-2]
    input[3] = dp[i-1][Q-1]

    old_xmb = xmb;
    for (k = 0; k <Q; k++) {
        //loop iteration i
        temp = max(input[k%4], xmb)
        temp = min(255, temp+arg1)
        temp = max(temp-rsc[k], 0)
        xme = max(xme, temp)
        input[k%4] = dp[i-1][k]
    //  dp[i][k] = temp //no longer needed

        //loop iteration i+i
        temp2 = max(dp[i][k], xmb)
        temp2 = min(255, temp2+arg1)
        temp2 = max(temp2-rsc[k], 0)
        xme = max(xme, temp2)
    //  input[k%4] = dp[i-1][k] //no longer
        needed
        dp[i][k] = temp2
    }
    ...
    xmj = max(xmj, xme)
    xmb = max(arg2, xmj)
    xmb = max(0, xmb-arg3)
    if(xmb!=old_xmb){
        recalculated = 1
        break; //this thread will exit, and
            score will be recalculated
    }
    else{
        score is correct and execute next two
             loop iterations
    }
    ...
}
```

Listing 3. Pseudo-code of the unrolled loop using speculative execution technology

V. Result and Discussion

We choose 6 HMMs of length 63, 112, 215, 423, 830, 1774. All the HMM are taken from Pfam database [17]. And the input sequence database is NCBI NR [21] database with more than 15.2 million sequences. The length of all sequences varying from 5 to 41943. We launch our kernel using the configuration of 28 blocks with 512 threads each. The block number is 28 because there are 28 warp scheduler on Tesla C2050 GPU. 512 threads per block is maximum number that supported by hardware because of the register limitation.

Figure 3 is the performance improvement of different kernel optimizations. "Basic" version is converted from SSE implementation manually, all ordinary CUDA optimization technologies shown in the first item of subsection IV-B are used in this configuration. The "Optimization 1" version is the configuration that apply the optimizations shown in the second item of subsection IV-B to "Basic" version. The "Optimization 2" version is our final version, which is the result of applying the optimizations shown in the last item of subsection IV-B to "Optimization 1" version.

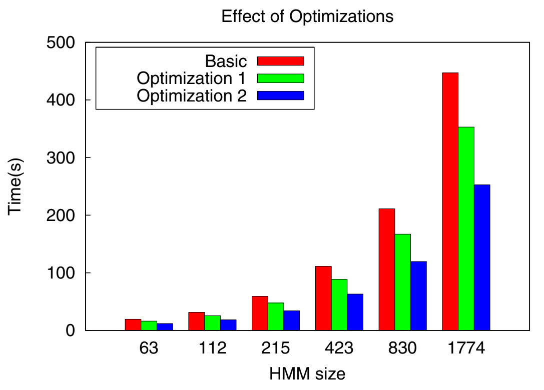

Figure 3. Performance improvement of kernel optimizations

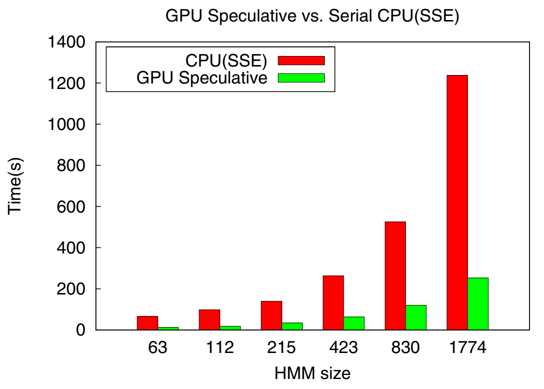

Figure 4. Execution time of first filter

The result is the same as we expected: "Optimization 1" runs faster than "Basic" version because the *int* type is natively supported while the *uchar4* is not; "Optimization 2" runs faster than "Optimization 1" because about half read and write operations to device memory are reduced. Though half access operations to device memory are reduced, less than half execution time is reduced. That is because there are not enough active threads, which means there are not enough active warps per cycle, and hence on-chip resource utilization is low.

As shown in figure 4, though there are many hardware restictions, speed of GPU is much faster than that of SSE version running on CPU, because the higher peak performance provided by GPU hardware.

As shown in figure 5, when HMM size is small, speedup is low. The reason is that to calculate the score of sequences against a small HMM, not much time is need, while the data

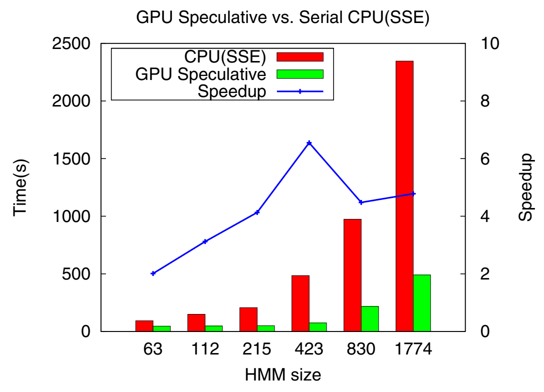

Figure 5. Final speedup compared to serial SSE HMMER

prepare thread need much time to prepare input sequences for p7_Pipeline. When the HMM size increases, time needed to calculated score of all sequences increase too, resulting in the increasing of the part that are parallelly executed on GPU. However, when HMM size changes from 423 to 830 the speedup decreases. The answer is shown in table 1: the recalculate count increases about 4.4x and passed MSVFilter count increased 1.4x, which introduce much work for GPU control thread.

Profiled by Compute Visual Profiler [20] provided by NVIDIA, we found that the average active warp per cycle is about 15, while there are 28 warp scheduler. That is to say, about half of compute capability is not used. This is because register file of Fermi architecture is too small for this application, which makes hmmsearch become a memory bound application on GPU. A feasible way to increase performance without modifying hardware is to decrease the per thread register file usage, which enables GPU to support more active threads to cover long latency device memory access operation.

Our implementation mainly reduce execution time in two aspects. First, host CPU and GPU execute in an asynchronous way, which means that the one that has longer execution time covers the other one. Second, the kernel execution time saved by various optimizations to the kernel.

The final result of our implementation is shown in Figure 5, we can see that the maximum speedup compared to the serial version is more than 6.5x.

HMM size	63	112	215	423	830	1774
Recalculate count(%)	0.37	0.53	0.12	0.23	1.02	0.92
Passed MSVFilter(%)	2.34	3.75	3.1	4.64	6.48	5.82

Table I
PERCENTAGE OF RECALCULATE AND PASSED MSVFILTER

VI. CONCLUSION

We have presented the details of our hmmsearch implementation on new generation NVIDIA GPUs. We show that the hmmsearch is a memory bound application on GPU and try to minimize the memory access operation of each loop iteration using speculative loop unrolling and other assistant technologies. Performance result shows that the hmmsearch achieves excellent performance improvement, up to 6.5x speedup compared to the serial implementation. The optimization technologies used to optimize hmmsearch such as loop unrolling, storing intermediate results into registers, making full use of texture memory, coalescing memory access and pipelining are also useful in parallelizing other applications onto GPU.

In the practice of parallelizing hmmsearch onto GPU, we have some experiences in programming GPU: First, const memory and texture memory are fast when accessed in a cache friendly pattern and should be considered as the position to store constant data first, even though there are L2 cache for device memory. Because the 768KB L2 cache is too small for 3GB device memory and thousands of threads, the constant data automatically loaded into L2 cache by hardware is probably replaced by other "hotter" data. While stored in constant or texture memory, the data uses a separate cache that cannot be affected by access to device memory. Second, making fully use of registers may greatly improve performance. The register is fastest in all kind of memories in GPU architecture and the 32KB capacity per SM is large enough for many applications. Third, to achieve high performance, coalesced memory access to shared memory, constant memory, texture memory and device memory is important.

And some perspective on GPU hardware: first, the size of shared memory is small. Shared memory per stream processor of Fermi architecture is even smaller than pervious architecture, 1.6KB per stream processor compared previous 2KB per stream processor. Second, maximum number of registers one thread can use is small, which is limited to 63 in Fermi architecture. When thread number is small and per thread register requirement is large, just like hmmsearch, this hardware configuration cannot satisfy the need of programmers. Third, total number of registers is not large enough. As a result, one SM cannot support sufficient active threads to run on it, which limit the average active warps on GPU and hence the final speedup. Finally, arithmetic operations on *char4* and other similar data types are not natively supported by current GPUs, which will limit the speedup of GPU implementation compared to the SSE implementation.

VII. ACKNOWLEDGMENTS

This work is supported financially by the National Basic Research Program of China under contract 2011CB302501, the National Natural Science Foundation of China grants

60970023, the National Hi-tech Research and Development Program of China under contracts 2012AA010902 and 2012AA010303, the National Science & Technology Major Projects 2009ZX01036-001-002 and 2011ZX01028-001-002-3.

REFERENCES

[1] NVIDIA. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf

[2] NVIDIA. CUDA C programming guide. NVIDIA, 4.0 edition, 2011.

[3] NVIDIA. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, version 1.1, 2009.

[4] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A Streaming HMMer-Search Implementation. *International Conference for High Performance Computing, Networking, Storage and Analysis(SC)*, 2005.

[5] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary. Evaluating the use of GPUs in Liver Image Segmentation and HMMER Database Searches. *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[6] Y. Sun, P. Li, G. Gu, Y. Wen, Y. Liu, and D. Liu. Accelerating HMMer on FPGAs Using Systolic Array Based Architecture. *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[7] T. Takagi and T. Maruyama. ACCELERATING HMMER SEARCH USING FPGA. *International Conference on Field Programmable Logic and Applications*, 2009.

[8] R. P. Maddimsetty, J. Buhler, R. D. Chamberlain, M. A. Franklin, and B. Harris. Accelerator Design for Protein Sequence HMM Search. *International conference on Supercomputing*, 2006.

[9] N. Abbas, S. Derrien, S. Rajopadhye, and P. Quinton, Accelerating HMMER on FPGA using Parallel Prefixes and Reductions, *International Conference on Field-Programmable Technology (FPT)*, 2010.

[10] P. Yao, H. An, M. Xu, G. Liu, and Y. Wang. CuHMMer: A Load-Balanced CPU-GPU Cooperative Bioinformatics Application. *International Conference on High Performance Computing and Simulation (HPCS)*, 2010.

[11] B. Wun, J. Buhler, and P. Crowley. Exploiting Coarse-Grained Parallelism to Accelerate Protein Motif Finding with a Network Processor. *International Conference on Parallel Architectures and Compilation Techniques(PACT)*, 2005.

[12] J. Lu, M. Perrone, K. Albayraktaroglu, and M. Franklin. HMMer-Cell : High Performance Protein Profile Searching on the Cell/B.E. Processor. *IEEE International Symposium on Performance Analysis of Systems and software(ISPASS)*, 2008.

[13] J. P. Walters, R. Darole, and V. Chaudhary. Improving MPI-HMMER's Scalability With Parallel I/O. *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[14] S. Isaza, E. Houtgast, F. Sanchez, A. Ramirez, and G. Gaydadjiev. Scaling HMMER Performance on Multicore Architectures. *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, 2011.

[15] J.P. Walters, B. Qudah, and V. Chaudhary. Accelerating the HMMER Sequence Analysis Suite Using Conventional Processors. *20th International Conference on Advanced Information Networking and Applications(AINA 2006)*, 2006.

[16] J. Landman, J. Ray, and J.P. Walters. Accelerating HMMer searches on Opteron processors with minimally invasive recoding. *In Proceedings of HiPCOMB,* 2006.

[17] Pfam database: http://pfam.sanger.ac.uk/

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

[19] NVIDIA. CUDA Toolkit Reference Manual. NVIDIA, Version 4.0, 2011.

[20] NVIDIA. Compute Visual Profiler User Guide. NVIDIA, Version 4.0, 2011.

[21] NCBI. The NR database. ftp://ftp.ncbi.nih.gov/blast/db/FASTA/nr.gz, 2011.

[22] S.R. Eddy, A new generation of homology search tools based on probabilistic inference. Genome Informatics, 23:205211, 2009.

[23] S. Quirem, F. Ahmed, and B. K. Lee, CUDA acceleration of P7Viterbi algorithm in HMMER 3.0, *30th IEEE International Performance Computing and Communications Conference (IPCCC)*, 2011.