Hotspot Analysis Based Partial CUDA Acceleration of HMMER 3.0 on GPGPUs

Fahian Ahmed, Saddam Quirem, Gak Min and Byeong Kil Lee

Abstract*— *With the introduction of many-core GPUs, there is widespread interest in using GPUs to accelerate non-graphics applications such as bioinformatics, energy, finance and several research areas. Even though the GPUs provide highly parallel processing capability, the communication interface between CPU and GPU could be a performance bottleneck due to heavy data transfer. If data transfer time is overwhelming the computation time on GPU, it would be better keep the computation on CPU instead of using GPUs. In this paper, we characterize the HMMER 3.0 and investigate performance hotspot functions. The HMMER is a bioinformatics application which is used in searching sequence databases for protein sequences. For our experiment, we use Nvidia CUDA that abstracts the GPU hardware. Based on the hotspot analysis of HMMER 3.0, we consider two factors for partial CUDA acceleration: one is the performance impact of major hotspot functions and the other one is data transfer overhead. Also, we verified that hotspot analysis based partial CUDA acceleration could provide better performance than full CUDA implementation.

Index Terms*—*CUDA acceleration, GPGPU, HMMER, Many-core processors

I. INTRODUCTION

Future microprocessor development efforts will continue to concentrate on adding cores rather than increasing single-thread performance [1]. Highly parallel graphics processing unit (GPU) is rapidly gaining maturity as a powerful engine for computationally demanding applications. The GPU's performance and potential offer a great deal of promise for future computing systems. However, the architecture and programming model of the GPU are slightly different from the commodity of single-chip or heterogeneous processors.

Even with powerful and massively parallel GPUs, it is difficult to achieve peak performance without the knowledge of graphics or graphics dedicated APIs However, with the introduction of new programming models such as Nvidia's CUDA that abstracts the GPU hardware, non-graphics users can easily map wide range of applications into many-core GPUs [1][2].

The GPGPU (General-purpose computing on graphics processing unit) is a technical snapshot of using a GPU, which has high data-parallel processing capability and typically handles computation only for computer graphics, to perform the computation in general-purpose applications traditionally handled by general-purpose CPU. Applications for GPGPU include bioinformatics, energy, finance and various research areas [1].

In this paper, we focus on HMMER 3.0 for hotspot analysis and partial CUDA acceleration. The HMMER is a bioinformatics application which is used in searching sequence databases for protein sequences. No GPGPU acceleration research on HMMER 3.0 is found based on our knowledge and investigation. Compare to existing version, HMMER 3.0 has more performance improvement features such as a heuristic filter, a log-likelihood model, etc. we characterize the HMMER 3.0 and investigate performance hotspot functions. Based on the hotspot analysis results, we investigate the performance impact from each hotspot acceleration and full acceleration. Also, we observe that the performance bottleneck can be from a structural issue between host device and GPU as a co-processor. Data transfer overhead between heterogeneous processors could be a performance bottleneck. We could solve this issue through the coarse-grain hotspot analysis and remove the cause of performance bottleneck.

The rest of paper is organized as follows: section II describes related works. Scalability and Performance hotspot analysis of HMMER is presented in section III. Section IV shows partial CUDA implementation of HMMER. Finally, concluding remarks and future works are presented in the last section.

II. RELATED WORKS

Major genetic applications known to make use of the GPU include Gromacs, NAMD, HMMER and most notably Folding@home. NVIDIA GPUs account for over 35% of Folding@home's native TFLOPS. HMMER itself is a database search application, and like many similar applications, GPUs have been applied for acceleration. Bakkum et al. [3] have previously ported SQLite to CUDA resulting in at least 20x speedups in query time. For HMMER application, various types of coprocessors were utilized for acceleration. Perhaps most interesting acceleration is done with the FPGA by Steve Derrie and Patrice Quinton [4]. This is where the P7Viterbi algorithm was implemented in hardware as a set of MUXs and LUTs. This FPGA implementation achieved a 50x speedup in one case. A CUDA implementation of HMMER is also present. Walters et al accelerated HMMER by focusing on the P7Viterbi algorithm at the core of the application. Using a single Tesla GPGPU, GPU-HMMER was capable of a 30x speedup with a large HMM size (number of states). Likewise, an earlier implementation of HMMER utilizing streaming processors (which includes NVIDIA GPUs), known as ClawHMMER, took a similar approach by targeting the P7Viterbi algorithm [5]. *While previous implementations are with old version of HMMER, we focus on newer version of HMMER with different memory allocation schemes.*

Manuscript received September 02, 2012.

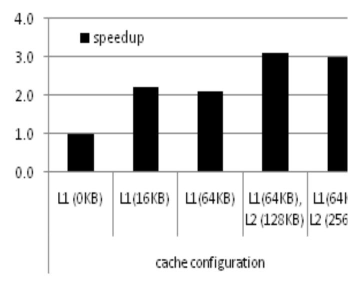
Fahian Ahmed, Department of Electrical and Computer Engineering, University of Texas at San Antonio, USA .
Saddam Quirem, Department of Electrical and Computer Engineering, University of Texas at San Antonio, USA.
Gak Min, Department of Engineering, Korea Broadcasting System, Korea.
Byeong Kil Lee, Department of Electrical and Computer Engineering, University of Texas at San Antonio, USA .

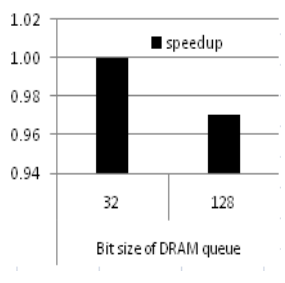
It is well known that PCI Express bandwidth can cause a throughput bottleneck when a significant amount of data is transferred between a CPU and a GPU in a heterogeneous system. A number of researchers have discussed bandwidth troubles that can arise with frequent or poorly managed data movement between devices. Schaa and Kaeli [6] examine multiple GPU systems and acknowledge that unless a full working set of data can fit into the memory on a GPU; the PCI Express will be a bottleneck. Owens et al. [7] express similar concerns. Fan et al. [8], Cohen and Molemaker [9] and Dotzler et al. [10] all recommend rewriting algorithms to limit PCI Express transfers as much as possible. The aforementioned studies have served as our motivation for this paper, and we decided to quantify the memory transfer overheads for HMMER.

III. SCALABILITY AND PERFORMANCE HOTSPOT ANALYSIS OF HMMER 3.0

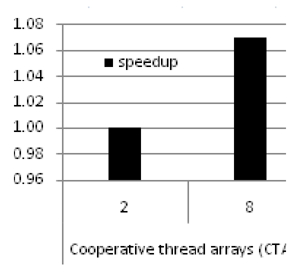
A. Scalability Analysis on GPU simulator

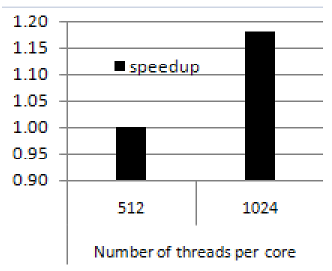
Since there is a limitation to use many GPU hardware components for scalability analysis, we used a GPU simulator, called GPGPU-Sim [11].





(a) cache configuration (b) DRAM queue size





(c) CTA (b) Number of threads per core

Figure 1: Speedup of HMMER (in IPC) for variation of cache configuration, bit size of DRAM, cooperative thread arrays (CTAs) and no of threads/core

Figure 1 shows speedup in IPC for variation of cache configuration (L1 and L2), bit size of DRAM queue, cooperative thread arrays (CTAs) and number of threads per core. Figure 1 (a) shows the performance measured in IPC and normalized to L1 Cache of 0. It is clear that for all programs, GPU performance is worst with no cache, most likely due to the large data transfer time between registers and the global memory. However, as the L1 and L2 cache size increases, HMMER speedup increases gradually. Figure 1 (b) shows the IPC variation for increasing the DRAM queue size for each program, normalized to DRAM queue size of 32. The outcome is not a surprise because if the bit size is increased and everything else kept the same, there would be an increase in delay due to larger packets being sent back and forth from global memory. Figure 1 (c) shows performance variation for Cooperative Thread Array (CTAs) or more commonly known as the number of blocks in the CUDA architecture. Figure 1 (d) shows the performance impact from the variation of threads per cores. In both cases as the CTAs and number of threads/cores increases, the performance of HMMER increases gradually. Based on this simulation, we observe that the performance of HMMER application can be scalable with the number of functional units and the number of other resources.

B. Hotspot analysis of HMMER 3.0

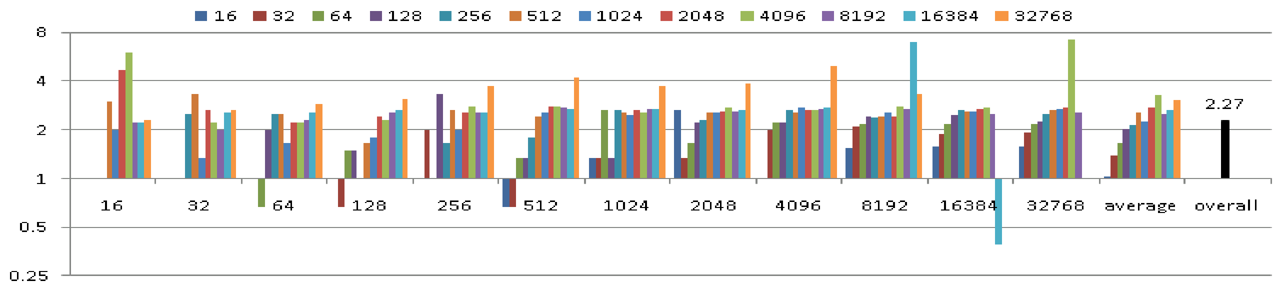
The Intel VTune Performance Analyzer [12] provides detailed information on the execution of the code. The VTune shows the performance issues, enabling to focus tuning effort and get the best performance boost in the least amount of time. Table 1 shows the machine configuration for CPU and GPU which is a heterogeneous computing. In our experiment, we use 12 cores Intel Xeon workstation with Nvidia CUDA support GPGPU which as GTX 460 and Tesla X1060.

Table 1: Heterogeneous machine configuration

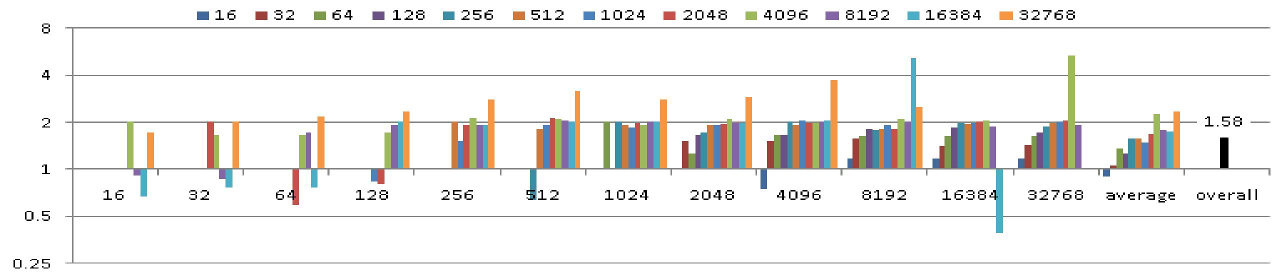
Configuration	Intel Xeon W5590	Nvidia GTX 460	Tesla C1060
No. of Cores	12	336	240
Main Memory	12GB	1GB	4GB
Memory I/O	64-bit	256-bit	512-bit

The Hotspots analysis helps understand the application flow and identify sections of code that took a long time to execute (hotspots). A large number of samples collected at a specific process, thread, or module can imply high processor utilization and potential performance bottlenecks. The HMMER 3.0 is applied to the Vtune to extract hotspot functions. There are several sub-programs in HMMER application like phmmer, jackhammer, hmmbuild, hmmsearch, hmmscan, smmaligh, etc. Each sub-program shows similar hotspot results, but here we only discuss about jackhammer. Jackhmmer program is for searching a single sequence query iteratively against a sequence database. The VTune Analyzer creates and run an activity that collects performance data of the application. An activity means lunching application onto Vtune profiler. Table 2 shows the sampling summary view of the Jackhmmer.

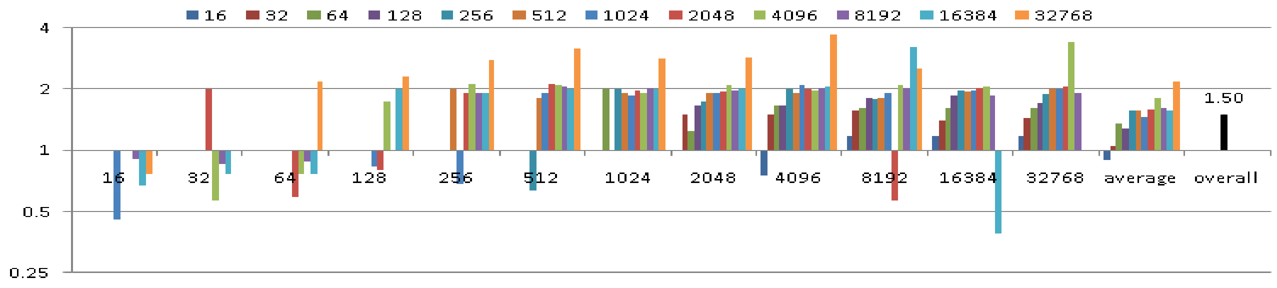
The Sampling summary provides data on the top most active functions in the system during the data collection. Each Row represent active functions with function name corresponding number of samples, percentage of CPU clock and total number of thread events.



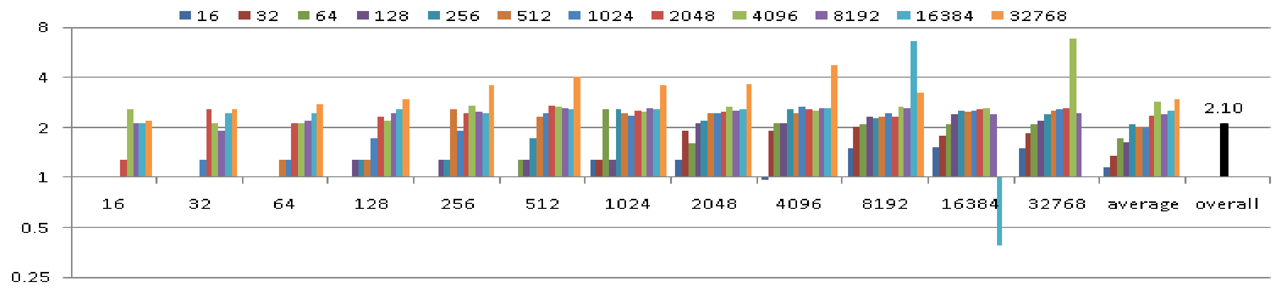
(a) Forward module speedup



(b) Backward module speedup



(c) P7_ Viterbi module speedup



(d) Combined module speedup

Figure 2: Performance comparison of Hotspot-based partial CUDA and full CUDA implementation

Table 2 Hotspot analysis: sampling summary view of the Jackhmmer

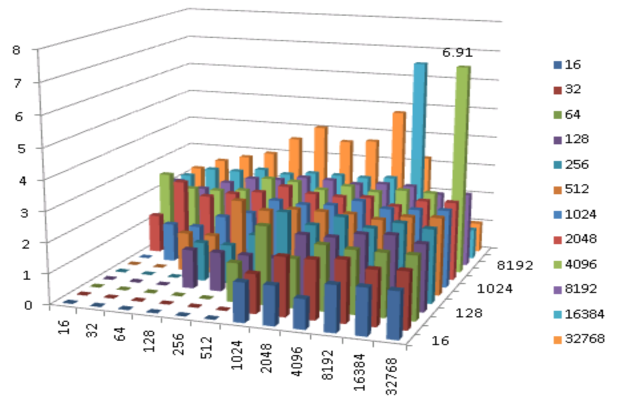
Name	Thread samples	CPU clock (%)	Thread events
forward_engine	64	36.57%	136192000
backward_engine	54	30.86%	114912000
p7_Viterbi	34	19.43%	72352000
p7_Decoding	9	5.14%	19152000
p7_Null2_ByExpectation	7	4.00%	14896000
p7_alidisplay_Create	1	0.57%	2128000
p7_oprofile_FGetEmission	1	0.57%	2128000
is_multidomain_region	1	0.57%	2128000
rescore_isolated_domain	1	0.57%	2128000
p7_MSVFilter	1	0.57%	2128000
get_postprob	1	0.57%	2128000
esl_hmm_Forward	1	0.57%	2128000

C. Hotspot analysis based CUDA acceleration

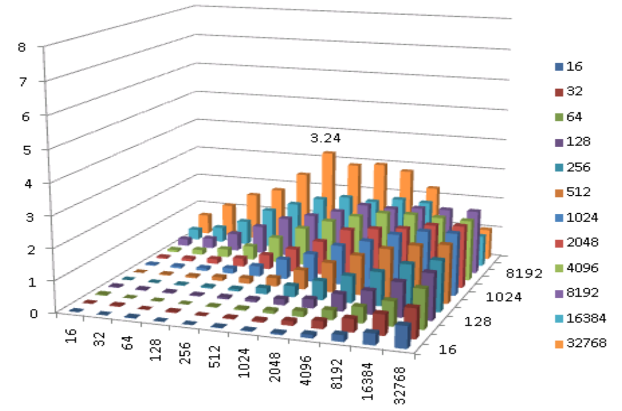
A key issue of programming with the GPU is the fact that before computations may be performed on the GPU, memory blocks must be allocated onto it, data transferred to it, and finally the data must be transferred back to the host after the computations are performed. This ultimately means that when working with smaller pieces of data, the CPU will undoubtedly outperform the GPU. There are multiple methods that aim to alleviate this issue, but cannot fully remove it. Thus, we aim to observe the borderline between CPU vs. GPU performance.

According to the Table 2 (hotspot analysis of HMMER), forward_engine, backward_engine and p7_Viterbi functions are the dominant of CPU clock which takes 37%, 31% and 20% of the total respectively. These three modules are the three canonical problems to solve with HMM (Hidden Markov Model). Actually, The Hidden Markov Model (HMM) is a variant of a finite state machine having a set of hidden states, an output alphabet (observations), transition probabilities, and output (emission) probabilities. Forward and Backward module compute the probability of a particular output sequence. On the other hand p7_Viterbi module finds the most likely sequence of (hidden) states which could have generated a given output sequence. Based on hotspot analysis, we investigate of performance improvement for these three individual modules with CUDA acceleration respectively. By comparison to full CUDA conversion, we can figure out which function has most impact from data transfer overhead between CPU and GPU.

With each implementation of the HMMER's modules, timers were utilized in order to properly compare the results. For the CPU version, the elapsed time was simply measured between the start of the function and after the completion of the function. The GPU implementations utilize two timers that measure the total time taken to allocate memory storage and transfer memory onto the GPU and back, and the kernel execution time. The second timer simply measures the time taken to execute the CUDA kernels.



(a) Kernel speedup of Combined module



(b) Total speedup of Combined module

Figure 3: Performance comparison of Kernel only and combined module

IV. HOTSPOT ANALYSIS BASED PARTIAL CUDA ACCELERATION

A. Partial CUDA acceleration for individual hotspot module

The Forward Algorithm is a recursive algorithm for calculating the observation sequence. Figure 2 (a) shows the speedup of forward engine only CUDA acceleration. Based on our experiments, the CUDA implementation of the forward engine shows an about 2.27x speedup over the original CPU-only implementation (C-version). The number of queries and sequence length is ranging from 16 to 32,768. Backward engine is exactly same with forward engine which calculates recursively backward variables going backward along the observation sequence. Figure 2 (b) shows an about 1.58x speedup for backward engine only CUDA modeling. P7_viterbi Module chooses the best state sequence that maximizes the likelihood of the state sequence for the given observation sequence. The P7_Viterbi algorithm iterates through every observation, from 1 to L (Sequence length), and determines a score for each state from 1 to M (queries). It is not possible to calculate the scores for each observation in parallel. Based on our experiments, the CUDA implementation of the P7Viterbi algorithm shows an about 1.50x speedup over the original implementation. The speedup increased exponentially as the number of threads launched (number of queries) doubled.

We also tried to combine three modules to see the performance impact. Figure 2 (d) shows all three modules, speedup is around 2.10x over the original implementation of the functions. It is interesting to note that among four combinations (forward, backward, p7-viterbi and combine), forward engine shows better speedup. The lowest performance is shown by p7-viterbi. The reason is coming from hotspot analysis. Since forward engine has highest clock time among three it is the most dominant module and CUDA conversion of forward module shows large impact on speedup. Backward engine is the second one and p7-viterbi shows least speedup according to hotspot analysis based partial acceleration. Combine modules shows a mixed speedup which lies between forward and backward module.

B. Data transfer overhead investigation

In order to investigate the real speedup from CUDA acceleration, we investigate the cycle time of memory storage allocation and data transfer time and include them to total cycle time. If we use a full CUDA acceleration with forward, backward and p7-viterbi, the highest speedup is 6.91x and the average speedup is around 2.10x as shown in Figure 3 (a). However, if we consider total time taken to additional overhead, then the highest speedup reduces to 3.24x over the original implementation. Figure 3 (a) and 3 (b) show the speedup of kernel itself and total execution time which includes kernel and data transfer overhead. Based on this investigation, it is clear that data transfer time has a great impact on speedup. If we consider the kernel execution time only, speedup looks much better. But if we consider both kernel and additional overhead, then speedup reduces significantly.

The reason is that some functions show higher data transfer time than kernel computations on GPU, but others are not. Therefore, full CUDA modeling could not be an ideal solution for all applications. Static or dynamic scheduling could be one solution to decide whether it will be CUDA compiled or executed to achieve maximum performance.

V. CONCLUSION

The GPGPU (General purpose computing on graphics processing unit) is a technique of using a GPU, which has high data-parallel processing capability and typically handles computation only for computer graphics, to perform the computation in general purpose applications traditionally handled by CPU. With the introduction of manycore GPUs, there is widespread interest in using GPUs to accelerate non graphics applications such as Hmmer.

Even though the GPUs provide highly parallel processing capability, the communication interface between CPU and GPU could be a performance bottleneck due to heavy data transfer. If data transfer time is overwhelming the computation time on GPU, it would be better keep the computation on CPU instead of using GPUs. We characterize the HMMER 3.0 and investigate performance hotspot functions. The VTune analyzer shows the performance issues, enabling to focus tuning effort and get the best performance boost in the least amount of time. The Hotspots analysis helps understand the application flow and identify sections of code that took a long time to execute (hotspots). Based on the hotspot analysis of HMMER 3.0, we consider two factors for partial CUDA acceleration: one is the performance impact of major hotspot functions and the other one is data transfer overhead. Also, we verified that hotspot analysis based partial CUDA acceleration could provide better performance than full CUDA acceleration.

Based on the hotspot analysis, we do CUDA modeling on a hotspot function and it shows significant performance improvement on GPGPU with minimal efforts. We also tried with the various data size to see its performance sensitivity, and we see that one of the performance bottlenecks would be data transfer overhead between CPU and GPU. From the preliminary checking of CPU-GPU task scheduling, we see that most hotspot module on CPU is not always most performance factors when applying CUDA implantation with GPUs. One of the reasons would be communication overhead between CPU and GPU, which should be investigated more thoroughly. That way, we can exploit more effective CPU-GPU task scheduling with static or dynamic mapping techniques. Also, more non-graphics applications need to be characterized and investigate its fitness to heterogeneous system.

REFERENCES

1. Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. GPU computing. IEEE Proceedings, 879-899, May 2008.
2. Jakub Kurzak, Alfredo Buttari, Piotr Luszczek, Jack Dongarra, "The PlayStation 3 for High-Performance Scientific Computing," Computing in Science and Engineering, vol. 10, no. 3, pp. 84-87, May/June, 2008.
3. P. Bakkum and K. Skadron "Accelerating SQL database operation on a GPU with CUDA," in proceedings of the 3rd workshop on general purpose computation on graphics processing units, ACM, pp.94-103, 2010.
4. Steven Derrie and Patrice Quinton, "Parallelizing HMMER for Hardware Acceleration on FPGAs", Proceedings in IEEE 18th International Conference Application-specific Systems, Architectures and Processors, 2007.
5. Daniel Horn, Mike Houston and Pt Hanrahan, "ClawHMMER: A Streaming HMMer-Search Implementation", presented at Supercomputing 2005, Washington, D.C., 2005.
6. D.Schaa and D. Kaeli, "Exploring the multiple GPU design space," in International Parallel and Distributed Processing Symposium., pp. 1–12, May 2009.
7. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," Proceedings of the IEEE, vol. 96, no.5, pp. 879–899,May 2008.
8. Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in ACM/IEEE Conference on Supercomputing, Pittsburgh, PA, pp. 47–58, November 2004.
9. J. Cohen and M. Molemaker, "A fast double precision CFD code using CUDA," in Parallel Computational Fluid Dynamics: Recent Advances and Future Directions, Moffett Field, CA, pp. 414–429, May 2009.
10. G. Dotzler, R. Veldema, and M. Klemm, "JCUDAmp: OpenMP/Java on CUDA," in 3rd International Workshop on Multicore Software Engineering, pp. 10–17, May 2010.
11. Ali Bakhoda, George L. Yuan, W.L. Fung, Henry Wong and Tor M. Aamodt, "Analyzing CUDA workloads using a detailed GPU Simulator," 2009 IEEE International Symposium on Performance Analysis of Systems and Software, 2009.
12. Vtune: Intel Performance Analyzer,
13. http://www.software.intel.com/en-us/intel-vtune/