Characterizing Performance and Power towards Efficient Synchronization of GPU Kernels

Islam Harb
Department of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
Electronic Research Institute, Egypt
iharb@vt.edu

Wu-Chun Feng
Department of Computer Science
Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, Virginia, USA
wfeng@vt.edu

***Abstract*—There is a lack of support for explicit synchronization in GPUs between the streaming multiprocessors (SMs) adversely impacts the performance of the GPUs to efficiently perform inter-block communication. In this paper, we present several approaches to inter-block synchronization using explicit/implicit CPU-based and dynamic parallelism (DP) mechanisms. Although this topic has been addressed in previous research studies, there has been neither a solid quantification of such overhead, nor guidance on when to use each of the different approaches. Therefore, we quantify the synchronization overhead relative to the number of kernel launches and the input data sizes. The quantification, in turn, provides insight as to *when* to use each of the aforementioned synchronization mechanisms in a target application. Our results show that implicit CPU synchronization has a significant overhead that hurts the application performance when using medium to large data sizes with relatively large number of kernel launches (i.e. ~1100-5000). Hence, it is recommended to use explicit CPU synchronization with these configurations. In addition, among the three different approaches, we conclude that dynamic parallelism (DP) is the most efficient with small data sizes (i.e., $\leq$128k bytes), regardless of the number of kernel launches. Also, Dynamic Parallelism (DP), implicitly, performs inter-block (i.e. global) synchronization with no CPU intervention. Therefore, DP significantly reduces the power consumed by the CPU and PCIe for global synchronization. Our findings show that DP reduces the power consumption by ~8-10%. However, DP-based synchronization is a trade-off, in which it is accompanied by ~2-5% performance loss.**

***Index Terms*—GPU, CPU Synchronization, Dynamic Parallelism**

I. Introduction

To address the lack of direct support for native inter-block synchronization on the GPU, researchers have adopted indirect mechanisms such as GPU barrier synchronization [2], implicit CPU barrier synchronization, explicit CPU barrier synchronization, and more recently, dynamic parallelism (DP). However, these mechanisms incur non-trivial overhead compared to a hypothetical native synchronization primitive implemented in hardware.

Synchronization within a GPU can be classified into intra-block and inter-block synchronization. Intra-block synchronization coordinates the threads *within* a streaming multiprocessor (SM) in the context of shared on-chip memory. On the other hand, inter-block synchronization coordinates data communication between threads that span *across* different streaming multiprocessors (SMs) in the context of global off-chip memory. Off-chip (i.e., global) memory access latency is significantly higher than that of the on-chip (i.e., local) memory. Therefore, inter-block synchronization incurs orders of magnitude higher overhead than that of the intra-block synchronization. In this study, we focus on inter-block communication/global barrier synchronization, which is also referred to as inter-streaming-multiprocessor (i.e., inter-SM) synchronization.

Traditionally, global synchronization is done via terminating the current kernel execution, then re-launching it again or even launching another kernel. By default, the CUDA kernel launches are asynchronous. That means the CPU will offload the computation to the GPU and return immediately, as shown in Figure 1(a). We refer to this mechanism as *implicit* CPU global synchronization. On the other hand, NVIDIA provides a mechanism to support synchronous (i.e. blocking) kernel launches by calling "cudaDeviceSynchronize()" API after the kernel launch. This API blocks at the CPU until the GPU finishes the current kernel computation, as shown in Figure 1(b). We refer to this mechanism as *explicit* CPU global synchronization. For this, the latter incurs larger overhead. However, under specific circumstances, our study shows that the *implicit CPU global synchronization* may experience a significant performance degradation, thus the use of *explicit CPU global synchronization* is required.

Recently, an indirect method for GPU-based synchronization, namely *Dynamic Parallelism (DP)* [10], [11], has been introduced in recent NVIDIA architectures (e.g. GK110), in an attempt to lessen the CPU-GPU communication overhead and enhance the dynamic load balancing as shown in [6], [7], [8], [12]. DP represents implicit On-GPU global barrier synchronization without CPU intervention. However, much of the recent work that has been done with DP indicates that it incurs significant overhead [6], [8], [9], [13], [14], thus it is claimed to be *impractical*. On the other hand, several researches [20], [21] took place to introduce alternatives to the Dynamic Parallelism because of its high overhead. Our analysis uncovers scenarios where DP outperforms the other synchronization mechanisms.

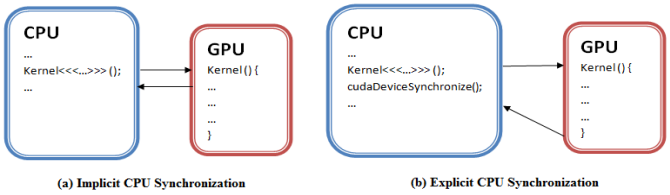

Fig. 1. The CPU Synchronization Mechanisms

The explicit GPU global synchronization [2] might be competitive with DP. However, NVIDIA introduces "__threadfence()" API to, theoretically, ensure correctness of inter-block communication. The overhead of the *explicit GPU synchronization* with the "__threadfence()" is significantly high [5]. In addition, it comes with a limitation on the number of blocks executing on the GPU, in which it should *not* exceed the number of the Streaming Multiprocessors (SMs). As such, explicit GPU global synchronization is opted to be out of scope of this paper.

Previous work in GPU kernel launch and synchronization has not provided guidance on when it is appropriate to use each of the aforementioned approaches to synchronization within a specific target application. Although some work has been done to characterize the overhead of synchronization primitives and protocols, it has been mainly from the hardware (i.e., platform) point of view. In addition, dynamic parallelism (DP) is missing from these previous studies.

Therefore, in this paper, we conduct a comprehensive study and characterization of the overhead for the different approaches to synchronization for the GPU. Our contributions are as follows.

- Guidelines to choose the most appropriate synchronization mechanism based on application's parameters.
- Quantification of the kernel launch time and the synchronization overhead for each of the mechanisms vs. the number of kernel launches and the data sizes.
- We are the first to use the Dynamic Parallelism as a mean of global synchronization. In addition, We show its power consumption advantage compared to the other mechanisms (i.e.,CPU-based global synchronization).
- Realization of synthetic micro-kernel and application benchmarks to stress-test the different approaches to synchronization.

The rest of the paper is organized as follows. Section II presents the work related to synchronization and dynamic parallelism (DP). Section III discusses the applications and its role in studying the overhead of the synchronization mechanisms. Section IV, then, analyzes and quantifies this overhead of each synchronization mechanism. In addition, we conduct a comparison of performance vs. power consumption for both the DP-based and the other synchronization mechanisms. Finally, section V concludes our work and discusses future work.

II. Related Work

Our work is related to the area of synchronization protocols for many-core architectures and characterization of the Dynamic Parallelism.

The explicit GPU-based synchronization can be realized by either lock-based or lock-free techniques as introduced in [2]. Both techniques require the number of blocks to be less than or equal to the number of the streaming multiprocessors in the GPU to avoid the potential deadlock. Their study shows that the explicit GPU-based synchronization may incur a significant overhead, relative to the implicit CPU synchronization, when using the memory fence API. Mehmet et al. [3] have used the wavefront parallelism to mitigate the explicit GPU-based synchronization overhead. Meanwhile, Gupta et al. [15] have introduced the persistent thread concept to overcome the limitations on the number of blocks in the explicit GPU-based synchronization mechanism. However, none of these work considered the Dynamic Parallelism.

The work of David et al. [4] focus on the synchronization over multiple layers with the emphasis on the cache-coherency and locks. Stuart et al. [18] conducted a research on the efficient synchronization primitives (e.g. atomic accesses) over many-core architectures. However, the their work is on characterizing the synchronization overhead based on many-core architecture and hardware targets. Both don't consider the applications configurations and parameters. In addition, the former study is meant for the CPU but it cannot be directly mapped to the GPU environment.

On the other hand, several efforts has been introduced to reduce or eliminate the global barrier synchronizations [16], [17], [19]. These are optimization studies to lessen the synchronization points within an application. They didn't provide any characterization of the overhead of the synchronization mechanisms.

Jin et al. [9] lead a study for characterizing the dynamically-formed parallelism on irregular (i.e. unstructured) applications on GPUs. They conclude that the Dynamic Parallelism causes ~1.21x slowdown due to its non-trivial overhead. Dimarco et al. [8] carried out a study on the use of the Dynamic Parallelism to accelerate clustering algorithms, which also confirms its significant overhead. However, both works are addressing DP for dynamic load balancing in irregularity in applications. It didn't discuss synchronization overhead. In addition, they didn't cover structured or regular applications. Our analysis provides DP overhead quantification and guidelines on when to use each of the global barrier synchronization, including the Dynamic Parallelism, for each target application.

III. Approach and Applications

We implement a synthetic micro-benchmark to analyze and understand the behavior of the CPU and GPU (i.e. Dynamic Parallelism) synchronization mechanisms over a variety spectrum of workloads. The micro-benchmark represents computations with different memory access characteristics. The kernels (i.e. computations) that require low to no read/write global memory access are classified as light-weight kernels.

On the other hand the kernels that require average to intense read/write global memory accesses are classified as average-to-heavy-weight kernels. Our micro-benchmark includes kernels with memory access patterns as follows.

- Empty Kernel.
- Shared-Memory Kernel: Computations read/write from/to shared memory only.
- Global Memory Kernel: Computations read/write from/to global memory. Allocation is done by either the CPU or the GPU.
- Local Memory Kernel: Computations read/write from/to private memory or registers only.
- Others: combination of the above primitives.

Apart from the micro-bench mark, we have selected two applications for more insights and evaluation. They are a good approximation of real-world applications.

- **The Lid-Driven Cavity (LDC)**: A computational fluid dynamic application that has stress, viscosity and pressure calculations on a mesh of a *default* size 3x4096x4096. Each mesh cell is a double-precision floating point that occupies 8 bytes [22].
- **The Heat2D**: NVIDIA open source heat transfer simulation in a two-dimensional space [1].

IV. Experiments Discussion and Evaluation

For each of the experiments, we did 20 runs and then took the average to make our results resilient to the external uncontrolled errors.

In order to quantify the synchronization overhead, we examine each of the synchronization mechanisms versus the number of kernel launches and the input data sizes (i.e. mesh sizes). The data type, in all applications, is double-precision floating point (i.e., 8 bytes). That means, mesh size can be translated into *"bytes"* unit via multiplying the dimensions by "8". For instance, mesh size of 128x128 is equivalent to 128x128x8 = 128 KBytes (KB). In addition, the mesh size affects directly on the number of blocks running on the GPU. Thus, it relates to the number of synchronization points and its overhead. The block size is fixed to 16x8 threads. Therefore, alternatively, the mesh size can be translated to the number of blocks which can be calculated as shown in 1. For instance, data mesh size of 128x128 is equivalent to 128 Blocks.

$$Blocks = \left\lceil \frac{Mesh\_Dim\_X}{16} \right\rceil * \left\lceil \frac{Mesh\_Dim\_Y}{8} \right\rceil \quad (1)$$

We implement the applications with the three different synchronization mechanisms: the implicit CPU, the explicit CPU and the Dynamic Parallelism. We evaluate their power consumption, performance and overhead on both Kepler K20c and Tesla K20Xm GPUs with CUDA 6.0. The computational kernel is kept the same across all the variants. The explicit CPU synchronization mechanism requires the addition of "cudaDeviceSynchronize()" at the host side only. As for the Dynamic Parallelism, we implemented an auxiliary kernel that is launched once from the CPU side, and then it will manage all the launches and synchronization of the computational kernel within the GPU.

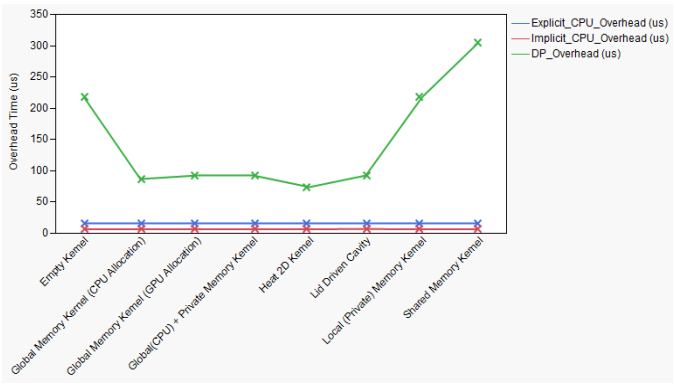

Fig. 2. The Synchronization Overhead Across Multiple Workloads

We used *NVIDIA Profiler* to collect numbers and analysis reports. It reports a breakdown that shows the kernel launch time (i.e. Overhead) and the execution time (i.e. Computation time) separately for the CPU synchronization mechanisms. However, with Dynamic Parallelism, it reports an integrated number for both launch and execution times . Since the computational kernel is untouched, the execution time should remain the same across all the synchronization mechanisms. Thus, we subtract the execution time, of the CPU synchronization run, from the integrated number reported in the DP run, to obtain the overall synchronization overhead (i.e. launch and sync).

The implicit CPU synchronization mechanism is recognized for its best performance and its least overhead among the aforementioned synchronization mechanisms. Therefore, we use it to characterize and classify our benchmark as shown in Figure 2. It shows the overhead of the three mechanisms with 1000 kernel launches and 4096x4096 mesh size each. The number of kernel launches (i.e., 1000) is recommended by the domain scientists for the LDC. The implicit CPU synchronization outperforms both the explicit CPU synchronization and the Dynamic Parallelism, which is already expected. It is worth to mention that the Dynamic Parallelism has significantly larger overhead with light-weight kernels (e.g. empty, local or shared memory computations) than that of the medium-to-heavy-weight kernels (e.g. global memory computations, LDC and Heat2D).

In the next subsections, we pick representatives of each of the light-weight and medium-to-heavy-weight kernels for further analysis. The "Empty" and the "Shared-Memory" Kernels represent the former category. Meanwhile, the "Lid-Driven Cavity" and the "Heat2D" Kernels represent the latter category.

A. Light-Weight Kernels

We examined the synchronization overhead versus the number of kernel launches (i.e. 1-10k) for the light-weight kernels. Figure 3 and 4 show the synchronization overhead for the "Empty" Kernel and the "Shared-Memory" kernel respectively. This experiment answers the research question on which

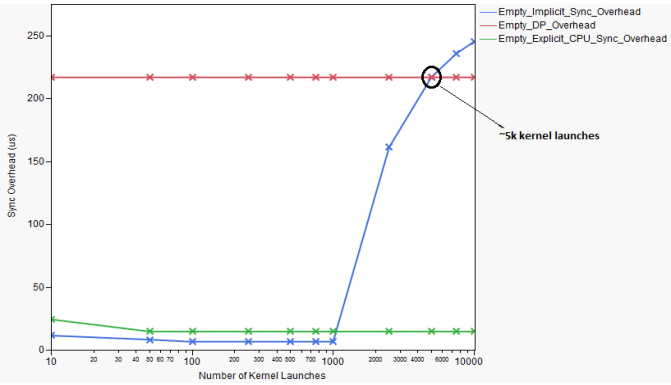

Fig. 3. The Synchronization Overhead vs. No. of Kernel Launches – Empty Kernel

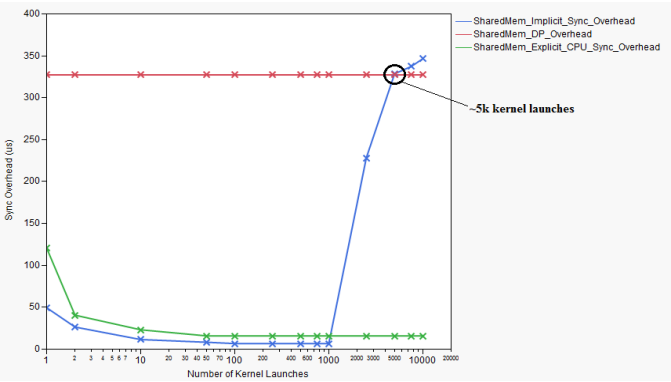

Fig. 4. The Synchronization Overhead vs. No. of Kernel Launches – Shared-Memory Kernel

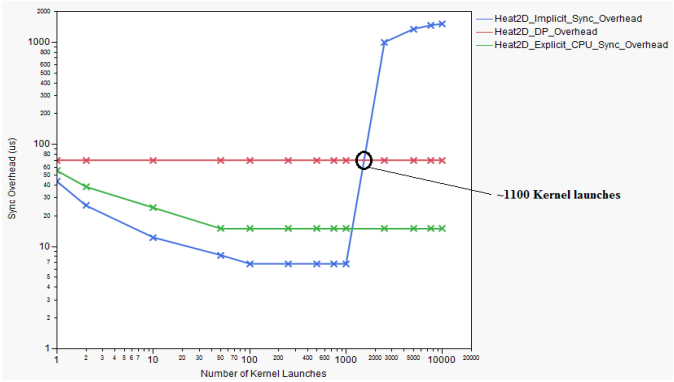

Fig. 5. The Synchronization Overhead vs. No. of Kernel Launches – Heat2D Kernel

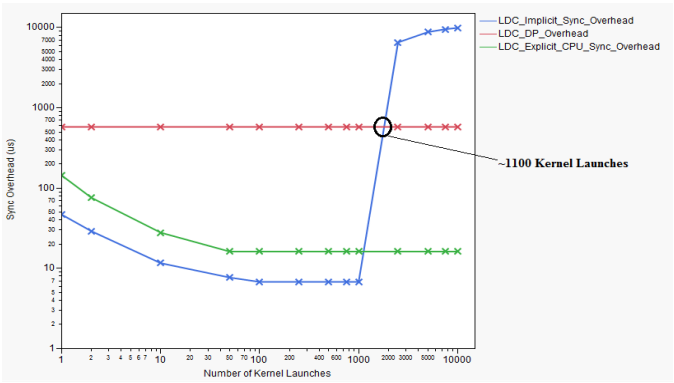

Fig. 6. The Synchronization Overhead vs. No. of Kernel Launches – LDC Kernel

synchronization mechanism is appropriate for the light-weight kernels, given the number of kernel launches. It shows the cut-off (i.e. ~5k launches), at which the Dynamic Parallelism and the Explicit CPU synchronization outperform the Implicit CPU synchronization. This is due to the fact that the implicit CPU synchronization is a non-blocking mechanism. That means, it allows multiple kernel launches from the CPU side, even if it can't be all served and hence queued. So at a certain limit, it has to do a time consuming flush for the accumulated tasks in the queues. This limit is at 5k kernel launches with the light-weight kernels.

B. Medium-to-Heavy-Weight Kernels

Similarly, we examined the synchronization overhead versus the number of kernel launches (i.e. 1-10k) for the medium-to-heavy-weight kernels. Figure 5 and 6 show the synchronization overhead for the "Heat2D" and the "LDC" Kernels. In this case, we aim to answer the research question on which synchronization mechanism is appropriate for the medium-to-heavy-weight kernels, given the number of kernel launches. It shows the cut-off (i.e. 1100 launches), at which the Dynamic Parallelism and the Explicit CPU synchronization outperform the Implicit CPU synchronization. This is due to the same reason of queues flushing.

C. Synchronization Overhead vs. Data Size

The data size is an effective factor in determining the number of blocks that are executing on the GPU. We believe that the synchronization overhead increases with the increase of the data size. Therefore, we need to answer the research question about the data size cut-off at which the *implicit CPU synchronization mechanism* remains robust (i.e. No performance degradation) given the previous cut-offs of the number of kernel launches. Figure 7 shows that the data size should be $\leq$ 128x128 (128 KB), in order to achieve high performance with large number of kernel launches (i.e. $\geq$ 1000).

On the other hand, we evaluate the overhead of the implicit CPU vs. the explicit CPU vs. the Dynamic Parallelism synchronization mechanism across the different data sizes. Figure 8 shows that the overhead of dynamic parallelism is the least among the three synchronization mechanisms, when the data size is small (i.e. <= 128x128 or 128 KB) regardless of the number of kernel launches. Therefore, the Dynamic Parallelism is the most appropriate synchronization mechanism for applications of relatively small input data sizes. This conclusion is confirmed and clarified in figure 9, that shows the synchronization overhead with the Lid-Driven Cavity (LDC) of 32 KB data size .

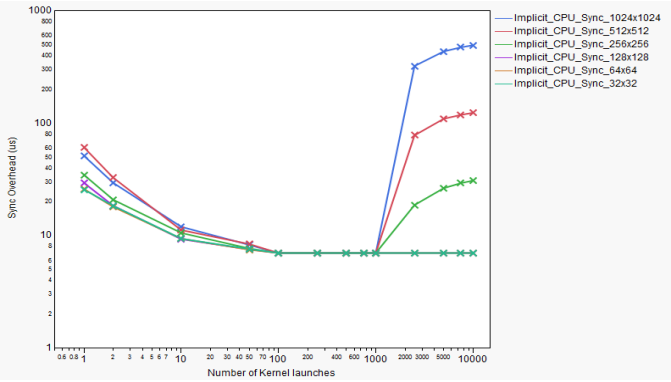

Fig. 7. The Implicit CPU Synchronization Overhead vs. Data Size – LDC Kernel

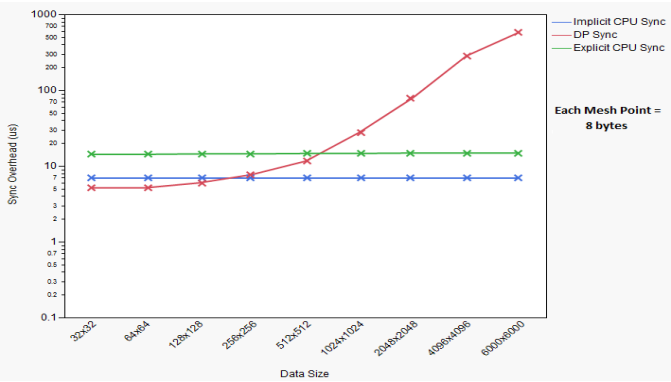

Fig. 8. The Synchronization Overhead vs. Data Size – LDC Kernel

D. Power vs. Performance Trade-Off

Unlike CPU-based global synchronization, DP globally synchronize the kernel running on the GPU without CPU intervention. Hence, Dynamic Parallelism reduces the power significantly by cutting the portions consumed by the CPU and the PCIe. We are concerned since power/energy saving is critical towards the exascale computing realization.

We used the *NVIDIA Management Library* (NVML) to collect power readings using "nvmlDeviceGetPowerUsage()"

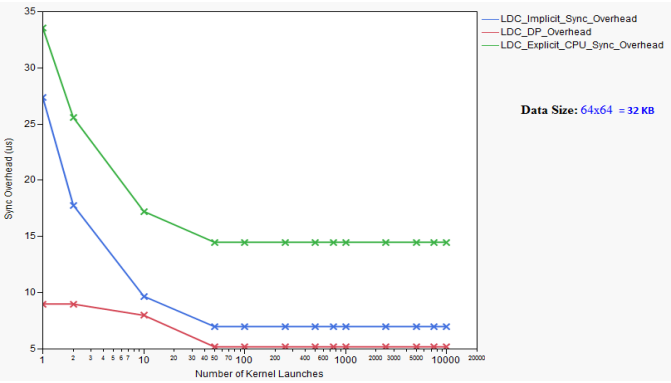

Fig. 9. The Synchronization Overhead vs. 64x64 Data Size – LDC Kernel

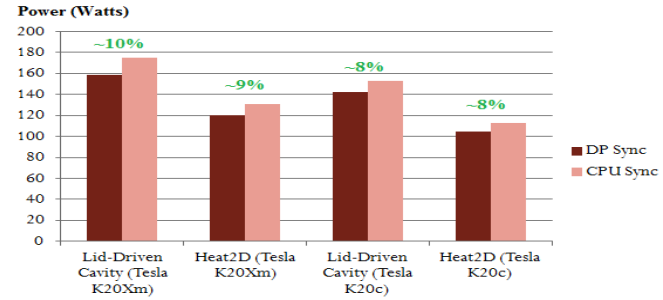

Fig. 10. DP vs. CPU: Power Consumption of the LDC and Heat2D Over Tesla K20c and Tesla K20Xm

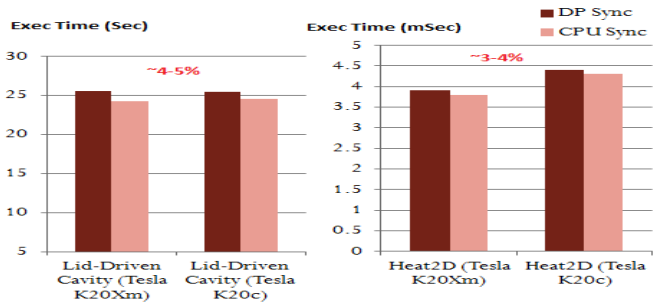

Fig. 11. DP vs. CPU: Execution Time of the LDC and Heat2D Over Tesla K20c and Tesla K20Xm

API. The power reading is sampled during the execution lifetime and then averaged out. The sample count is ~25-30.

In this section, implicit CPU-based and the DP-based global synchronizations have been compared with respect to the power and performance (i.e. execution time). we exclude the synthetic micro-benchmarks as numbers/readings will not be realistic with respect to the power consumption. Therefore, we report only the LDC and the Heat2D benchmarks. Our experiments are performed over both Tesla K20c and K20Xm platforms. Figure 10 shows the power consumption of the LDC and the Heat2D for both the *implicit CPU-based* and the *Dynamic Parallelism GPU-based* global synchronization mechanisms. DP global synchronization reduced the power consumption significantly by ~8-10% and ~8-9% for both the LDC and the Heat2D respectively. However, this power consumption improvement has a trade-off, in which there is a slight performance degradation accompanied with the DP mechanism compared to the CPU-based mechanism. Figure 11 shows that the DP global synchronization slightly increases the execution time by ~4-5% and ~2-3% for both the LDC and the Heat2D respectively.

V. Conclusion and Future Work

There is no explicit support for the Inter-Block synchronization. Several global synchronization mechanisms and protocols have been introduced during the past couple of years. However, there is still neither solid quantification nor comprehensive characterization for the overhead of the different synchronization mechanisms. Hence, we carried out a study

to characterize the overhead of the known synchronization mechanisms. Our goal is to answer research questions about when and for which application, a specific synchronization mechanism should be used. We covered the implicit CPU synchronization, the explicit CPU synchronization and the Dynamic Parallelism in this study. Meanwhile, we are looking into extending our work by including the explicit GPU-based synchronization [2] with the *Persistent Thread* [15] approach .

Our results have challenged the idea that the *implicit CPU synchronization* mechanism always has the best performance and the least overhead. With the large number of kernel launches, the implicit CPU synchronization shows a significant overhead and performance degradation. This is due to the fact of the non-blocking kernel launches that are queued along with memory writes that keep filling up the buffer and need to be flushed at a certain limit. We defined this limit, in our study, by both the number of kernel launches and the data size. Our results determined the cut-offs of the number of kernels launches and the data size, at which the performance degradation occurs. The number of kernel launches cut-off is ~5000 and ~1100 for the light-weight and medium-to-heavy-weight kernels respectively. But, this cut-offs are valid when the data size is relatively large (i.e. $\geq$ 128 KB). In addition, our results show that the DP is the most appropriate way of global synchronization among the three mechanisms with small data sizes (i.e. $\leq$ 128 KB). The DP performance advantage is observed regardless of the number of kernel launches. On the other hand, the DP significantly reduced the power consumption by ~8-10% compared to the implicit CPU-based global synchronization mechanism. However, it yield to a slight loss in the performance by ~2-5%.

In the future, we would like to extend our benchmark to be more representative by including more computation patterns such as Berkley's 13 dwarfs. We also need to build our own generic model that predicts the appropriate synchronization mechanism for the target application over the many-core architectures (i.e. GPU). One thought is to use an approximation to the BSP model [23] in which communication cost will be simplified into just global synchronization (i.e. Data Transfer will be omitted). Processors, in the BSP context, will be equivalent to the Streaming Multiprocessors in the GPU. As mentioned earlier, since the computation kernel is the same across all mechanisms, therefore, the computation parameters and rate can be either neglected or set to a constant value. Finally, automatic translation from/to any of the different global synchronization mechanisms will be of a great benefit to improve the programmability.

REFERENCES

[1] NVIDIA, Sanders, J. and Kandrot, E.: CUDA by Example – An Introduction to General-Purpose GPU Programming., Available:http://developer.download.nvidia.com/books/cuda-by-example/cuda-by-example-sample.pdf

[2] Xiao, S., Feng, W. : Inter-block GPU Communication via Fast Barrier Synchronization, In: IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10), pp. 1–12, 2010

[3] Belviranli, M., Deng, P., Bhuyan, L., Gupta, R., Zhu, Q.: PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization, In: Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15), pp. 25–35, 2015

[4] David, T., Guerraoui, R., Trigonakis, V.: Everything You Always Wanted to Know about Synchronization But were Afraid to Ask, In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13), pp. 33–48, 2013

[5] Feng, W., Xiao, S. : To GPU synchronize or not GPU synchronize?, In: Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10), pp. 3801–3804 , 2010

[6] Dong, J.,Wang, F., Yuan, B.: Accelerating BIRCH for Clustering large scale streaming data using CUDA dynamic parallelism, In: Intelligent Data Engineering and Automated Learning (IDEAL13), pp. 409–416, Springer, 2013

[7] Chang, D., Kantardzic, M., Ouyang, M.: Hierarchical Clustering with CUDA/GPU, In: Proceedings of the 22nd International Conference on Parallel and Distributed Computing and Communication Systems (ISCA PDCCS09), 2009

[8] DiMarco, J., Taufer, M.: Performance Impact of Dynamic Parallelism on Different Clustering Algorithms, In: proceedings SPIE Defense, Security, and Sensing, International Society for Optics and Photonics, Baltimore, Maryland, USA, 2013

[9] Wang, J., Yalamanchili, S.: Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications. In: IEEE International Symposium on Workload Characterization (IISWC'14), 2014

[10] NVIDIA, CUDA C Programming Guide, September, 2015, Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[11] NVIDIA, Cuda Dynamic Parallelism Programming Guide, August, 2012, Available:http://dirac.ruc.dk/manuals/cuda-5.0/CUDA_Dynamic_Parallelism_Programming_Guide.pdf

[12] NVIDIA, Jones, S.: How Tesla K20 Speeds QuickSort, a familiar comp-sci code, Available:http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code

[13] Yang, Y., Zhou, H.: CUDA-NP: Realizing Nested Thread-level parallelism in GPGPU applications, Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP14), pp. 93–106, 2014

[14] Oden, L., Klenk, B.,Froning. H.: Energy-efficient Computing on Distributed GPUs using Dynamic Parallelism and GPU controlled Communication, In: 2nd Workshop on Energy-efficient SuperComputing (E2SC), Germany, 2014

[15] Gupta, K., Stuart, J.A., Owens, J.D.: A study of Persistent Threads style GPU programming for GPGPU workloads, In: IEEE Innovative Parallel Computing (InPar'12), pp. 1–14, 2012

[16] Yan, S. Long, G., Zhang, Y.: StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization, In: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'13), pp. 229–238, 2013

[17] Bondhugula, U., Bandishti, V., Cohen, A., Potron, G., Vasilache, N.: Tiling and Optimizing Time-iterated Computations on Periodic Domains. In: Proceedings of the 23rd international ACM conference on Parallel architectures and compilation (PACT'14), pp. 39–50, 2014

[18] Stuart, J., Owens, J.: Efficient Synchronization Primitives for GPUs, CoRR, abs/1110.4623(1110.4623v1), October 2011

[19] Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat Combining and the Synchronization-Parallelism Tradeoff, In: ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10), pp. 355-364, 2010

[20] Chen, G., Shen, X.: Free Launch: Optimizing GPU Dynamic Kernel Launches Through Thread Reuse, In: Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15), pp. 407-419, 2015

[21] Wang, J., Rubin, N., Sidelnik, A., Sudhakar, Y.: Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPU, In: Proceedings of the 42nd International Symposium on Computer Architecture (ISCA'15), pp. 528-540, 2015

[22] Pickering, B. P., Jackson, C. W., Scogland, T. R., Feng, W.-C., and Roy, C. J.: Directive-based gpu programming for computational fluid dynamics, In AIAA SciTech, 52nd Aerospace Sciences Meeting, January 2014.

[23] Valiant, L. G. : A bridging model for multi-core computing, Journal of Computer and System Sciences, Volume 77 Issue 1, Pages 154-166, January, 2011