Introduction to Operating Systems

UE17CS302

Project Report

Table of Contents

Verification of Producer-Consumer Synchronization in GPU Programs

Authors

Rahul Sharma

Stanford University

sharmar@cs.stanford.edu

Michael Bauer

NVIDIA Research

mbauer@nvidia.com

Alex Aiken

Stanford University

aiken@cs.stanford.ed

Team Members 5G

Dev Bhartra - PES1201700186 - 16

Ruben John Mampilli - PES1201700005 - 10

Dhruv Vohra - PES1201700281 - 26

Aprameya Kulkarni - PES1201701583 - 50

Manvith J - PES1201701774 - 56

SUMMARY OF THE PAPER

This work presents a formal operational semantics for named barriers and defines what it means for a warp- specialized kernel to be *correct*. It considers the higher performance, but more complex, warp-specialized kernels in GPU's based on producer-consumer named barriers.Algorithms(Sound and Complete) are proposed for verifying the correctness of warp-specialized kernels. WEFT, a verification tool for checking warp-specialized code is also presented.

- ➔ **Why the necessity of GPUs**: GPUs are an established general purpose computing platform for applications that require significant computational resources and memory bandwidth.The interaction of a complex GPU memory hierarchy, including different on-chip software and hardware managed caches, coupled with the large number of threads to consider, makes it easy to introduce data races and other correctness bugs into GPU kernels.

- ➔ **Progress till date:** Previous attempts at writing tools capable of verifying the correctness of GPU kernels have all assumed the standard data-parallel GPU programming model supported by CUDA and OpenCL in which a barrier is used to synchronize all the threads in a Thread-block.It is impossible to create deadlocks using the traditional *syncthreads* primitive(Thread-block) because all invocations map to a single named barrier.

- ➔ **What even are Warp-Specialized kernels**?
  - ◆ Threadblock-wide barrier is limiting for kernels in which threads within the same threadblock perform different computations.Warp-specialized kernels on the other hand assign different computations to warps (groups of 32 threads) within the same thread-block in order to achieve important performance goals (maximizing memory bandwidth ,fitting extremely large working sets on-chip). To perform synchronization between different warps, warp-specialized kernels use the producer-consumer named barriers available in PTX on NVIDIA GPUs. Note :The paper however does not talk about AMD GPUs.
  - ◆ In NVIDIA GPUs, named barriers allow warp-specialized kernels to encode producer-consumer relationships between arbitrary subsets of warps in a threadblock. The hardware within an SM(Streaming Multiprocessor) makes scheduling decisions at the granularity of warps. When a warp is scheduled, all threads within the warp execute the same instruction.
  - ◆ There is no performance penalty if threads in different warps diverge but threads belonging to the same warp executes the same instruction stream. Communication in warp-specialized kernels is asymmetric*(n* Producers, *m* Consumers).Each SM on current NVIDIA GPUs contains 16 physical named

barriers.Once a named barrier completes, it is immediately re-initialized so that it can be used again.Different generations of a named barrier can have different numbers of participants.

➔ **Why to verify:** To motivate the need for verification of warp-specialized kernels, an example from the Heptane chemistry kernel emitted by the Singe DSL compiler is presented.Where dataflow crosses warp boundaries, data must be communicated through shared memory. All crosswarp data flow edges are assigned a named barrier to use when synchronizing access to shared memory.A *happens-before relationship* must be established between the completion of the previous generation of a named barrier and all participants of the next generation to ensure synchronization.

➔ **How to verify:**There are three important properties to check for warp-specialized kernels:

- *Deadlock Freedom*: Checking that the use of named barriers does not result in deadlocks
- *Safe Barrier Recycling*: Named barriers are a limited physical resource and it is important to check that IDs of named barriers are properly re-used
- *Race Freedom:* checking that shared memory accesses synchronized by named barriers are race free.

➔ **Proposed model terms :**

- ***PTX*** provides two instructions for arriving at a named barrier: sync and *arrive*. A *sync* instruction causes the threads in a warp arriving at the barrier to block until the barrier completes. Alternatively, a *arrive* instruction allows a warp to register arrival at a barrier and immediately continue executing without blocking. Both *sync* and *arrive* instructions require the total number of threads participating in the barrier to be specified
- Threads are disabled when they block on a **barrier**. The thread count is configured by the first thread that reaches a barrier.The hardware does not care about the state of the barriers when the CTA exits as the barriers are reset before starting a new CTA.On executing a non-blocking arrive at the barrier, the control and the barrier map are updated.On reaching a *sync*, the thread is disabled and gets added to the list of blocked threads while the control remains unchanged.If too many threads reach a barrier, then transition to the error state.The error productions ensure that an execution either reaches done, goes to error state, or deadlocks
- This approach to verifying the correctness of a **CTA** involves guessing *happens-before* relationships from a concrete execution, performing a static analysis to ensure that the guessed relationships hold for all possible

executions, and using the relationships to prove the important correctness properties

- For state *s* and a CTA *T*, A partial trace or a subtrace is a sequence of configurations (s0, T0), . ,(sn, Tn) such that for any two successive configurations we have (sj , Tj)-> (sj+1, Tj+1). A complete trace ends in either done, err, or a deadlock.Concept of time is used to define a happens-before relationship
- For read, write, and arrive, the time *t* provides the execution step when these are executed .For sync it provides the step when the corresponding barrier is recycled
- Gen(τ) maps a synchronization command to a generation ID observed in the trace τ. The generation ID of unexecuted commands is set to 0.Gen is used to implement happens-before relationships.CTAs with the same generation mapping for all traces are called well-synchronized
- A configuration (s, T) is well-synchronized if for any two traces τ 1 and τ 2 that start from (s, T), for all synchronization commands c, we have Gen(τ 1)(c) = Gen(τ 2)(c) != 0.The non-zero check ensures that no trace of a well synchronized configuration can deadlock or go to an error. Therefore a check for well synchronization subsumes both safe barrier recycling and deadlock freedom. It also ensures that the synchronization behavior is deterministic: in all traces the same commands synchronize together
- If two commands accessing the same shared variable '*g*' do not have a happens-before relationship between them, then they must both be reads, otherwise there is a data race.Only the threads with the same lane id(same warp) can have data races.Statically check that all traces assign the same generations to commands as Gen(τ) to ensure well synchronization.

➔ **Proposed model flow :**

- First Gen(τ) is used to construct a static happens-before relation R. The relation starts empty. Then add successive commands to R. Next, add the edges corresponding to the inter-thread happens-before relations. If c1 is a arrive and c2 is a sync such that the two commands are in the same generation, then add (c1, c2) to R. Now, for c1 and c2 corresponding to sync in the same generation, add (c1, c2) and (c2, c1) to R.The transitive closure of R thus yields the full static happens-before relation.After constructing the happens-before relation for all program commands, check that there exists happens-before relationships between successive generations of the same barrier
- The *soundness* of R is direct: all tuples added to R are sound because there is no out-of-order execution of commands within individual thread programs and because the well synchronization property imposes barrier generations in all traces. For state *s* and CTA *T*, In the induction step, we have a well-synchronized configuration (s1, T1) .Suppose in the given trace,

the configuration shifts to (s2, T2) from (s1, T1),then checks ensure that the new generation introduced in going from T1 to T2 has a happens-before relationship with generations in T1 and (s2, T2) is well-synchronized.

- *Completeness* follows because well-synchronized programs have a precise R. The time complexity is the cost of computing the transitive closure, which for 'n' commands can be O(n^3) in the worst case
- For two commands c1 and c2 that are accessing the same shared memory location with at least one write,a race is reported if (c1, c2) ∈/ R and (c2, c1) ∈/ R. Also we need to check that there are happens-before relationships between the write in P1 and the write in P2, between the write in P1 and the read in P2 and vice versa. A path between any two accesses to the shared memory ensures that 2 programs are race-free.

WEFT, a verification tool:

WEFT is a verification tool developed by the authors that works towards the alignment of threads in the GPU environment. The input will be the PTX assembly code and the translated output will be the formal language as described above. The translation takes place in a series of steps:

WEFT emulates the 1st encountered CTA (cooperative thread array) till it either terminates, gets deadlocked or encounters an error.

It then initializes two registers, the CTA id and the thread id. It makes no assumptions about the input arguments. All the threads are emulated concurrently so that the blocking by the named barriers can be implemented. This keeps on going until there is a block on the named barriers, and then resumes the thread after the barrier has been completed.

WEFT has a separate mode for **warp-synchronous execution**. Here, the threads in the same warp are executed in the lock step. This mode helps to model the current architecture, which has a behaviour that when any thread arrives at a named barrier, it is equivalent to the entire warp arriving.

When a PTX instruction corresponds to a command in the formal language, the command is recorded in the program that is being emulated. WEFT is sophisticated enough to keep track of the data exchanged through shared memory.
This emulation continues until either all the threads have successfully returned without any errors, or if a deadlock has arisen, or finally, if WEFT is unable to emulate that instruction. Deadlocks are immediately reported to the user.
In the case that the instruction is unable to be emulated, the reason is because of no access to the inputs of the CTA, leading to no dynamic data in the frame buffer memory. WEFT is not smart enough to overlook irrelevant instructions, such as floating point math. This one of the limitations of WEFT. This eventually is returned as an error. As an

assumption, we consider that for all warp specialized kernels, WEFT successfully generates straight line thread programs.

Taking a look at the **Optimised WEFT verification algorithm**. It goes like this:

- WEFT generates thread programs for a CTA.
- It then checks if the programs are well synchronised or not, and if the shared memory is race condition free.
    - The modified WEFT algorithm, allows scaling to much more complex kernels.
    - This scaling isn't easy due to the property of transitive closure computation, which can require the verification of hundreds of millions of combinations of the commands.

The **optimisation** is approached like this:

1. WEFT makes use of a **barrier dependence graph.** Each node of the graph is related to its predecessor. All read-write commands are omitted. The emulation of the CTA takes place with only the synchronisation commands.
2. We use the above paradigms to check if the CTA is well synchronised. All the memory accesses are ignored.
3. The barrier dependency graph is then constructed. One barrier is defined as one node. These are dynamic barriers. If the program is well synchronised indeed, the dependency graph will be a directed acyclic graph or **DAG**.
4. WEFT then needs to check if the memory access are race free. It is near impossible to check this condition for all possible commands. Thus WEFT employs the use of constant time race tests. This involves the computation of latest *happens-before* and the earliest *happens-after* points that are reachable in every thread from every point in the program.
5. After that, it is easy to determine if a race condition exists.This gives us an idea of all the commands that run in parallel. This check is run for only the nodes of the dependency graph, not every set of programming points, making WEFT much more efficient and effective. WEFT avoids heavy computation by sorting the barrier graph using topological sort. This results in a **time complexity of O(D^2)** for *D* dynamic barrier nodes. This results in fast computation. Generally, a fixed number of barriers (16) are executed simultaneously. For a kernel with *N programs* the **space complexity is of the form *O(DN).*** As is visible from this, this complexity is independent of the number of kernel commands. For most kernels, this is around 10GB of space, which isn't an issue for most modern computing devices.
6. WEFT then examines all pairs of shared memory accesses to common addresses, which involve at least one write operation, to check if the race condition is still present. This happens in a constant time complexity.

The Experimental Setup and findings:

In Total, 13 CudaDMA kernels, along with 13 Singe compiler kernels were tested.
The performance of WEFT is limited by the memory latency and memory bandwidth. This depends on the processor being used as well the specific kernel being tested. As found experimentally, the ideal settings involved the use of 4 threads, 2 per NUMA. The measure of performance here is the time required to verify the kernel and the memory required to do so.
Few kernels are not well synchronized, and although an error will be returned, it ‘s hard to determine the cause leading to the kernel not being well synchronised. This is one of the limitations of the WEFT.
Some of the findings are listed below:

1. WEFT is able to scale upto 8k dynamic barriers and upto 14 million commands, and a kernel with this magnitude of complexity will require a mere 15 minutes to verify.
2. The algorithm is sound, meaning that all the errors will be reported back, and these reports can be trusted.
3. Every error reported by WEFT must be investigated as this may lead to other unforeseen issues later.
4. The time complexity of the algorithm in use is a small polynomial, which means that the execution time is predictable.
5. Two types of race conditions are encountered:
    a. Benign race conditions
    b. Harmful race conditions
6. Reverse Time Migration (RTM) kernels from CudaDMA suite of kernels had benign races, which were as a matter of fact intentional by the develope. This is due to multiple threads loading the same value from the working memory and then writing it to the same location. This leads to some magnitude of performance degradation.
7. Two phase RTM kernels were found to have bug which involved write after read operations. Incidentally, the discovery of this bug resolved a 2 year old case.
8. 2 singe kernels also had the issue of write after read races. Involving kepler architecture, both the DME and Heptene chemistry kernels were a victim of this bug. The cause of this bug was an incorrect heuristic being used.
9. Despite extensive testing, these write after read cases never manifest in practice in current GPU architectures.

Criticism on the model:

1. Where is the Generation information stored and shared? How long does it retain its values i.e does it reset to 0 after a while? If the above questions remain unanswered it can degrade the performance due to overhead of variables stored and shared.
2. The model works on PTX instructions on NVIDIA. It doesn't have any base for AMD GPUs. This leads to a smaller domain of developers collaborating on this.
3. The proposed model needs the flow of progress beforehand itself. Because it needs to make the static *happens before* relationships beforehand itself which it then uses to check if there is any problem, based on the three parameters as described above.
4. The concept of using named barriers is slow. Faster threads in a warp have to wait for the slower ones so that the number becomes equal to the number expected by the barrier.
5. The root cause of the error isn't found by the WEFT verification, as it will only inform that there is an issue at a certain program point in the emulation.

Limitations of the paper:

- While there is significant parallelism in the kernel, the use of named barriers constrains the synchronization so that it is both deterministic and known at compile-time. Each thread executes straightline code with no dynamic branches or loops
- No claim of handling inter-CTA synchronization used by persistent uber-kernels such as OptiX. Does not consider software level intra-CTA synchronization as well
- Symmetry assumption that all CTAs execute the same program (albeit with different input data) and therefore only need to model and verify a single CTA to establish correctness
- Does not attempt to reason about data passed between threads through global memory, both because of the weak semantics and because global memory is not used for communication in any of the warp-specialized kernels of the time
- Abstract thread programs for each thread, where everything has been abstracted away except the instructions required to reason about synchronization and shared memory accesses
- The read and write commands are assumed to play no role in the semantics of named barriers
- Does not assume nor require warp-synchronous execution. Warp-synchronous execution is the assumption that all threads within a warp will execute in lock-step.

SUMMARY OF REFERENCES

- **Accelerating CUDA Graph Algorithms at Maximum Warp**
  Graphs are powerful data representations favoured in many computational domains. Modern GPUs have recently shown promising results in accelerating computationally challenging graph problems but their performance suffers heavily when the graph structure is highly irregular, as most real world graphs tend to be. The paper revealed that the poor performance is caused by work imbalance and is basically due to the discrepancy between the GPU programming model and the underlying GPU architecture.
  This paper proposed a new warp-centric programming method that exposes the underlying GPU architecture to users. The paper proceeded with the methods implemented to improve the performance of applications with highly imbalanced workloads.

  The paper concludes by giving concrete results on how GPUs are under-utilized and suffer from workload imbalance for real-world graph instances. The method proposed by the authors of the paper, when applied to graph algorithms resulted upto 9 times speedup when compared to the traditional GPU implementations. The main understanding from the paper is that it is generally difficult to write programs for it to run on any GPU architecture with a good level of performance and to escape race conditions and deadlocks.
  Reference:https://stanford-ppl.github.io/website/papers/ppopp070a-hong.pdf

- **Improving GPU Performance via Large Warps and Two-Level Warp Scheduling**
  Due to their massive computational power, GPUs are a popular platform for executing general purpose parallel applications. GPU programming allows user to create thousands of threads, each executing the same computing kernel. GPUs exploit this parallelism in two ways. First, threads are grouped into fixed sized batches known as warps, and second, many such warps are concurrently executed on a single GPU core. Despite this, the resources on GPU are still underutilized, resulting in performance far short of what could be delivered.
  To improve GPU performance, computational resources must be more effectively utilized. To accomplish this, the paper produced few ideas which are related to how warps can be arranged in different ways in the scheduler. The paper highlights how the ideas proposed result in performance improvements and a better utilization of GPU resources. An important insight to note with respect to the objective of the main paper (Producer-Consumer Utilization in GPUs) is that a constructive way to utilize a GPU in an effective way is proceeding with warp based kernels.
  Reference: https://users.ece.cmu.edu/~omutlu/pub/large-gpu-warps_micro11.pdf

- **Leveraging Warp Specialization for High Performance on GPUs**
  The Paper presents a Domain Specific Language(DSL) compiler that leverages wrap specialization to produce high performance code for GPU. The ideology in consideration is that most users try different methods and implementations to enhance their code so as to completely utilize the GPU resources however the effort of developing code in a defined way could be bypassed by creating a compiler (like Singe) which can partition computations into sub-computations which are then signed to different warps within a thread block. Fine-grain synchronization between warps is performed efficiently in hardware using producer-consumer named barriers.Partitioning computations using warp specialization allows Singe to deal efficiently with the irregularity in data access pattern and computation.
  The paper further goes on to explain the working of Singe, its internals and the performance upgrade achieved. The paper concludes by stating that using a warp-specialized DSL compiler, the kernels are up to 3.75 times faster than previously optimized purely data-parallel GPU code.
  Reference: https://cs.stanford.edu/~sjt/pubs/ppopp14.pdf

- **CUDA**
  When a CUDA kernel is launched on a GPU, a collection of thread-blocks or cooperative thread arrays (CTAs) is created to execute the program. A GPU program has an arbitrary number of non-interfering CTAs. Each CTA is composed of up to 1024 threads. The hardware inside of the GPU dynamically assigns CTAs to one of the streaming multiprocessors (SMs) inside of the GPU. To communicate between threads within the CTA, CUDA provides an on-chip software-managed shared memory visible to all the threads within the same CTA.To coordinate access to shared memory, CUDA supports the syncthreads primitive, which performs a CTA-wide blocking barrier. Synchronizing between threads within the same CTA is the only synchronization supported by CUDA because threads within the same CTA are the only ones guaranteed to be executing concurrently.

FURTHER RESEARCH ON THE INTERNET

- **What is a Thread-Block?**
  A thread block is a programming abstraction that represents a group of threads that can be executed serially or in parallel. For better process and data mapping, threads are grouped into thread blocks. The number of threads varies with available shared memory
  Reference: https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)

- **What are Streaming multiprocessors?**
  Each architecture in GPU (say Kepler or Fermi) consists of several SM or Streaming Multiprocessors. These are general purpose processors with a low clock rate target and a small cache. An SM is able to execute several thread blocks in parallel. As soon as one of its thread blocks has completed execution, it takes up the serially next thread block. In general, SMs support instruction-level parallelism but not branch prediction.
  Reference: https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)

- **What are Warps?**
  On the hardware side, a thread block is composed of 'warps'. A warp is a set of 32 threads within a thread block such that all the threads in a warp execute the same instruction. These threads are selected serially by the SM.
  Once a thread block is launched on a multiprocessor (SM), all of its warps are resident until their execution finishes. Thus a new block is not launched on an SM until there is sufficient number of free registers for all warps of the new block, and until there is enough free shared memory for the new block.
  Consider a warp of 32 threads executing an instruction. If one or both of its operands are not ready (e.g. have not yet been fetched from global memory), a process called 'context switching' takes place which transfers control to another warp. When an instruction has no outstanding data dependencies, that is, both of its operands are ready, the respective warp is considered to be ready for execution. If more than one warps are eligible for execution, the parent SM uses a warp scheduling policy for deciding which warp gets the next fetched instruction.
  Reference: https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)

- **What is CudaDMA?**
  - An API for efficiently copying data from global to shared memory
  - Allows experts to implement optimized tuned code, accessible through a simple API
  - Decouples size & shape of thread block from size & shape of data
  - Works with or without warp specialization

- **What is PTX?**
  Parallel Thread Execution is a pseudo-assembly language used in Nvidia's CUDA programming environment. The nvcc compiler translates code written in CUDA, a C++-like language, into PTX, and the graphics driver contains a compiler which translates the PTX into a binary code which can be run on the processing cores.
  Reference : https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

- **What is a static dataflow graph?**
  Designs that use conventional memory addresses as data dependency tags are called static dataflow machines. These machines did not allow multiple instances of the same routines to be executed simultaneously because the simple tags could not differentiate between them.
  Designs that use content-addressable memory (CAM) are called dynamic dataflow machines. They use tags in memory to facilitate parallelism.

INSIGHTS

As GPU usage continues to expand into new application domains with more task parallelism and larger working sets, warp specialization will become increasingly important for handling challenging kernels that do not fit the standard data-parallel paradigm.
This paper introduces two important concepts in GPU programming: Warp specialised kernels and named barriers, along with a verifier known as WEFT.
GPUs are now an established general purpose computing platform for applications that require significant computational resources and memory bandwidth. To perform synchronization between different warps, warp-specialized kernels use the producer-consumer named barriers available in PTX on NVIDIA GPUs.
The verification of the kernels (by WEFT) is done based on 3 properties: deadlock freedom, safe barrier recycling and race freedom.
The WEFT algorithm is sound, meaning that all the errors will be reported back, and these reports can be trusted. The time complexity of the algorithm in use is a small polynomial, which means that the execution time is predictable.
CUDA is a data parallel programming model that enables GPU acceleration needed for computation. While CPUs ensure minimum latency, GPUs allow for maximum throughput through fine grain pipelining.
The performance of WEFT is limited by processor and the bandwidth provided to it. Some of the common consumer level GPUs from NVIDIA are GTX 1080Ti having 3584 CUDA cores, the RTX 2080Ti having 4352 CUDA cores and Quadro GP100 having 3584 CUDA cores respectively. The number varies greatly and so does the performance.
A significant limitation of the WEFT model is the inability to find the root cause of an error that arises in the process of emulation.

Multiple inefficiencies have been found in various commercially used codes using WEFT, and hence this should be used as a standard check across all relevant programs.

CONCLUSION AND FUTURE DIRECTION

GPUs are the future of parallel processing as hardware acceleration is improving computational efficiency in many modern applications commercially, for example Data Mining for cryptocurrency investments. General purpose programming languages such as C and Python will still dominate but for GPU accelerated supercomputing, the methods as discussed in this paper will overtake others as more creativity is put into exploiting the inherent parallelism of GPU and the sequential offload capabilities of CPUs.

It would be interesting to see how AMD manages to keep up with Nvidia in this field concerning their dedicated GPUs, which are also available for consumer use, at a lower price. Such a programming paradigm shouldn't be limited by the fact that it will run only on the proprietary CUDA architecture, and should be available for open source development.

APPENDIX 2:Copy of Paper(s) Read as Marked by Team member or notes made

1. Karimi, Kamran & Dickson, Neil & Hamze, Firas. (2010). A Performance Comparison of CUDA and OpenCL. Computing Research Repository - CORR. arXiv:1005.2581.
2. Hong, Sungpack & Kim, Sang Kyun & Oguntebi, Tayo & Olukotun, Kunle. (2011). Accelerating CUDA graph algorithms at maximum warp. Sigplan Notices - SIGPLAN. 46. 267-276. 10.1145/2038037.1941590.
3. Narasiman, Veynu & Shebanow, Michael & Lee, Chang Joo & Miftakhutdinov, Rustam & Mutlu, Onur & Patt, Yale. (2011). Improving GPU performance via large warps and two-level warp scheduling. Proceedings of the Annual International Symposium on Microarchitecture, MICRO. 10.1145/2155620.2155656.
4. Bauer, Michael & Treichler, Sean & Aiken, Alex. (2014). Singe: Leveraging Warp Specialization for High Performance on GPUs. ACM SIGPLAN Notices. 49. 119-130. 10.1145/2555243.2555258.