GPGPU Workload Analysis Based on CUDA Kernels

Ashraf E. Suyyagh

GPGPU Workload Analysis Based on CUDA Kernels

Ashraf Suyyagh
mrsuyyagh@gmail.com

Abstract

The recent architectural trend of mutli-core and many core processing has dominated the world of commodity computing where such state of the art designs are found in the CPUs and GPUs of every modern computer. Moreover, general purpose computing on GPUs is taking advantage of already massively parallel, multithreaded processing cores to achieve orders of magnitude speed up over state of the art CPU counterparts. This simulation project explores architectural design space flexibility of GPUs and investigates which design choices affect peak performance in an attempt to give an insight for tomorrow's designs. A workload of five non trivial non-graphical general purpose applications of scientific and financial nature is run on a detailed-cycle research simulator which runs CUDA kernels. Many design choices are explored and their performance impact is analyzed, the design options are limited to the streaming multiprocessor units in which the number of processing shader units, register count, shared memory size is studied. Furthermore, the number of threads per thread blocks run on a streaming multiprocessor and a thread scheduling option is also investigated.

1. Report Organization

This report starts with presenting a quick design abstract of modern NVIDIA GPUs based on the TESLA architecture, and then presents the baseline configuration of the GPU design to which all further design choices are compared, such design options are listed and elaborated on in section 2. The research Simulator GPGPU Sim is introduced in section 3. The workload set is introduced in section 4 where each application is fully explained, the section ends with a summary of the properties of each application. Section 5 provides the simulation results of the design choices studied and brief analysis is given of the cases in hand. Finally, the report concludes with an overall analysis.

2. Simulation Options

NVIDIA GPUs based on the current TESLA architecture offer multiple scalable Streaming Multiprocessors (SMs), each encompassing a group of processors (also called shaders, or shader processors). Each SM has its own shared memory block and a set of registers. Threads are issued in thread blocks and executed to completion without preemption by an entire SM. This simulation attempts to vary these parameters and investigate possible performance gains or losses.

The baseline configuration of the GPU core used in this simulation consists of 28 streaming multiprocessors each encompassing 32 processing cores for a total of 896 cores with eight memory channels. Each SM has 16kb of shared memory and 16384 registers. This base configuration allows for up to 1024 threads in each thread block and for coarse-grained parallelism on the thread block level, with up to eight thread blocks per SM running concurrently (In contrast to the current NVIDIA TESLA GPU architecture where eight cores per SM are implemented, only up to 512 threads per thread block is allowed and coarse grained parallelism on the thread block level is prohibited by the CUDA model!). Additionally, no L1 or L2 caches are used! Table 2 summarizes the base configuration and presents the several design options explored.

Hardware Simulated	Basic Configuration	Different Configurations Simulated
No. of Streaming Multiprocessors	28	-
No. of processors per SM	32	8/16/32
No. of threads in thread block	1024	512/1024/1536/2048
No. of registers / SM	16384	4096/8192/16384/24576/32768
Shared memory size (bytes)/ SM	16384	16384/24576/32768
No. of concurrent thread Blocks	8	4/8/12/16

Table 1: Basic Configuration and Simulations Parameters Explored

In this simulation project, one architectural perspective which was explored was the ratio in between the thread warp size issued to the number of the general shader processor count in each SM. Current GPU designs from NVIDIA issue a warp size of 32 threads onto eight shader cores at a quadrupled rate, thus executing a warp per cycle. I have further explored the two possibilities of doubling the number of cores and scaling their numbers to match that of the warp size! Furthermore, based on the previous results, another option of selecting a warp size of 16 instead of 32 has been simulated while simulating higher and lower options was limited by the simulator capabilities.

Moreover, varying the number of registers and the total size of the shared memory per SM was investigated. In addition, increasing the thread count per thread block was further studied! To elaborate more, further simulations were conducted to analyze speed up gains when increasing the thread count and scaling the SM hardware resources accordingly!

A final simulation aspect was to analyze the variation of the maximum allowed number of concurrent thread blocks to be executed by a single SM.

3. Simulator

GPU Simulators for general purpose computing are very scarce and few were developed within the past couple of years due to the novelty of the domain, moreover, most implementations are not yet mature and many are only in their beta stages, most notably to mention are: Barra, Ocelot and GPGPU SIM [3]. For the purpose of this simulation project, the GPGPU SIM simulator (which is developed at the University of British Columbia) was chosen for it provides detailed statistical results and allows for much wider range of design and simulation options in comparison with the other mentioned simulators. GPGPU SIM is a detailed cycle performance simulator for many core architectures, specifically GPUs running CUDA code, this means that it provides a cycle level model for every part of the microarchitecture in contrast to cycle accurate simulators, that is it does not match the hardware 100% but provides very close estimates [4]

GPGPU SIM offers two types of simulation: functional and performance. In functional simulation, the architecture, the instruction set and the overall accuracy is simulated. On the other hand, performance simulation deals with the timing model; that is GPGPU SIM reports the number of cycles spent by running the CUDA kernels, it covers shader cores, caches, interconnect network, memory controllers and graphics DRAM.

4. Workload

No official benchmark suite has yet been developed for GPU general purpose computing simulators. Researchers tend to either use some of the complex CUDA programs provided by the NVIDIA SDK [2] or compile their own workload which often varies in between general purpose computing domains; financial, graphical as well as scientific domains: mathematical, biological...etc. In this simulation project and for the lack of an official comparison baseline, I will use a subset of programs simulated and used by Bakhoda et al [1] in their work presented at ISPASS09. The workload set consists of:

Graph Algorithm: Breadth-First Search (BFS): an application which performs breadth-first search on a graph. Each node in the graph is processed by a different thread; therefore the amount of parallelism in this application scales with the size of the input graph. BFS suffers from performance loss due to heavy global memory traffic and branch divergence. The graph file provided describes a random graph with 65,536 nodes with 6 edges per node on average.

3D Laplace Solver (LPS): a highly parallel finance application which uses shared memory and ensures coalesced global memory accesses. It runs for one iteration on a 100x100x100 grid.

MUMmerGPU (MUM): a parallel pairwise local sequence alignment program that matches query strings consisting of standard DNA nucleotides (A,C,T,G) to a reference string for purposes such as genotyping, genome resequencing, and metagenomics. The reference string is stored as a suffix tree in texture memory and has been arranged to exploit the texture cache's optimization for 2D locality. Since each thread performs its own query, the nature of the search algorithm makes performance also susceptible to branch divergence.

N**eural Network (NN):** Neural network uses a convolutional neural network to recognize handwritten digits. Pre-determined neuron weights are loaded into global memory along with the input digits. This version is modified to allow for the recognition of multiple digits at once to increase parallelism. Nevertheless, the last two kernels utilize blocks of only a single thread each, which results in severe underutilization of the shader core pipelines. Recognition of 28 digits from the Modified National Institute of Standards Technology database of handwritten digits is simulated.

Ray Tracing (RAY): Ray-tracing is a method of rendering graphics with near photo-realism. In this implementation, each pixel rendered corresponds to a scalar thread in CUDA. Up to 5 levels of reflections and shadows are taken into account, so thread behavior depends on what object the ray hits (if it hits any at all), making the kernel susceptible to branch divergence. A rendering of a 256x256 image is simulated

Table 2 summarizes the configuration and characteristics of the above mentioned workloads:

Benchmark	Grid Dimension	Thread Block Dimensions	Concurrent Thread Blocks/SM	Total Threads	Shared Memory	Constant Memory	Texture Memory	Barriers
BFS	128,1,1	512,1,1	4	65563	Y	N	N	N
LPS	4,25,1		6	12800	Y	N	N	Y
NN	6,28,1 50,28,1 100,28,1 10,28,1	13,13,1 5,5,1 1,1,1 1,1,1	5 8 8 8	28392 35000 2800 280	N	N	N	N N N N
MUM	782,1,1	64,1,1	3	50000	N	N	2D	N
RAY	16,32,1	16,8,1	3	65563	N	Y	N	Y

Table 2: Workload Properties

5. Simulation Results

Since there are 896 processing cores in the base configuration, the maximum theoretical attainable IPC is therefore 896. Figure 1 shows the practical IPC values for the benchmark workloads used! The design options and speed ups are compared to these practical values.

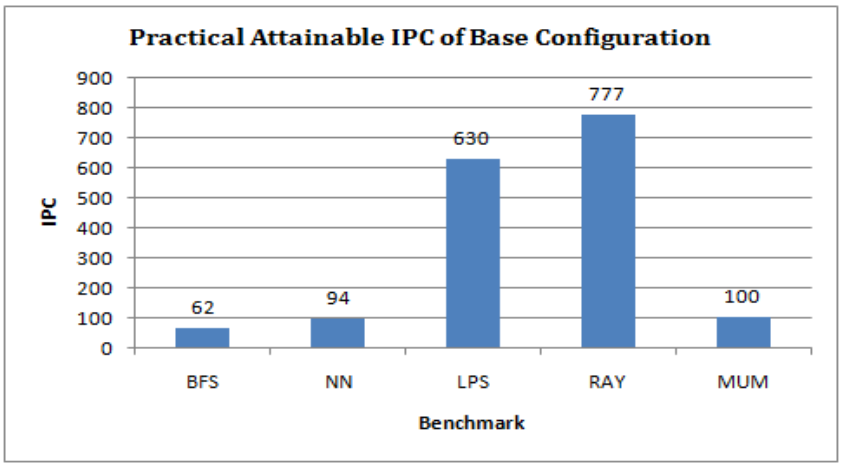

Figure 1

Figure 2 displays the results of the first design option to be simulated which was the relationship in between the warp size (Warp in CUDA terminology is the a unit of 32 threads scheduled to run concurrently in an SM) and the number of processor cores on which it is to be executed, the number of processors was selected to be ¼, ½ and equal to the warp size, or in other words, simulating the speed up gains when scaling the number of processing units in an SM. Current GPU hardware limits the number of processors per SM to eight due to high design and implementation costs.

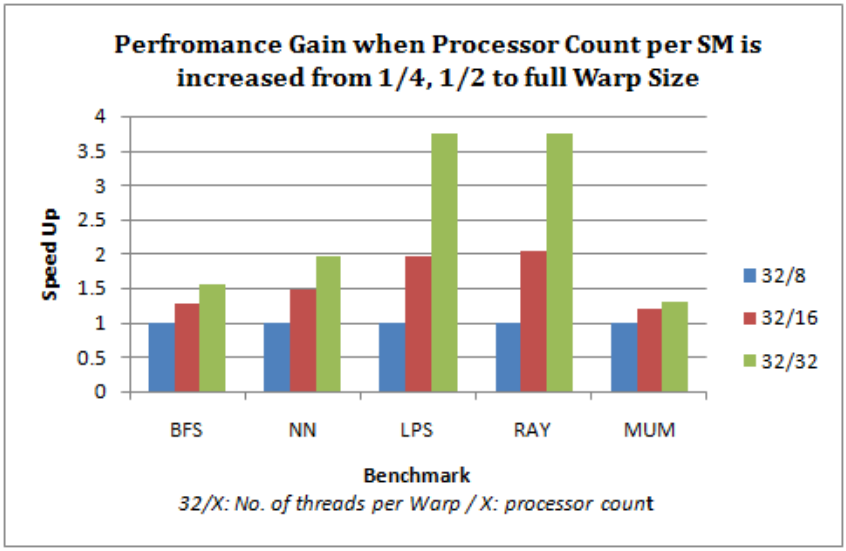

Figure 2

A speed up of two and four was expected when the number of processors was doubled and quadrupled, however, this was not the case for all the workload benchmarks since not all applications

are inherently and completely parallel! For example, the nature of the BFS program allows for branch divergence and thus threads are executed sequentially! The NN application has certain sequential code which cannot be parallelized!

The second option to be explored was varying the number of registers per streaming multiprocessor. Figure 3 shows that no substantial speed up gains were realized except for the RAY application and only in the initial jump from register file size of 4096 to 8192 bytes. This might be due that this increase in physical registers size surpasses the resource amount needed by the applications.

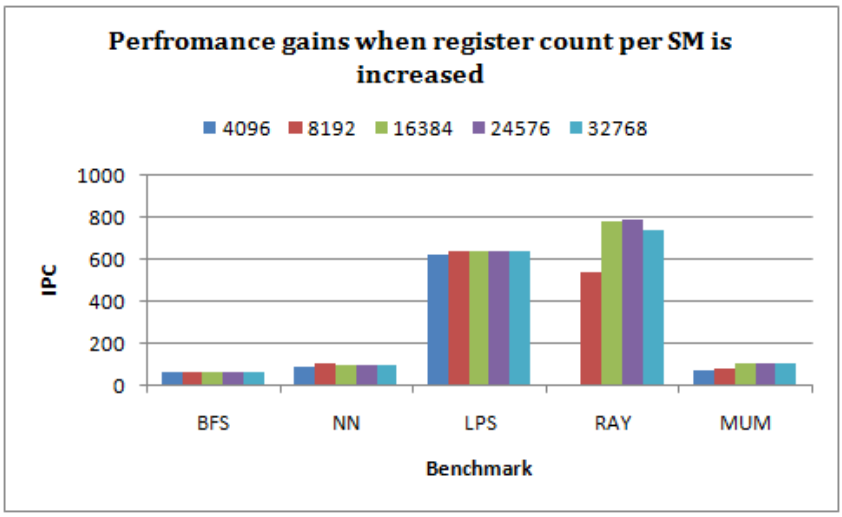

Figure 3

In addition, the size of the shared memory in between all threads issued to the same SM unit was investigated! Memory sizes of 50%, 125% and 150% was put into test. No speed up gains were observed at all! Such a result was expected for the NN, MUM and RAY workloads for they do not utilize this memory! Interestingly however, that the LPS and BSF benchmarks showed no improvement though they extensively use this shared memory. This might also due to the fact that the size amount of shared memory is well beyond applications needs! Refer to Figure 4.

Therefore, based on the abovementioned results and observations, another set of tests was conducted in which the number of threads in a thread block which is scheduled to run in an SM was varied from 512 threads to 2048 threads and to explore if the extra resources of shared memory and registers are exploited! Figure 5 shows the recorded results. No speed up gains were recorded however. A feasible explanation in this case is that performance is being limited due to global memory access and interconnect congestions. [1]

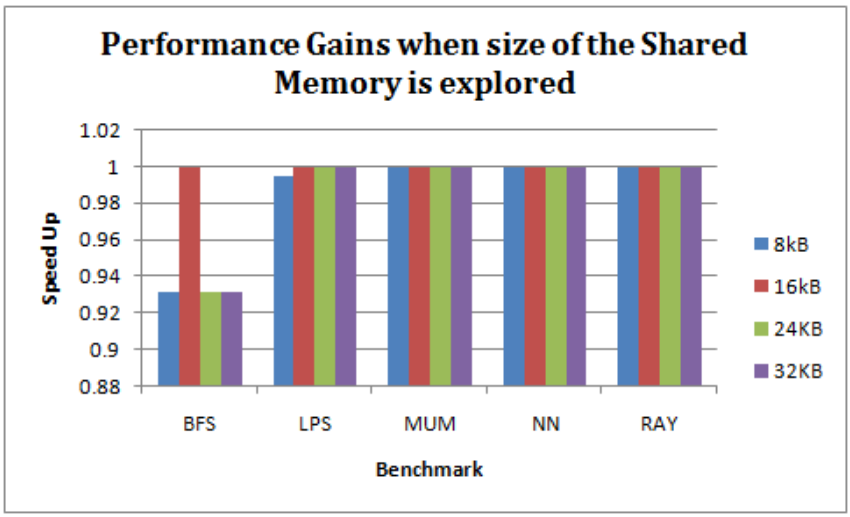

Figure 4

A final simulation conducted was varying the number of concurrent thread blocks allowed to run in one SM. Figure 6 shows that the NN application shows increasing gains as the level of thread block level parallelism is raised. In addition, the MUM application only shows one step of improvement

when the number of concurrent TB is doubled from 4 to 8 but no further performance is gained when additional concurrent TBs are allowed to execute. Referring back to table 2, it is shown that this increase is due to the fact that these benchmarks are by design written to support being scheduled in a concurrent fashion with up to eight TBs, meanwhile the others are limited to three TBs.

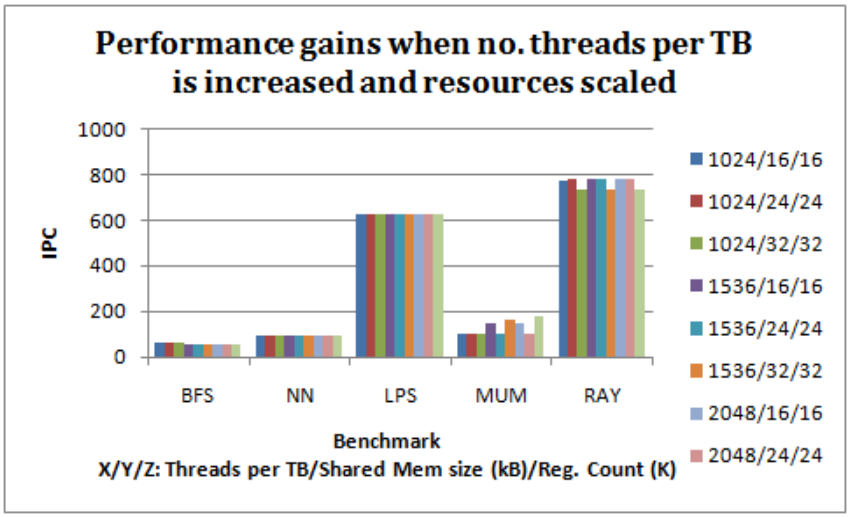

Figure 5

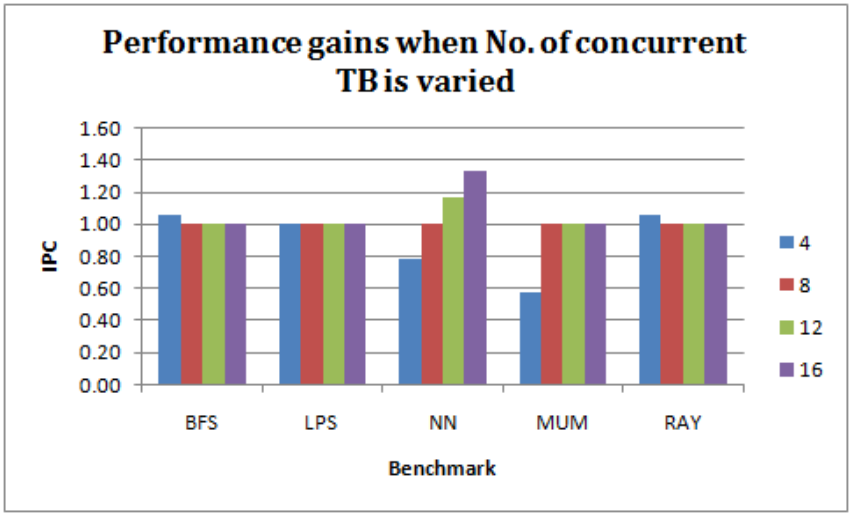

Figure 6

6. Discussion and Conclusions

Several design space options have been explored in this simulation project in relation to the parameters of the streaming multiprocessor. It has been found that increasing the numbers of shader processors per SM does not necessarly unleash speed ups that are linearly proportional to the increase in processor count even when the number of threads issued per thread block is also raised. This is due to the fact that not all workloads are inherently and completely parallel where certain segments of the kernel are to run in a sequential fashion. Moreover, applications with high control flow instructions with high probability of warp divergence might further hinder performance gains depending on divergence handling policy, that is when divergent threads are executed in sequential order and not taking advantage of the parallel design, no speed up gains are expected and use is made of the scaled shader processor count.

When the number of registers per SM and size of the shared memory is increased, most simulated programs performance remained unchanged. This result was expected for applications which do not make use of the shared memory, minor performance gains were recorded for some applications when register count was increased, others remained unchanged. A possible explanation was that increase of resources surpassed the needs of the workload. Further studies on resource extensive and greedy applications are to be analyzed in order to draw a firm conclusion.

Increasing the number of threads per thread block was expected to further enhance performance, yet no gains were observed, scaling shared memory and register count with thread number increase had no effect. Careful analysis suggested that though the results did not match our expectations it was due to global memory access congestion and DRAM channel controller stalls. Memory was the bottleneck in this case.

Scheduling thread blocks to execute on each streaming multiprocessor unit in a coarse grained fashion was dependent on the nature of the workload; that is the number of independent thread blocks found in each application, the higher the independency the higher the gains. This results was consistent with the results obtained by Bachoda et al [1]

References

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong and T. Aamodt, "*Analyzing CUDA Workloads Using a Detailed GPU Simulator*" IEEE International Symposium on Performance Analysis of Systems and Software, Boston, Massachusetts, 2009

[2] NVIDIA CUDA Website: www.nvidia.com/cuda

[3] GPGPU SIM website: http://www.ece.ubc.ca/~aamodt/gpgpu-sim/

[4] A. Bakhoda, W. Fung, H. Wong and T. Aamodt, *'Tutorial on GPGPU-Sim: A Performance Simulator for Massively Multithreaded Processor Research"*, The 42nd Annual IEEE/ACM International Symposium on Microarchitecture, NY, December 2009