Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 15, Number 1, pages 36–37, January 2026

Developing a generator for CUDA C files to reduce synchronization overhead (preliminary version)

Daisuke Takafuji
Faculty of Information Science, Shunan University
843-4-2 Gakuendai, Shunan City,
Yamaguchi, 745-8566, Japan
E-mail: takafuji@shunan-u.ac.jp

***Abstract*—In most GPU implementations of parallel algorithms, multiple separated CUDA kernels are called from the host one by one for barrier synchronization. However, CUDA kernel calls have large overheads and barrier synchronization degrades computing resource usage. The Single Kernel Soft Synchronization (SKSS) technique performs only one CUDA kernel and assigns tasks to CUDA blocks dynamically. The SKSS technique can fully exploit GPU computing resources, and many studies have reported performance improvements achieved using SKSS. Unfortunately, efficient GPU implementations using the SKSS technique require extensive knowledge and experience in CUDA programming. To address this issue, we present in this paper a tool that converts CUDA C files without SKSS into SKSS-enabled CUDA C files.**

***Index Terms*—GPU, CUDA, Synchronization**

I. Introduction

A Graphics Processing Unit (GPU) is a specialized circuit designed to accelerate computation or building and manipulating images [1]. The most recent GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many applications developers [1]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [2], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors, since they have hundreds of processor cores and very high memory bandwidth. We can use the CUDA C++ Best Practices Guide [3] to implement high-performance CUDA applications. From the guide, we can learn optimization strategies across memory usage, parallel execution, and instruction-level efficiency. In [4], Funasaka et.al. have proposed *Single Kernel Soft Synchronization (SKSS) Technique* to fully utilize GPU computing resources. We have used the SKSS technique [5]–[7], and our GPU-based implementations demonstrated substantial speedup through the application of the SKSS technique. In most GPU implementations of parallel algorithms, multiple separated CUDA kernels are called from the host one by one for barrier synchronization. However, CUDA kernel calls have large overheads and barrier synchronization degrades computing resource usage. The SKSS technique performs only one CUDA kernel and assigns tasks to CUDA blocks dynamically. In many applications, GPU implementations achieve a high acceleration rate through the SKSS technique.

Unfortunately, efficient GPU implementations using the SKSS technique require extensive knowledge and experience in CUDA programming. To address this issue, we present in this paper a tool that converts CUDA C files without SKSS into SKSS-enabled CUDA C files.

II. SKSS Technique

In this section, we introduce the Single Kernel Soft Synchronization (SKSS) technique on the GPU, used in the kernels.

First, we explain CUDA programming model of GPUs necessary to understand GPU implementations. Please see [2] for the details. NVIDIA GPUs have streaming multiprocessors with multiple cores. Usually, a CUDA program performs kernel calls with multiple *CUDA blocks* in turn. Every CUDA block have the same number of threads, which execute the same program. Each CUDA block is loaded in one of the streaming multiprocessors for execution. Threads in a CUDA block are partitioned into a set of 32 threads called *a warp*. A warp dispatched to cores in a streaming processor by a scheduler, and all 32 threads in it work synchronously.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [2]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs.

Using SKSS technique, the kernels can perform no barrier synchronization and invokes only one kernel call. The technique is used in [4]–[8], and is described as follows.

When we use the SKSS technique, we must define the task assigned CUDA blocks and define a constant number Z, which represents the task ID when CUDA blocks terminate. In the kernel using the SKSS technique, CUDA blocks are assigned to tasks in the order of task IDs. For this purpose, we use a zero-initialized global counter `x` in the global memory. We invoke enough CUDA blocks and the first thread of every CUDA block increments `x` using CUDA atomic function `atomicAdd(&x,1)`, which exclusively increments the value of `x` by 1 and returns the value of `x` before increment. Clearly,

the first threads of CUDA blocks succeeding in executing atomicAdd receive unique return values 0, 1, 2, ... in turn. If the first thread receives the value greater than Z then the CUDA block terminates. Otherwise, the CUDA block performs a task with IDs equal to the return value.

III. Outline of Our Generator

As described in Section II, the SKSS technique can fully exploit GPU computing resources, and many studies have reported performance improvements achieved using SKSS. Unfortunately, efficient GPU implementations using the SKSS technique require extensive knowledge and experience in CUDA programming. We present a tool that converts CUDA C files without SKSS into SKSS-enabled CUDA C files.

Our tool provides only basic functionality, which translates functions in CUDA C files without SKSS into SKSS-enabled counterparts. It is written by Python 3.12.3 [9]. When a non-SKSS CUDA C file is given as input, the tool generates an SKSS-enabled file. By enclosing a non-SKSS CUDA C file between the following two lines, it can be converted into an SKSS-enabled version.

```
#pragma __SKSS__ BEGIN(NUM)

#pragma __SKSS__ END
```

To define the task assigned CUDA blocks, a counter variable is declared, and for each CUDA block launch, `blockIdx.x` is replaced with this variable. That is, the counter variable represents a task ID. The counter variable is stored in global memory, allowing it to be accessed (read and written) by all blocks. It is also incremented using atomicAdd.

The argument `NUM` specified after `BEGIN` represents a constant number, which represents the task ID when CUDA blocks terminate.

Because this tool merely inserts `#pragma` directives, it is applicable to both CUDA C and CUDA C++.

IV. Conclusion

By focusing on the SKSS technique for GPU architectures, we have presented a tool that converts CUDA C files without SKSS into SKSS-enabled CUDA C files. This tool helps generate SKSS-enabled CUDA C files. Because the SKSS technique can fully exploit GPU computing resources, we conclude that this tool is useful.

Since no suitable example was found, this paper does not present any usage examples of this tool. In the future, we plan to find examples that can demonstrate the effectiveness of this tool and verify its implementation on GPUs. Furthermore, our tool in this paper provides only basic functionality, and it has strict constraints for its use. We aim to improve our tool by extending its functionality.

References

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] NVIDIA Corporation, "NVIDIA CUDA C++ programming guide version 13.0," November 2025.

[3] ——, "NVIDIA CUDA C++ best practice guide version 13.0," https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, October 2025.

[4] S. Funasaka, K. Nakano, and Y. Ito, "Single kernel soft synchronization technique for task arrays on CUDA-enabled GPUs, with applications," in *Proc. of 2017 Fifth International Symposium on Computing and Networking (CANDAR)*. Proc. of 2017 Fifth International Symposium on Computing and Networking (CANDAR), Nov. 2017, pp. 11–20.

[5] N. Yamamoto, K. Nakano, Y. Ito, D. Takafuji, A. Kasagi, and T. Tabaru, "Huffman coding with gap arrays for GPU acceleration," in *Proc. of the 49th International Conference on Parallel Processing (ICPP2020)*, Aug. 2020, pp. 1–11.

[6] D. Takafuji, K. Nakano, and Y. Ito, "Efficient parallel implementations to compute the diameter of a graph," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 11, 2023.

[7] D. Takafuji, K. Nakano, Y. Ito, and A. Kasagi, "GPU implementations of deflate encoding and decoding," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 19, 2023.

[8] D. Merrill. (2017) CUB: A library of warp-wide, block-wide, and device-wide GPU parallel primitives. [Online]. Available: https://nvlabs.github.io/cub/

[9] Python. [Online]. Available: https://www.python.org/