Performance estimation and application mapping on different GPUs

Abstract

The objective of my research is to get benefit of advancement in GPU architecture in the state of art software framework. Our work is divided into two parts; first, we have developed a model to estimate performance of GPUs with cache, second, we have analyzed the existing map-reduce framework to optimized the same for new GPU architectures. For performance estimation of GPUs with cache, first we try to estimate computation time and then follow it up with estimation of memory access time. We have developed a model to count the number of instructions in the kernel. We have found our instruction count methodology to give exact count. Memory access time is calculated in three steps; address trace generation, cache simulation and average memory latency per warp. Finally, computation time is combined with memory access time to predict the total execution time. This model has been tested with micro-benchmarks as well as real life kernels. We have found that our average estimation errors for these applications range from -7.76% to 55%.

In the second part of our work, we have enhanced the performance of MARS Map-Reduce(MR) framework. These improvements are mainly in the context of different GPU architectures. Our experiments show an average of 2.5x speedup of MARS MR framework on Fermi architecture. Cache reconfiguration effect is also explained here. Depending upon the application, we have achieved performance benefits ranging from 10% to 200% for various cache sizes. Our optimized group phase gives an average of 1.5x speed up. Which has been achieved by reducing the number of comparisons per thread. In the other significant optimization technique, delayed writing during auxiliary functions is implemented. This reduces significant cache misses and thus achieved about 2x to 6x speedup for these functions.

[1]

1 Introduction

Due to power and temperature related issues, the increasing CPU frequency to improve performance is not a viable option. Possible alternatives for this problem are many core processors, specialized accelerators and re-configurable architectures [1]. Accelerator specific coding demands significant time and skill. Moreover, mapping of kernels to an accelerator is an iterative process before a “near” optimal mapping can be obtained. To reduce the number of iterations, it is better to predict approximate performance of the kernels in advance. Estimation of performance can be done either using developed empirical expressions or by simulation. Some mathematical models [4] have been used for estimating performance on traditional GPUs. But there is no prior work, to the best of my knowledge that model the execution time of GPUs with cache. As modern GPUs have cache hierarchy and as the proprietary information on the internal details of the architecture is not shared/publicly available, this task becomes even more complex.

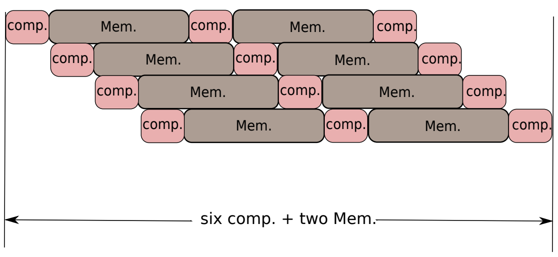
Different accelerators exhibit different ways of mapping for the same kernel. Map-Reduce (MR) is dedicated to processing large distributed data sets and mainly used for web applications [2]. MR framework works in two phases; map phase and reduce phase. Map phase is responsible for the execution of kernel in parallel on all cores of GPU. Reduce phase may not have as many instances as map phase but they are responsible for accumulating the output of map phase into final result. All inputs and outputs of map phase and reduce phase are in the the form of Key/Value pair. This makes the model quite generic. It provides a uniform interface structure irrespective of the application [3].

Our study here presents comparative merits and demerits of MR framework for different GPU architectures. We have studied and experimented with the MARS MR framework [3] developed for traditional GPUs before exploring its behavior for the Fermi architecture. We have implemented two optimization strategies and achieved significant performance improvements. The GPUs 9800GT, Quadro600 and GTX590 are used in our experiments.

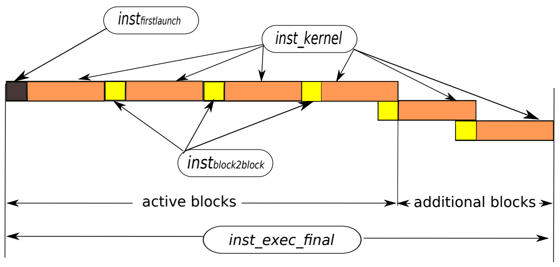
2 Methodology

First we explain the methodology for the performance estimation for GPUs with cache. We have prepared a variety of test cases and analyzed the results [5]. This work is divided into two parts; first we calculate computation cycle and

[1]Arun Kumar Parakh, Dept of Computer Science and Engg, Indian Institute of Technology Delhi, New Delhi, India, aparakh@cse.iitd.ac.in



(a) Basic theme



(b) Instruction count

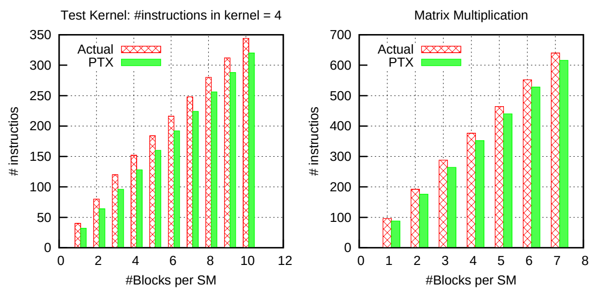
Figure 1: GPU execution style

$$Exec\_cycles = Mem\_cycles + Comp\_cycles + \frac{Comp\_cycles}{\#Mem\_inst} \times (N-1) \tag{1}$$

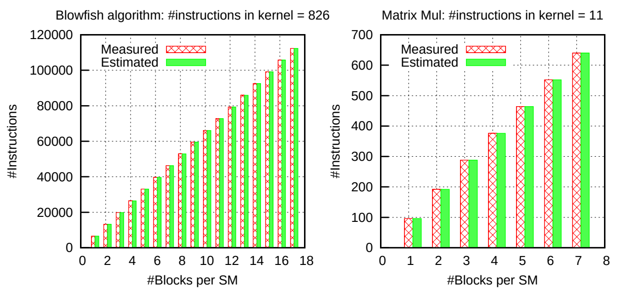
$$Comp\_cycle\_single\_type = inst\_exec\_final \times inst\_clock\_cycles \tag{2}$$

$$Comp\_cycle\_multi\_type = Comp\_cycle\_ker\_sched - warps_{block} \times blocks_{additional} \tag{3}$$

$$Mem\_cycles = \frac{\sum_{WarpID=1}^{\#Warps}(\sum_{mem\_inst=1}^{\#mem\_inst}(MAX\_latency)_{mem\_inst})_{WarpID}}{\#Warps} \tag{4}$$



(a) PTXvsActual



(b) Blowfish and MM

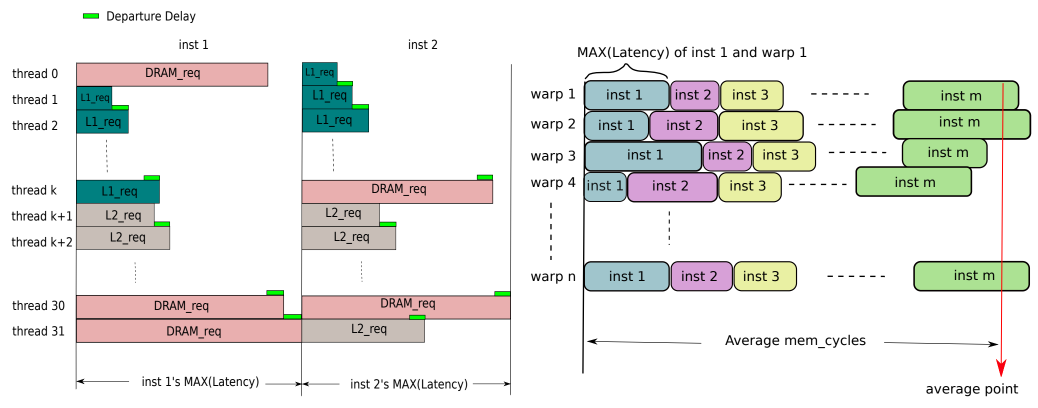
Figure 2: measured vs estimated instruction count

then calculate memory cycles. Figure 2(a) shows that instruction count from PTX file is different from actual count. In computation cycles, we have devised an empirical formula to get the exact instruction count. This formula is verified for more than fifteen micro-benchmarks as well as three real life applications. The applications are blowfish algorithm, matrix multiplication and image smoothing. Results of some of them are presented in Figure 2(b) and our results show an exact count. Instruction count model is shown in Figure 1(a)&(b). It demonstrate the number of instructions that are encountered in different phases, like kernel launch, block execution, block to block transfer etc. We have formulated this in eq. 2 and eq.3.

Memory access time is calculated from memory cycles for all memory instructions in the kernel. As per the memory access pattern and GPU execution style, we have devised various expressions to calculate memory cycles and final expression is shown in eq.4. We have also developed a in-house cache simulator for GPUs and the same is also verified with dineroVI. Figure 3 demonstrates the memory cycles required for executing a kernel on multi threaded SIMT architecture.

3 Results of performance estimation

Results of three different applications are presented in Figure 4. Blowfish algorithm has two parts; one is encryption and the other is decryption. The block size considered here is 256 threads. This is fixed due to the requirement of shared memory for storing P and S arrays. In this case we have over-estimated the execution time with a constant ratio. It is because we assumed that computation period is uniformly distributed over the whole warp execution time. But



(a) Memory Latency Test (b) Computation of average memory cycles

Figure 3: Memory Cycle Count

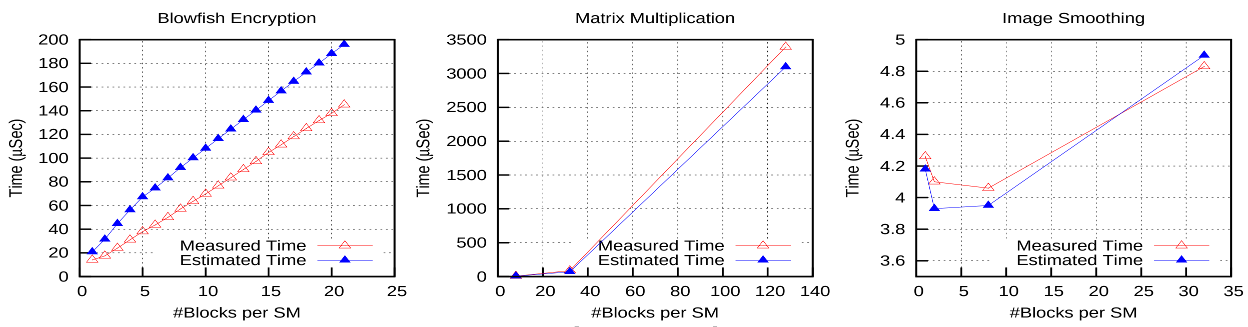


Figure 4: measured vs estimated time comparison

in actual execution initial computation period is very small and a large amount of computation period is sandwiched between two memory requests (Figure 1(a)). Results of Matrix multiplication and Image smoothing are more accurate. In these two applications memory access density is more as compared to Blowfish and this ensures that the effect of non-uniform distribution of computation period has less impact.

4 Performance analysis of Map-Reduce framework

Performance enhancement of map-reduce framework is the second part of my research. We have used MARS MR framework [3] for the analysis of behavior of different GPUs. Important phases of MARS are $mapperCount$, $prefixSum$, $mapper$, $group$, $reducerCount$ and $reducer$. We have used eight different applications to perform the study and experiments for the MARS on different architectures. These applications are Inverted Index (II), Matrix Multiplication (MM), Page View Count (PVC), Page View Rank (PVR), Similarity Scores (SS), String Match (SM), Word Count (WC), and Image Smoothing (IS). Due to space constraints, results of only three applications are presented here.

5 Experiments and Analysis

We have performed different sets of experiments to compare the architectures, explore the MARS and realize the impact of Fermi architecture. All the applications have implemented on 9800GT, Quadro600 and GTX590 GPUs with the help of MARS MR framework. Figure 5 shows the execution times for three different applications. The x-axis of the graphs represent the size of the data used in the applications. The Quadro600 GPU is on an average 1.8x slower as compared to the 9800GT. On the other hand GTX590 is on an average 5x faster as compared to Quadro600 and on an average 2.5x faster than 9800GT. Variations in the architecture have imposed both positive as well as negative impact on the performance.

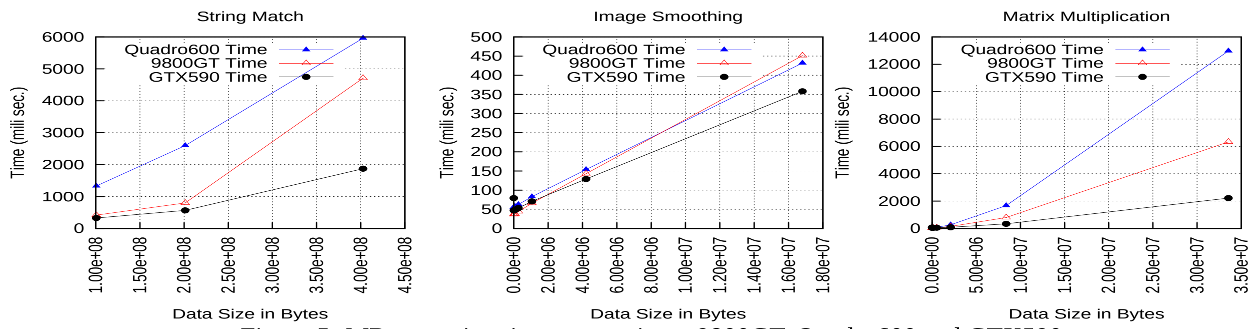


Figure 5: MR execution time comparison: 9800GT, Quadro600 and GTX590

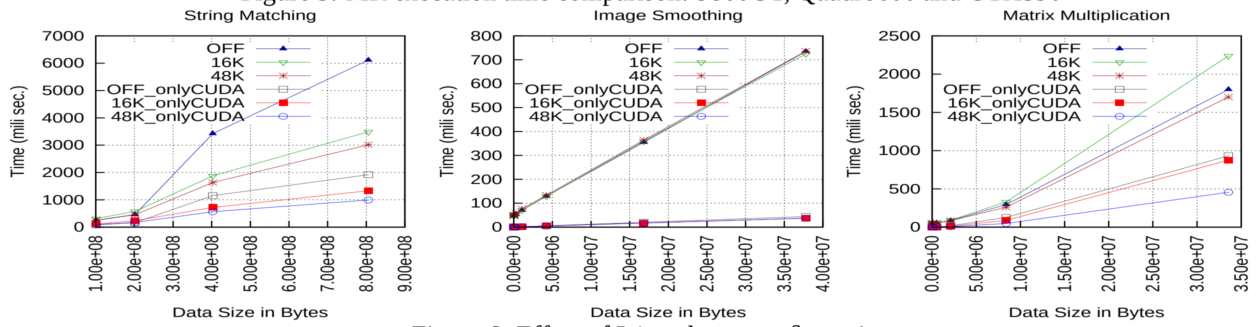


Figure 6: Effect of L1 cache re-configuration

6 Effect of L1 re-configuration

L1 cache of Fermi GPUs can be configured in three different ways. L1 can be switched 'OFF' at compile time and/or size of L1 can be set to 16KB or 48KB at run time. Effect of L1 configuration is shown in Figure 6. Performance variation ranges from 10% to 200% with different configurations for different applications. L1 with 16KB size results in worst performance while 48KB size and switched 'OFF' L1 are giving nearly equal performance boost. One possible reason is that the conflict cache misses are reduced significantly for 48KB with respect to 16KB. It is easy to conclude that the size of memory access and density of memory accesses are the key deciding factor for the selection of proper configuration of L1 cache. Our performance metrics and results are very helpful in the selection of the best configuration for implementers of these applications.

7 Performance enhancement of MARS

MR framework gives a generic application port mechanism but at the cost of the performance. Two techniques have been developed for the performance enhancement in MARS MR framework. One is focused on "group phase" and other is targeted to the "auxiliary functions". Objective of these techniques is to get benefit of new GPU technologies. Each technique is explained in the following subsections,

7.1 Group phase enhancement

It was observed from the profiled data that group phase contributes between 43% and 90% to the over all execution time of the application. The comparison part of bitonic sort in group phase took more then 90% of group phase time for string comparison. The original algorithm of string comparison, character wise comparison is used to implement string comparison. Therefore total number of comparisons are equal to the number of characters in the smaller strings and same number of memory requests are generated. In our methodology we compare two strings using packed data of four bytes. We read data as a chunk of four bytes and assign to an integer variable. We have saved approximately 75% comparisons. Figure 7 shows that a maximum 2x speed up and average of 1.5x speed up has been achieved.

7.2 The auxiliary functions enhancement

The $mapperCount$ and $reducerCount$ functions are treated as the auxiliary functions as they are not part of application algorithm but they play an important role in the map and reduce phases of MARS execution. These functions have three consecutive read and write instructions. This means a total of six instructions in which read and write operations are executed alternatively. When a read instruction is executed through a warp, one miss can bring data for

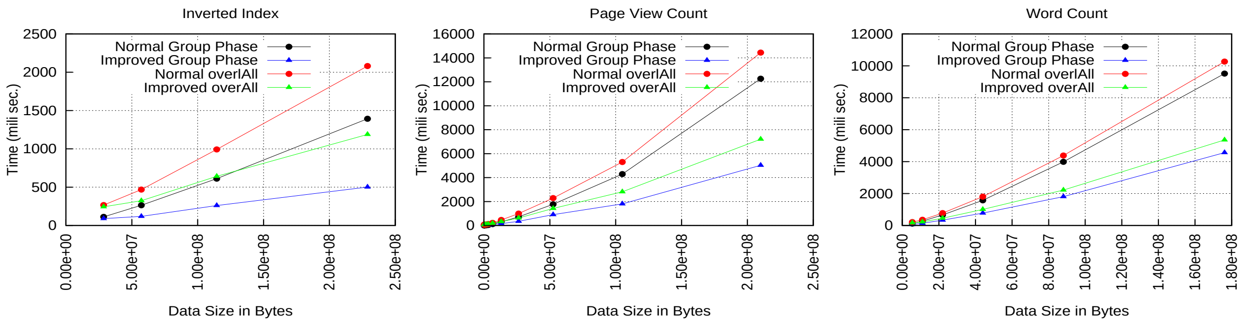


Figure 7: MARS Enhanced by optimizing group phase

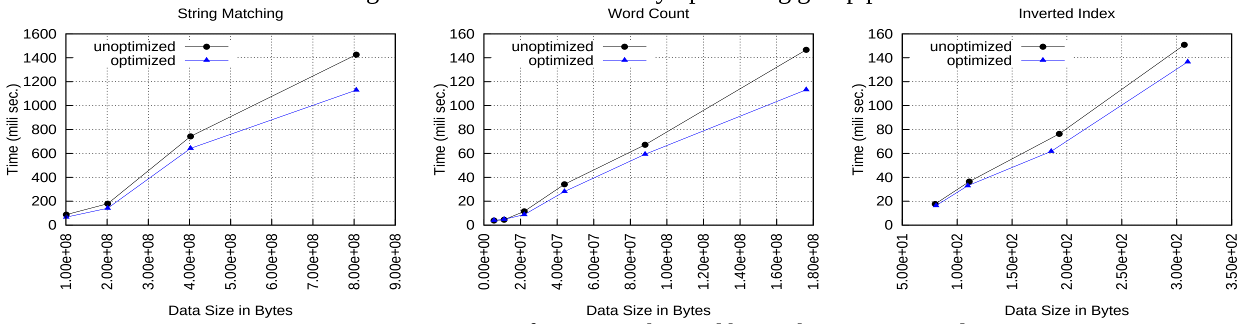


Figure 8: mapperCount function enhanced by cache sensitive coding

all other threads of that warp for this read instruction. But next instruction is write which makes the whole line dirty and remaining threads of the warp again generate cache miss. Instead of writing alternatively, we have used delayed writing for these functions. First all read operations are performed into the shared memory. Finally we write the result into the global memory. By this way we have saved around 93 cache misses per thread in place of 96 cache misses per thread. We use synchronization barrier so that all read operations must finish before starting of the write operation. Figure 8 shows the time taken by $mapperCount$ function for unoptimized and optimized code.

8 Conclusion

Modern GPUs are integrated with many heterogeneous platforms as accelerator. Our performance estimation model will be very useful for estimating performance of large applications, specially when application partitions are required to be ported onto a heterogeneous platform. The average estimation errors for the three selected applications, namely blowfish, matrix multiplication and image smoothing were 55%, -7.76% and 1.8% respectively (IPDPSw'2012 [5]). Our study explains the possibility of exploiting the features of new GPU architectures for MR framework. Our results clearly demonstrate that the MR can be used universally for different kind of applications with minimum amount of performance penalty. Developers have to concentrate only on improving the algorithmic parts of the application and deployment of algorithms on distributed architectures is taken care of by MR framework. Our performance enhancement technique performed for MARS on FERMI architecture helps to reduce the performance gap between CUDA implementation of the application and MR implementation.

References

[1] Ben Cope, Peter Y.K. Cheung, Wayne Luk, and Lee Howes. Performance comparison of graphics processors to reconfigurable logic: A case study. IEEE Transactions on Computers, 59.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplied data processing on large clusters. OSD2004, 2004.

[3] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques.

[4] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. SIGARCH Comput. Archit. News, 37, June 2009.

[5] Arun Kumar Parakh, M. Balakrishnan, and Kolin Paul. Performance estimation of gpus with cache. In Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSw), 2012 IEEE 26th International, pages 23842393, may 2012.