Journal of Computer Science 10 (9): 1811-1818, 2014
ISSN: 1549-3636

doi:10.3844/jcssp.2014.1811.1818 Published Online 10 (9) 2014 (http://www.thescipub.com/jcs.toc)

HYBRID IMPLEMENTATION AND PERFORMANCE ANALYSIS FOR HIGH PERFORMANCE COMPUTATION WORKLOAD

Joseph Issa

Department of Electrical and Computer Engineering, Notre Dame University, Lebanon

Received 2014-03-03; Revised 2014-03-31; Accepted 2014-04-26

ABSTRACT

Given the need to achieve maximum performance possible, offloading intensive computation workload to GPU is a key to achieve this goal. Offloading most of the workload to GPU may not results in desired performance, so a middle approach is more suitable such as splitting the workload between the CPU and the GPU can be considered as an optimized approach. In this study, we used a popular high performance computation workload which can also be implemented using a hybrid approach in which part of the workload is offloaded to the CPU. We also present a performance estimation method which is verified to estimate performance with in 5% error margin.

Keywords: Performance Analysis, Hybrid Computation

1. INTRODUCTION

In general, the CPU was the focus when it comes to application with intensive computation requirements. Recent development of GPU introduced new features to handle intensive parallel computation which makes it compete with the CPU. While most of the high performance computation algorithm now focuses on GPU for data intensive computation, there is still a small window for the CPU to perform parallel task with the GPU which leads to hybrid implementation. Although there are big architecture difference between the CPU and GPU, we can still think of both of them sharing the same architecture space with different parameters. For example, the CPU comes with small number of cores compared to GPU, but the CPU core can deliver better single thread performance. On the other hand, the GPU can hide memory latencies by managing large number of threads while the CPU does this through cache. The GPU can't handle very well task parallelism, but it is more suitable to process data parallelism. The CPU can't handle data parallelism, but it can handle very efficiently task parallelism due to its efficient branching feature. In summary the CPU and GPU trade off one architectural feature for another, so it is reasonable to assume that some applications are more suitable for one or another. CPUs and GPUs are built using different approaches.

The CPU is designed for different applications and can provide fast response times to a single task. Architectural features such as branch prediction, out-of-order execution and super-scalar are directly related to this performance improvement. These features come at the expense of increased power consumption and complexity in addition to increase in die size per core.

GPUs are built for rendering and other graphics applications that have a large amount of data parallelism, which means that each pixel to be displayed on screen can be processed independently. CPUs on the other hand are designed to pack small number of processing cores while keeping within a given power and thermal limitations. This results in GPUs trading off single threads performance for increase parallel processing. CPUs can provide a better performance for single thread for throughput computing workloads. GPUs provide many parallel processing units which are ideal for throughput computing.

The paper is organized as follows; we start in section 2 with related work section, in which we compare our work with different published papers. In section 3, we present the Monte Carlo (MC) performance model and Hybrid implementation with performance analysis. In section 4 we discuss experimental results for the estimation model as well as the hybrid performance implementation data results. In section 5, we conclude and discuss future work.

2. RELATED WORK

In this study, we propose an analytical model to estimate performance for MC benchmark with error<5% between measured and estimated data. We also present a hybrid implementation for MC in which the workload is shared between the CPU and GPU. Several researchers have worked on estimation processor performance for given benchmarks using different estimation methods including simulation trace-based methods. Our performance estimation models identifies performance dependencies and bottlenecks for a given processor and workload. We also present a hybrid implementation for MC workload to maximize performance. The model can be used to estimate performance for different processor settings (i.e., frequency, number of cores, Instructions-Per-Cycle (IPC), efficiency and execution time).

Goel *et al*. (2010) presented a per-core linear power model using sampled performance counter on single and multithreaded applications. Error deviation is <5% for all tested workloads.

Bakthavatsalam and Mehata (2014) proposed a Hybrid instruction set implementation, as compared to our method in which the hybrid implementation between CPU and GPU is implemented within OpenCL code itself.

Pennycook *et al*. (2011) presented a hybrid model of MPI and CUDA for NAS-LU benchmark and compares it to different processors and GPU architectures. Our MC hybrid design approach uses OpenCL code to use parallel implementation in which part of the computation task is off-loaded to CPU while GPU is running other computation task in parallel.

Aoki *et al*. (2011) presented Hybrid OpenCL implementation for multiple nodes in network environment. The concept is similar to what we presented in this study between GPU and CPU, but in (Aoki *et al*., 2011) the performance is compared between Hybrid Open CL and OpenCL with MPI implementation.

Yu *el al.* (2014) developed a new simplified computation method based on new parallel computation techniques in which computation time is minimized. In this study, we propose a different approach by offloading computation load to GPU when it is possible to offload.

3. PERFORMANCE PREDICTION MODEL

In this section, we derive a set of equations for the MC benchmark performance model. We collected most data using the CUDA profiling tool provided by Nvidia. These equations are based on generic GPU architecture but are also specific to benchmark behavior. The number of warps is generally the total number of threads divided by 32. Applying this to the benchmark, we get the following Equation 1:

$$\text{warp\#}=\frac{M*256}{32} \quad (1)$$

where, M is the number of options, loop_per_warp_thread is defined as Equation 2:

$$\text{loop\_per\_warp\_thread}=\frac{N}{1024} \quad (2)$$

where, N is the simulation path. Using the CUDA profiling tool, we analyzed the logic loops through instructions issued per warp as shown in **Fig. 1** Equation 3:

$$\text{Instructions\_per\_warp\_thread} = 54.4\times\text{loop\_per\_warp\_thread} \quad (3)$$

And the cycles per warp thread can be calculated as Equation 4:

$$\text{cycles\_ per\_warp\_thread\_instr} = \text{per\_warp\_thread/IPC/Efficiency} \quad (4)$$

The total number of cycles which is needed to determine the total time is defined by Equation 5:

$$\text{total\_number\_of\_cycles} = \text{instruction\_per\_warp\_thread}\times\text{Warp\#/SM\#/(efficiency IPC)} \quad (5)$$

where, instruction per warp thread is defined by Equation 6:

$$\text{Instructions\_per\_warp\_thread} = 54.4\ \text{loop\_per\_warp\_thread}\ 74.64 \quad (6)$$

The final prediction equation for options per second is defined by Equation 7:

$$\text{BW (Options/sec)} = \frac{M}{\text{Total\_time}} \quad (7)$$

where, Total Time is defined as Equation 8:

$$\text{Total\_time} = \text{Total\_cycle} / \text{Core\_Frequency} \quad (8)$$

M = Defined as the number of options
SM# = Number of core shaders frequency is the GPU core frequency
IPC~1.0 = Efficiency ~0.92
N = The simulation path

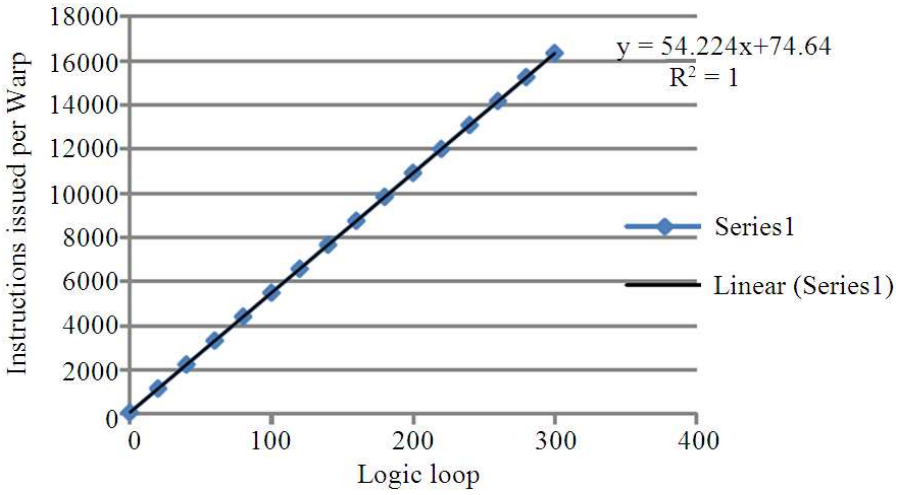

Fig. 1. Logic Loop Vs Instructions issued per Warp

From these equations, we see that the MC benchmark performance is directly proportional to number of cores, IPC, efficiency and core frequency and is inversely proportional to simulation path *N*, *warp#* and total time.

3.1. MC Hybrid Configuration

For high data intensive computation, the trend is to offload intensive computation to vector processor like GPU, compared to traditional approach of performing computation only on the CPU. Splitting the workload in-between the CPU and GPU might be a better alternative. In this section, we will discuss MC option pricing algorithm in a hybrid implementation. MC is a two block-processing paradigm, one that generates samples and second that so the actual processing. Traditionally the CPU was the platform of choice for computing application. Recent development in the GPU space introduced what seems like a competitor for the CPU. As a result, there is currently a tendency to overrate the utility of the GPU in computing applications, in the same way that workloads where processed on the CPU in the past, the current trend if to offload computation to GPU. There is a big architectural difference between CPU and GPU; each exhibits strengths in areas where other is weak. The CPU comes with small number of cores compared to GPU but each of its cores delivers better single threaded performance. The GPU hides memory latencies by effectively managing a large number of threads; the CPU does it through cache memories. The GPU can't handle very well task parallelism but it's very suitable for data parallel applications; the CPU may not handle as well data parallelism but it handles very well task parallelism due for example to the presence of an efficient branching mechanism. Cache memory takes up quite a lot of the CPU die but on the GPU it's less important. CPU and GPU trade off one architectural aspect for another, each specializing in another direction according to the demands of the market. It's reasonable to assume that certain applications are more suitable for one or another. For the purpose of this study all measurements where performed on an Intel i7-2600K CPU and on a GTX 480 NVIDIA GPU. All implementations where done in OpenCL 1.1 for both platforms. For the 2 devices 1 context implementation tests were performed only on Intel i7-2600K with an internal GFX driver build with support for GPU OpenCL enabled. In **Fig. 2**, we show offloading computation for GPU, which leaves small window of opportunity for CPU.

MC tries many scenarios and offers an estimation of the most probable outcome. The algorithm is composed of a random numbers generator software Pseudo Random Numbers Generator-(PRNG) and actual processing specific to the domain of application. The input is a set of options and the output is their corresponding expected values and confidence levels. The single device implementation is obvious. The implementation possibilities for a hybrid (CPU+GPU) architecture are done such that each device processes a chunk of the input set of options proportional with its compute capability were both PRNG and MC run on each device. Each device runs one of the MC components were one does PRNG and the other MC. Following measurements of the single device implementations we conclude that PRNG behaves

better on the CPU both in single and in double precision. In addition, MC apparently performs much better on the GPU than on the CPU (almost 100 times faster). In the MC case, we should note that the very large difference is mostly because the comparison done between the GPU with fast math enable and the CPU without fast math (not yet supported on the CPU). When fast math is disabled on the GPU or if the CPU algorithm is modified to use a fast implementation of the exponential function, the performance gap is reduced to about 3-6 times still in favor of the GPU. From the hybrid point of view, the choice seems to be to perform the PRNG on the CPU and after that MC on the GPU. Now from the point of view of the Open CL implementation and considering only two devices at a time, we constructed the below diagram to show hybrid options. We can have 2 discrete devices in 1-2 contexts or 2 joined devices in 1-2 contexts. (An example of 2 discrete devices is i7-2600K and GTX 480; an example of 2 joined devices is i7-2600K and HD3000 GPU). The advantages of single vs. multiple contexts is, we can share memory objects if both devices reside in the same memory space reducing overhead and we can use events to synchronize between executions on all queues included in the same context. Considering these facts, it seems better to go for the single approach where possible. A crucial element of the hybrid implementation is the ability of the common Open CL API 'enqueue' functions to execute asynchronously-in other words to return control back to the host thread immediately after being issued. This is shown in **Fig. 3** where a problem was broken into 3 chunks.

In **Figure 3** graph A shows the execution duration if chunks are processed sequentially (even though PRNG takes place on the CPU and MC on the GPU). Graph B shows the performance gain of achieving parallel execution on both the CPU and the GPU when compared with the A case. The second approach is also very useful to hide memory IO overhead when the CPU and the GPU do not share the same memory space. In the A case every call is blocking and so, even if we have PRNG execute on the CPU and MC on the GPU, the chunks are executed sequentially. Having the calls execute asynchronously enables parallel execution on both devices and more efficient use of the available hardware resources. In the hybrid model addresses only the case of two devices and it is suitable for both single and dual context. The main difference between single and dual context is how the memory transfers are implemented; for the single context, approach explicit memory transfers are not implemented. For dual context memory, transfers are done through a combination of memory enqueue calls. The core to our hybrid design is running the PRNG on the CPU and MC on the GPU as shown in **Fig. 4**. Explicit memory transfers belong on the GPU Q. The problem gets broken down into N chunks and the algorithm loops over the problem in N+1 steps. On the first step only the CPU Q performs. On the last step only the GPU Q performs. Basically we issue asynchronously the enqueue calls (kernel and memory) to the CPU and GPU queues and then synchronize on each step.

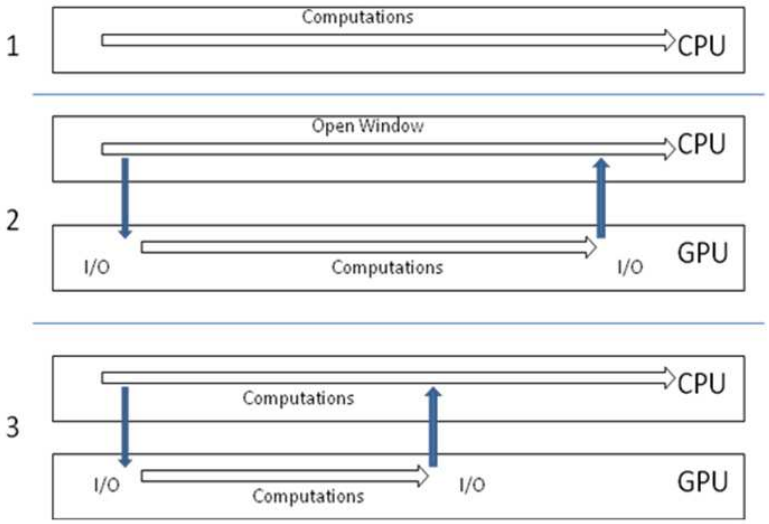

Fig. 2. Offloading computation to the GPU

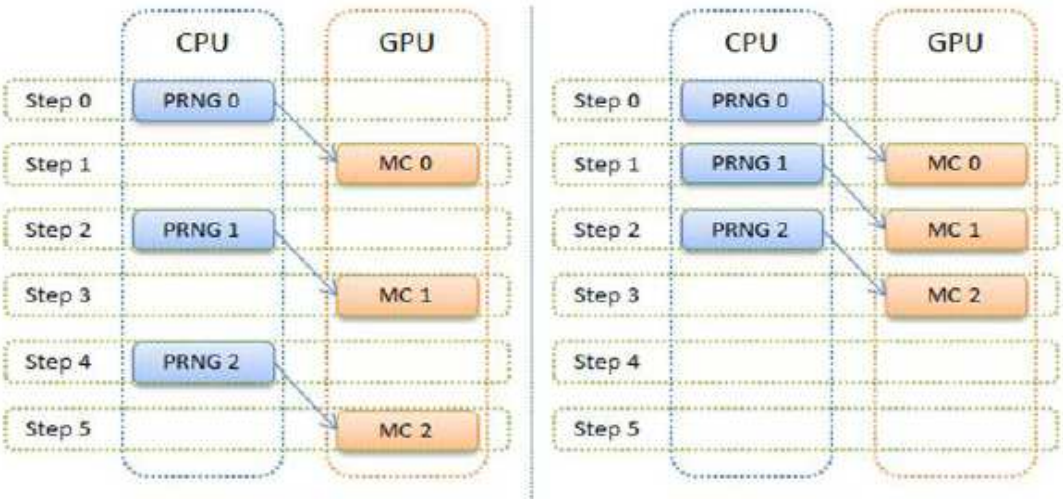

Fig. 3. Sequential versus parallel execution

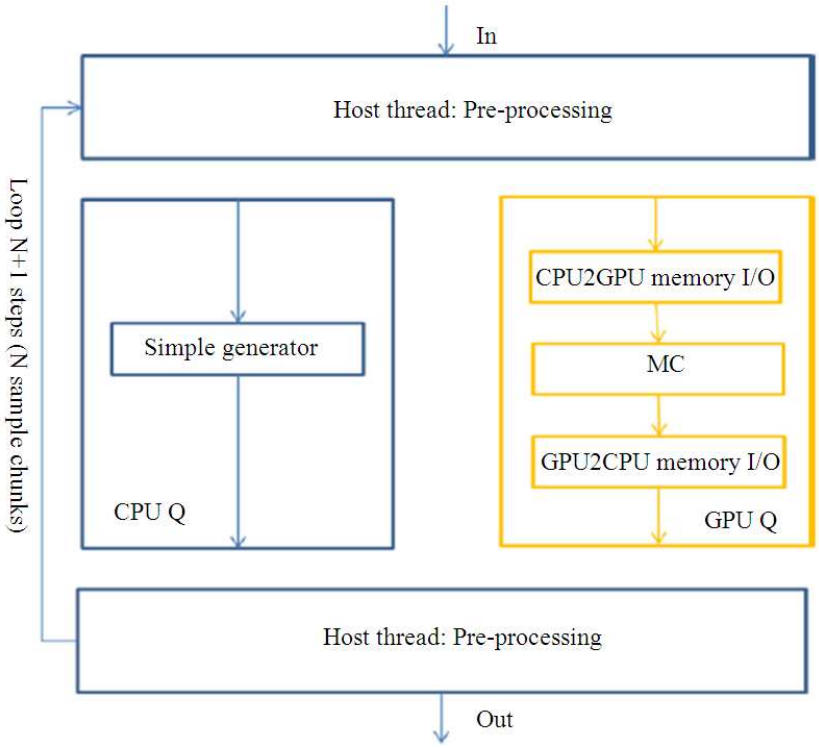

Fig. 4. MC Hybrid design

After synchronization, additional post-processing is required to integrate results over the whole sample set. The sample set is what needs to be transferred towards the GPU as it is input for the actual MC processing.

The measurements show that hybrid performance depends heavily on the problem size (sample set size and option count) and on the CPU/GPU ratio of compute capabilities. The problem size matters because it is the

main factor for workload balancing. Improperly balancing the two stages of the pipeline (CPU and GPU) leads to wasting compute power and memory transfers not being hidden. CPU/GPU compute ratio is also important because if it is too small can lead to a too small performance gain to make hybrid worth implementing. The original PRNG implementation, the one for the GPU, performed very poorly on the CPU (in single precision, almost 40 times slower on the CPU) so it was necessary to be redesigned. The best approach was to first make a sequential implementation for the CPU and then convert that code to be used as a kernel (so on several threads) in OpenCL-an approach similar to implementing for Message Passing Interface (MPI). The number of threads is not that important on the CPU as it is on the GPU-and this was to be expected as the CPU single thread performance is much better than that of the GPU single thread performance.

The MC hybrid performance model is divided into two categories The GPU only time and the Hybrid time for GPU and CPU. We experiment the model for single precision and double precision for different options and sample sizes. We start with GPU time equation, which is given by Equation 9 and 10:

$$GPUTime = (RandTime + MCTime) \times N_{GPU} \quad (9)$$

$$Hybrid\ Time = T \times N_H \quad (10)$$

We assume the I/O time is negligible ~0 and MC Time = T for one pipeline step duration. The hybrid time equation is given by Equation 11 to 13:

$$Hybrid\ Time = T \times N \quad (11)$$

Where:

$$N_H\ N_{GPU} + N_1 \text{ and } N_{GPU} \quad (12)$$

And:

$$T_{GPU} = Rand\ Time \times N + MC\ time \quad (13)$$

The difference between GPU and Hybrid time is derived as Equation 14:

$$GPUtime\text{-}HybridTime = RandTime \times N\text{-}T \quad (14)$$

In the results section, we implement the equation derived in this section to calculate the GPU and hybrid time for different options/sample size to derived the benefit for using hybrid model instead of just GPU for single and double precision floating point. Hybrid performance depends on many factors, such as problem size, load balancing, hiding memory operations using parallelism, algorithm optimization and capabilities of devices which are best when the devices are well balanced. The bigger the processing power gap between the devices the less performing hybrid will be.

4. EXPERIMENTAL RESULTS

4.1. MC Performance Experimental Results

First, we verify MC using the Nvidia Tesla 2050 (CUDA cores = 448, or SM# = 448/32 = 14) and NV GTX580 (CUDA cores = 512 or SM# = 512/32 = 16) graphics cards. The data results show an error of<5% between the estimated and measured data at different core frequencies and numbers of cores, shown in **Fig. 5**. For both cards (Tesla 2050 and GTX580), the simulation path N = 256*1024 and options number M = 2048.

For MC Hybrid model performance results, solving the equations we derived in hybrid model section, if we know the RandTime for CPU and T, we can calculate N for which there is a performance gain. For the single precision case, T~65%* RandTime, which means that N is at least 2, this will give us theoretical performance gain of 35%.

From **Fig. 6**, the random time for CPU reaches a max of 2863ms for number of samples = 27, while the GPU it reaches a max of 1844 ms for samples = 27. Therefore, the CPU RAND time is only 1.55 slower than the GPU RAND time. For 800 options and 128×1024×1024 samples, the GPUtime = 4883 ms and Hybridtime = 3526 ms which is ~27.8% improvement in performance. Performance can be increased by increasing the sample size, for example, Options = 800, Samples = 1024 MB, Hybrid Chuck = 16 MB and GPU chunk = 128 MB, we calculated Hybrid time to be 28.8% faster than GPU only time. If we change the sample size to 2048 MB, we calculate the hybrid time to be ~29% faster than GPU only time. We repeat the same experiment for double precision.

For double precision, the CPU Random generating numbers is 5× times faster than the GPU. From **Fig. 7**, CPU RAND time is~1360 ms, while the GPU RAND time is ~6950 ms. In case of 64 options, 120×1024×1024 sample size, the GPU time is calculated at 11419 ms while the Hybrid time 0is 5210 ms, this is ~54% improvement in performance.

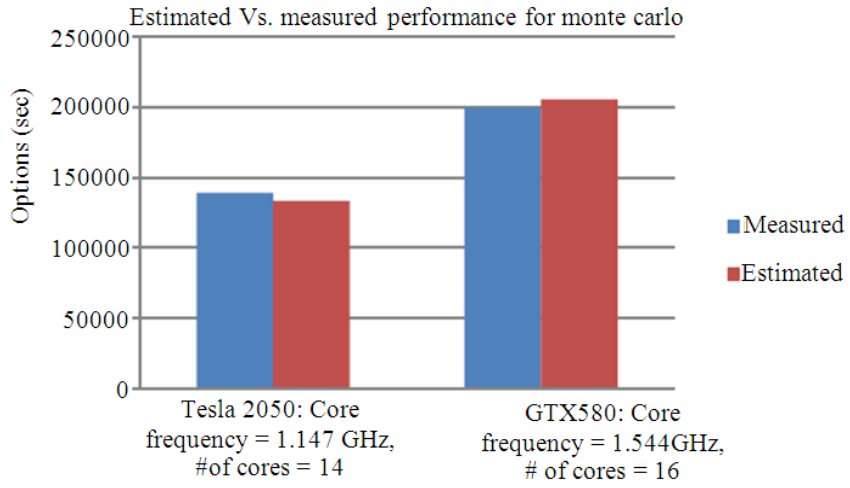

Fig. 5. Estimated Vs Measured for MC

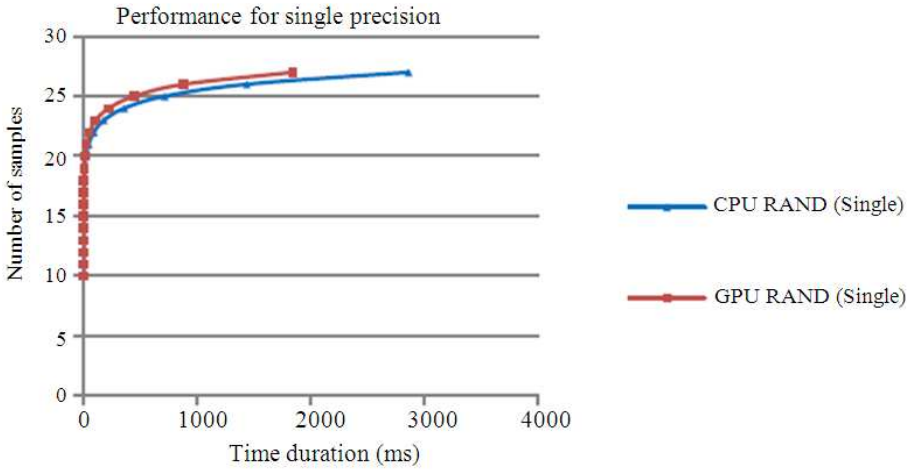

Fig. 6. GPU and CPU RAND timing for single precision for different random sizes

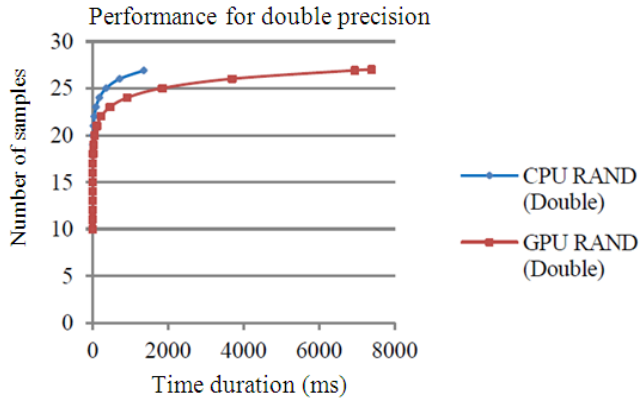

Fig. 7. GPU RAND and CPU RAND timing for double precision for different random sizes

5. CONCLUSION

In this study, we analyzed MC benchmark; we developed a performance estimation model as a function of several processor architecture parameters related to performance. We verified the model by testing different processors configuration running a given benchmark and compared measured results with estimated results. All tested experiments for performance show deviation error between estimated and measured data of <5%. We also presented a hybrid implementation and performance model for MC workload in which we analyzed performance benefits for MC workload when running in hybrid mode instead of GPU only mode. The Hybrid model is implemented from the perspective of data and task decomposition. Pipelining should be used to hide memory traffic between devices. Parallel execution on all devices can be achieved using asynchronous operations, which either is synchronizing using events or simply use an in-order queue and asynchronous operations. Hybrid performance depends on many factors such as problem sizes, algorithms optimizations, capabilities of devices in which it's best when the devices are well balanced; the bigger the processing power gap between the devices the less performing hybrid will be. In conclusion, the hybrid implementation for MC shows the CPU is about 3-6 times slower than the GPU for data parallel problems, but the CPU can perform much better on task parallel problem. For double precision PRNG, the i7 CPU is about 6 times faster than the GTX 480 and about 3 times faster on CPU as compares to Tesla M2090. We conclude that hybrid implementation is a possibility to achieve higher performance under certain circumstances like we have described in this study.

6. REFERENCES

Aoki, R., S. Oikawa, T. Nakamura and S. Miki, 2011. Hybrid openCL: Enhancing openCL for distributed processing. Proceedings of the IEEE 9th International Symposium on Digital Objerct Identifier, May, 26-28, IEEE Xplore Press, Busan, pp: 149-154. DOI: 10.1109/ISPA.2011.28

Bakthavatsalam, G. and K.M. Mehata, 2014. A case for hybrid instruction encoding for reducing code size in embedded system-on-chips based on RISC processor cores. J. Comput. Sci., 10: 411-422. DOI: 10.3844/jcssp.2014.411.422

Goel, B., S.A. McKee, R. Gioiosa and K. Singh, 2010. Portable, scalable, per-core power estimation for intelligent resource management. Proceedings of the Internatoinal Green Computing Conference, Aug. 15-18, IEEE Xplore Press, Chicago, IL., pp: 135-146. DOI: 10.1109/GREENCOMP.2010.5598313

Pennycook, S.J., S.D. Hammond, S.A. Jarvis and G.R. Mudalige, 2011. Performance analysis of a hybrid MPI/CUDA implementation of the NAS-LU benchmark. Proceedings of the 1st International workshop on Performance Modeling, Benchmarking And Simulation of High Performance Computing Systems, (PCS' 11), ACM New York, pp: 23-29. DOI: 10.1145/1964218.1964223

Yu, Y., A.C. Megri, K.M. Flurchick and K.C.D. Bahadur. 2014. The improvement of the computational performance of the zonal model POMA using parallel techniques. Am. J. Eng. Applied Sci., 7: 185-193. DOI: 10.3844/ajeassp.2014.185.193