PES UNIVERSITY, BENGALURU

FINAL SEMESTER ASSESSMENT (FSA) – B.TECH. (CSE) – IV SEM
SESSION: AUGUST – DECEMBER, 2019

UE17CS302 – INTRODUCTION TO OPERATING SYSTEMS

Assignment Report

On

"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.edu**

TEAM:

SL No	Name	SRN	Roll No
1	Akanksha Tonne	PES1201700275	19
2	Rachana Sudhindra Dani	PES1201700950	29
3	Rishika Malla	PES1201700162	10
4	Radhika Sadanand	PES1201701702	54
5	G.Rachana Rao	PES1201700960	30

TABLE OF CONTENTS

SUMMARY

1.Abstract And Introduction:

Previous efforts have focused on kernels alone written within data-parallel GPU model thereby not giving preference to higher performance. Primarily, it focused on more complex kernel i.e. warp-specialized kernels. The main objective of the paper given to us is to define what it means for a warp specialized program to be correct and to measure the correctness of such programs. There are algorithms for verifications.
The paper also presents the WEFT algorithm to verify the correctness of the warp specialized code.

KEYWORDS to be understood:

Data parallel GPU programming model : It is a computing model where all threads execute the same set of instructions on different arrays or data matrices or distributed components. For example CUDA is one of those models.

Operational semantics : A category of formal programming language semantics in which certain desired properties of a program, such as correctness, safety or security, are verified by constructing proofs from logical statements about its execution and procedures, rather than by attaching mathematical meanings to its terms.

Thread blocks: In parallel computing many light weight processes can be combined together to get what are called as thread blocks. The threads in a given thread blocks can be either executed serially or in parallel. In CUDA, The threads in a block run on same stream processor and they communicate with each other via shared memory, barrier synchronization or atomic operations.

Barrier synchronization: That some threads in the parallel execution should be barred or stalled until all the threads executing in that thread block can reach the same point after which all the threads continue to run at that point.

Warp-Specialized : Kernel with warp specialization improves performance by allowing the compiler to do a better job of instruction scheduling and resource allocation. Warp specialization separates memory and compute operations into two different instruction streams.
Why are we leaving behind traditional data parallel model? because it fails at some points that are very useful while achieving our aim. So those points are Large working sets, Irregular Computations and Irregular data accesses. Warp specialization can be used as an alternative programming model for mapping irregular and large working set applications onto GPUs.

CUDA:

CUDA is a general purpose programming language for programming GPUs. Each CUDA-enabled GPU consists of a collection of streaming multiprocessors (SMs). In this model, kernels execute using a streaming paradigm; load data on-chip, perform a computation on thee data, and write back off-chip. A barrier is used to coordinate and synchronize the execution of these phases and the threads in the thread block respectively. Data Parallel kernels in which all threads perform roughly the same thread-block perform different computations.

Different computations are assigned to groups of 32 threads i.e. warps by warp-specialized kernels within the same thread block in order to achieve maximizing bandwidth in CudaDMA and fitting extremely large working sets in singe. For synchronization between warps, we use named barriers which is a part of core language.

Three important properties to check for warp-specialized kernels.

1. Deadlock Freedom: making sure that the use of named barriers doesn't result in deadlocks

2. Safe barrier Recycling: making sure whether or not the ID's named barriers are properly re-used since it is the limited physical resource.

3. Race Freedom: making sure synchronization is race free.

No current GPU verification tools are capable of checking code containing named barriers. Using WEFT, which is a verifier that is sound, complete and efficient.

2. Background and Motivation

This elaborates out the need for verification of code.

2.1 GPU Architecture and Programming:

Different frameworks are not consistent over the same programming model and they provide variations on the same data parallel model.

A collection of thread blocks is called CTA's is created to execute the program. Named barriers are important for the construction of warp-specialized kernels. CUDA provides an on-chip software-managed shared memory that can be seen by all the threads within same Caveach CTA is assigned to one of the streaming multiprocessors inside of the GPU.

Syncthreads performs CTA wide-blocking barrier. Logically CTA's represent a flat collection of threads. Hardware makes scheduling decisions within an Somewhen a warp scheduled all the threads execute the same instruction. The SM first executes the warp for not taken branch then executes the taken branch with the complementary set of threads in the warp masked. The resulting execution of branches within a warp is called branch divergence and is also a common source of performance degradation. There will be no performance degradation as long as all threads within a warp continue to execute the same instruction.

Warps

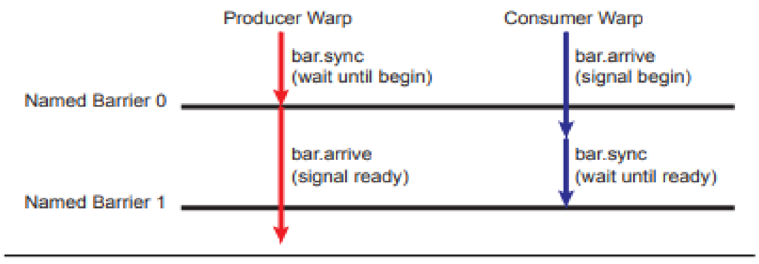
Each CTA is decomposed into warps where a warp is 32 contiguous threads in the same CTA. System Manager performs scheduling at warp-granularity:

- Each warp has its own program counter
- All threads in a warp execute in lock-step

2.2 Warp Specialization and Named Barriers

A warp-specialized kernel is one in which individual warps are assigned different computations via control divergence contingent upon warp Ids. It differentiates warps into compute and DMA. CudaDMA specializes warps in compute and DMA warps to optimize the loading of data in shared memory. Singe Compiler-handles mapping of static data flow graph onto CTA and it can leverage task parallelism on GPUs in the absence of significant data-parallelism, and fit very large working sets into on-chip memories by blocking data for the GPU register file. Warp specializations kernels communicate through shared memory. They are asymmetric, with one or more warps acting as producers, and one or more other warps acting as consumers. DMA warps act as producers while compute warps act as consumers in CudaDMA. They are also independent of warp count.

GPU supports named barriers through instructions: 2 instructions – sync and arrive
A sync instruction causes the threads in a warp arriving at the barrier to block until the barrier completes. Alternatively, an arrive instruction allows a warp to register arrival at a barrier and immediately continue executing without blocking. Producer- Consumer synchronization can be maintained by the former instructions.



Using arrive and sync operations, programmers can encode producer-consumer relationships in warp-specialized programs. The figure above illustrates using two named barriers to coordinate movement of data from a producer warp (red) to a consumer warp (blue) through a buffer in shared memory. The producer warp first waits for a signal from the consumer warp that the buffer is empty. The consumer warp signals the buffer is ready by performing a nonblocking arrive operation. Since the arrive is non-blocking, the consumer warp is free to perform additional work while waiting for the buffer to be filled. At some point the consumer warp blocks on the second named barrier waiting for the buffer to be full. The producer warp signals when the buffer is full using a non-blocking arrive operation on the second named barrier. It is important to note that named barriers support synchronization between arbitrary subsets of warps within a CTA, including allowing synchronization between a single pair of warps as in this example.

```
__global__ void __launch_bounds__(64,1) example_deadlock(void) {
  assert(warpSize == 32);
  assert(blockDim.x == 64);
  int warp_id = threadIdx.x / 32;
  if (warp_id == 0) {
  // bar.sync (barrier name), (participants);
    asm volatile(”bar.sync 0, 64;”);
    asm volatile(”bar.arrive 1, 64;”);
  } else {
    asm volatile(”bar.sync 1, 64;”);
    asm volatile(”bar.arrive 0, 64;”);
  }
}
```

Listing 1. Example of deadlock using named barriers.

In the above code, the 2 threads one belonging to warp with id 0 and other with warp with id 1 are running parallel. The both warps are calling on sync which is a blocking operation. Warp with id 0 at named barrier 0 is waiting for the warp with warp id 1 to register itself at named barrier 0 by calling arrive. On the other hand, warp with id 1at named barrier 1 is waiting for the warp with warp id 0 to register itself at named barrier 1 by calling arrive. Hence, one thread waits for other in a cyclical fashion resulting in a deadlock.

2.3 Motivating Kernel

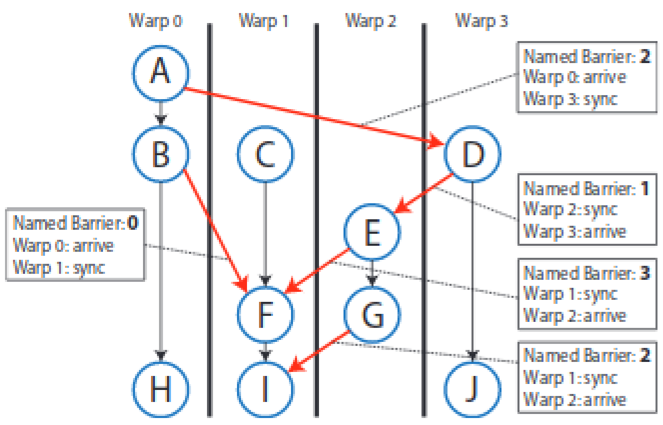


Figure 2. Motivating warp specialization example.

The graph is given for the mapping of static dataflow graph onto warps. They are straight codes and when there is a edge between 2 warps, the warps use the named barrier access the shared memory and maintain synchronization.

Once the named barrier completes, it is reinitialized so that it can be used again. Different generation(each use of a named barrier) can have different number of participants. These also create failures like deadlocks unlike CUDA.

Properties of the warp specialized kernels:

- Reuse of named barrier

To avoid conflicting arrivals at named barriers from participants of different generations, 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. In the figure 2 above, named barrier 2 is being reused. The happens-before relationships is established by the paths D -> E -> F -> I and D -> E -> G. The 2 paths use named barriers 1 and 3. The happens-before relationship ensures the participating warps in the next generation of barrier 2 both have a happens-before relationship with the previous generation of barrier 2 and hence barrier 2 can be safely recycled.

When all barriers successfully execute and are reused correctly, it is called well synchronization kernel but checking for this property in a CUDA code is impractical as it consists of more than 10K lines.

- Synchronization pattern is completely static

The named barriers constraints parallelism in the kernel so that it is both deterministic and known at compile-time. It executes straight line code without having any dynamic branches or loops. These properties shows that one will not obtain complete solutions in the presence of arbitrary control flow and synchronization. Another reason for not using dynamic branches in GPU's is that it is highly expensive.

<u>3.OPERATIONAL SEMANTICS</u>

We are checking for deadlock-freedom, proper named barrier recycling, and shared memory data race freedom within individual CTA's. We rely on symmetric assumption that all CTA's execute the same program. The validation is limited to access to a GPU's shared memory.

3.1 Syntax

A GPU program has an arbitrary number of non-interfering CTA's each consisting of (32 to 1024 threads) that can synchronize access to shared memory. We denote

Thread – P

CTA-T

CTA with threads P_1, P_2, P_3…….P_N - $P_1||P_2||P_3$….. $||P_N$

Shared memory variables –g

Thread identifier- id

Named barriers – B (typically 16)

Specific barrier name – b

Command – c

Program point -ή

We use i to range over thread identifiers

A thread has the following grammar

$$
\begin{array}{lcl}
P & ::= & \texttt{return} \mid c; P \\
c & ::= & \texttt{read}\ g \mid \texttt{write}\ g \\
  &   & \mid \texttt{arrive}\ b\ n \mid \texttt{sync}\ b\ n
\end{array}
$$

A thread has sequence of commands (straight line code). In each command c a thread can either read/write from a shared memory location or perform a synchronization operation. Blocking synchronization operation is sync and non-blocking is arrive. The read-write commands play no role in semantics of named barriers. The first argument b represents the name (ID) of a named barrier, and the second argument n represents the expected number of threads to register at this generation of the barrier. Successive commands are separated by a semicolon. A thread terminates by executing a return. Sync 0 N is sync across all threads in CTA on barrier 0. Program point $\dot{\eta}$ and command $c_{\dot{\eta}}$ correspond if η is the program point just after $c_{\dot{\eta}}$. The first program point of thread i is denoted by η^i_I and the last by η^i_F. Our semantics does not assume warp-synchronous execution that is all threads within a warp execute in lock-step.

3.2 State

state ‘s’ of CTA consists of the following

Enabled map $\dot{\varepsilon}$ - It maps thread identifiers to Booleans indicating whether a thread is enabled or not. Threads are disabled when they block on barrier.

Barrier map B –It maps the barrier names to a triple consisting of a list I of threads that have synced at the barrier, a list A of threads that have arrived at the barrier, and the thread count, describing the number of threads the barrier is expecting to register if it is configured.

The thread count of unconfigured barriers is denoted by $\perp$. An empty list of thread identifiers is denoted by [] and “::” adds a thread to a list. The number of elements in a list L is denoted by |L|. For a map A, we use A[x/y] to denote the map that agrees with A on all inputs except y and maps y to x. Similarly, A[x/Y] denotes a map that agrees with A on all inputs that are not in Y and maps all $y \in Y$ to x. The function ite(e1,e2,e3) stands for “ if e1 then e2 else e3”.

The initial state I has $\forall i.E(i) = true$ and $\forall b.B(b) = ([\,], [\,], \perp)$. All threads are enabled and ready to execute, no thread has registered at any barrier, and all barriers are unconfigured. We use done to denote a CTA with no more commands to execute.

3.3 Semantics

We consider a CTA T with threads P_1, P_2, P_3…….P_N executing in parallel. The rules have the following form:

$$\mathcal{E}, \mathcal{B}, T \rightsquigarrow \mathcal{E}', \mathcal{B}', T'$$

$$\mathcal{E}, \mathcal{B}, i, P \rightsquigarrow \mathcal{E}', \mathcal{B}', i, P'$$

A CTA/thread program executing in some state takes a single step and results in a new CTA/thread program and a new state.

$$\frac{\begin{array}{c}\forall b. \Big(\mathcal{B}(b) = ([], [], \bot) \lor \big(\mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \land |\mathcal{I}| + |\mathcal{A}| < n\big)\Big) \\ \mathcal{E}, \mathcal{B}, i, P_i \rightsquigarrow \mathcal{E}', \mathcal{B}', i, P_i'\end{array}}{\mathcal{E}, \mathcal{B}, P_1 || \ldots || P_i || \ldots || P_N \rightsquigarrow \mathcal{E}', \mathcal{B}', P_1 || \ldots || P_i' || \ldots || P_N}$$

For all barriers b, either b is unconfigured or needs to register more threads. We non-deterministically choose a thread in CTA and execute it for one step. If we have N thread programs executing concurrently, anyone can make step from P^i to P_i’

If some barrier has registered required number of threads, then wake up all threads registered at barrier

$$\frac{\begin{array}{c}\mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \quad |\mathcal{I}| + |\mathcal{A}| = n \\ T = P_1 \mid\mid \ldots \mid\mid P_N \quad \forall i. P_i = ite(i \in \mathcal{I}, \mathtt{sync}\ b\ n; P_i', P_i') \\ T' = P_1' \mid\mid \ldots \mid\mid P_N' \quad \mathcal{E}' = \mathcal{E}[\mathbf{true}/\mathcal{I}]\end{array}}{\mathcal{E}, \mathcal{B}, T \rightsquigarrow \mathcal{E}', \mathcal{B}[([], [], \bot)/b], T'}$$

If n threads registered at b, enable all threads blocked on b, change b to an unconfigured state, and update the control of all threads in I. This rule recycles barrier.

Execution terminates when all threads execute a return

$$\frac{}{\mathcal{E}, \mathcal{B}, \mathtt{return} \mid\mid \ldots \mid\mid \mathtt{return} \rightsquigarrow \mathcal{E}, \mathcal{B}, done}$$

The first synchronization operation configures the barrier and determines the number of threads required.

$$\frac{\mathcal{B}(b) = ([], [], \bot) \qquad \mathcal{B}' = \mathcal{B}[([], [] :: id, n)/b]}{\mathcal{E}, \mathcal{B}, id, \mathtt{arrive}\ b\ n; c \rightsquigarrow \mathcal{E}, \mathcal{B}', id, c}$$

The arriving thread configures the barrier with the thread count (n) and gets added to the list of arrives. Programs with invalid thread count are rejected in the pre-processing phase before execution phase begins.

If the first thread to register with a barrier is sync, the thread updates the enabled map and is added to list of blocked threads.

$$\frac{\begin{array}{c}\mathcal{E}(id) = \mathbf{true} \quad \mathcal{E}' = \mathcal{E}[\mathbf{false}/id] \\ \mathcal{B}(b) = ([], [], \bot) \quad \mathcal{B}' = \mathcal{B}[([] :: id, [], n)/b]\end{array}}{\mathcal{E}, \mathcal{B}, id, \mathtt{sync}\ b\ n; c \rightsquigarrow \mathcal{E}', \mathcal{B}', id, \mathtt{sync}\ b\ n; c}$$

When a non-blocking arrive is executed at barrier, the control and the barrier map are updated.

$$\frac{\mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \quad 0 < |\mathcal{I}| + |\mathcal{A}| < n}{\mathcal{E}, \mathcal{B}, id, \texttt{arrive}\ b\ n; c \rightsquigarrow \mathcal{E}, \mathcal{B}[(\mathcal{I}, \mathcal{A} :: id, n)/b], id, c}$$

On reaching sync, the thread is disabled and gets added to the list of block threads.

$$\frac{\begin{array}{c}\mathcal{E}(id) = \text{true} \quad \mathcal{E}' = \mathcal{E}[\text{false}/id] \quad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \\ \mathcal{B}' = \mathcal{B}[(\mathcal{I} :: id, \mathcal{A}, n)/b] \quad 0 < |\mathcal{I}| + |\mathcal{A}| < n\end{array}}{\mathcal{E}, \mathcal{B}, id, \texttt{sync}\ b\ n; c \rightsquigarrow \mathcal{E}', \mathcal{B}', id, \texttt{sync}\ b\ n; c}$$

For each generation of named barrier, the expected number of participants must match with for each thread program that registers with the barrier.

If too many threads reach a barrier then transition to state err

$$\frac{\mathcal{E}(id) = \text{true} \quad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \quad |\mathcal{I}| + |\mathcal{A}| = n}{\mathcal{E}, \mathcal{B}, id, \texttt{sync}\ b\ n; c \rightsquigarrow \texttt{err}, id, \texttt{sync}\ b\ n; c}$$

The transition to the error state occurs when there is a mismatch on thread count (similar for arrive).

$$\frac{\mathcal{E}(id) = \text{true} \quad \mathcal{B}(b) = (\mathcal{I}, \mathcal{A}, n) \quad n \neq m}{\mathcal{E}, \mathcal{B}, id, \texttt{sync}\ b\ m; c \rightsquigarrow \texttt{err}, id, \texttt{sync}\ b\ m; c}$$

If any thread produces an error then the execution terminates in an error state.

$$\frac{\mathcal{E}, \mathcal{B}, i, P_i \rightsquigarrow \texttt{err}, i, P_i}{\mathcal{E}, \mathcal{B}, P_1 || \ldots || P_i || \ldots || P_N \rightsquigarrow \texttt{err}, P_1 || \ldots || P_i || \ldots || P_N}$$

These error productions ensure that execution either reaches done, goes to err, or deadlocks.

4. Algorithm

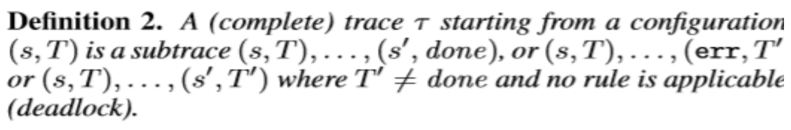
Verifying the correctness of CTA involves guessing happens-before-relationships and using them to prove correctness of properties

4.1 Preliminaries

We use (s, T) to abbreviate $\mathcal{E}, \mathcal{B}, \mathcal{T}$. .A configuration C is a pair of a state s and a CTA T.

Definition 1. *A partial trace or a subtrace is a sequence of configurations $(s_0, T_0), \ldots, (s_n, T_n)$ such that for any two successive configurations we have $(s_j, T_j) \rightsquigarrow (s_{j+1}, T_{j+1})$.*

A complete trace ends in either done, err, or a deadlock



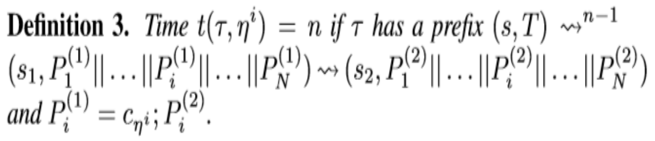
Definition 2. *A (complete) trace τ starting from a configuration (s, T) is a subtrace $(s, T), \ldots, (s', done)$, or $(s, T), \ldots, (\texttt{err}, T'$ or $(s, T), \ldots, (s', T')$ where $T' \neq done$ and no rule is applicable (deadlock).*

C->C' – C evaluates to C' in one step

C->*C' – C evaluates to C' in zero or more steps

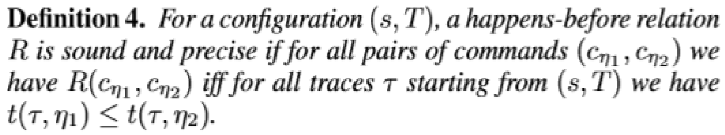
C->^{m}C'- C evaluates to C' in m steps

If η is the program point just after the command cη then we define a quantity t(τ,η) which provides the step at which cη is executed in the trace τ.



Definition 3. *Time $t(\tau, \eta^i) = n$ if τ has a prefix $(s, T) \rightsquigarrow^{n-1}$ $(s_1, P_1^{(1)} \| \ldots \| P_i^{(1)} \| \ldots \| P_N^{(1)}) \rightsquigarrow (s_2, P_1^{(2)} \| \ldots \| P_i^{(2)} \| \ldots \| P_N^{(2)})$ and $P_i^{(1)} = c_{\eta^i}; P_i^{(2)}$.*

Time t(T,.) provides the execution step when read , write and arrive are executed in T. Sync , the step is when the corresponding barrier recycled . Time helps us to figure out happens before relationships.

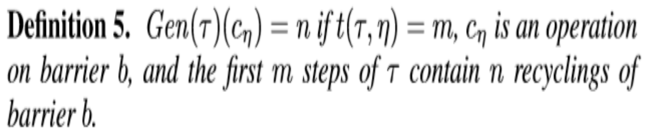


Definition 4. *For a configuration (s, T), a happens-before relation R is sound and precise if for all pairs of commands (c_{η_1}, c_{η_2}) we have $R(c_{\eta_1}, c_{\eta_2})$ iff for all traces τ starting from (s, T) we have $t(\tau, \eta_1) \leq t(\tau, \eta_2)$.*

Happens before relationship includes commands that execute simultaneously i. e sync commands are included in happens before relationship

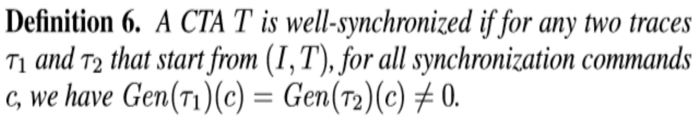
Gen(T) maps a synchronization command to a generation ID . Example : A gen Id of 2 for command sync 0 n indicates that the command was used to register on barrier 0 after the barrier is recycled.

The gen ID of unexecuted commands is set to 0.

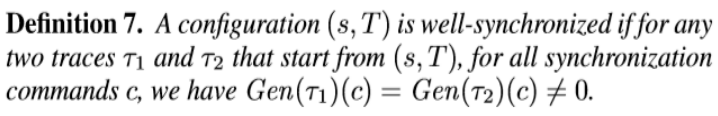


Definition 5. *$Gen(\tau)(c_\eta) = n$ if $t(\tau, \eta) = m$, c_η is an operation on barrier b, and the first m steps of τ contain n recyclings of barrier b.*

If the generation mapping for all traces in the CTA are same , then it is said to be well-synchronized.



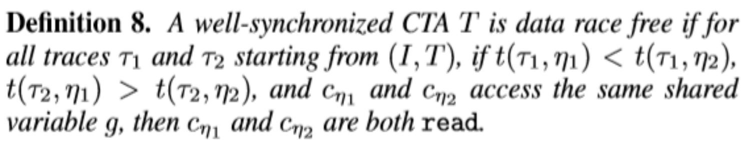
Definition 6. *A CTA T is well-synchronized if for any two traces τ_1 and τ_2 that start from (I, T), for all synchronization commands c, we have $Gen(\tau_1)(c) = Gen(\tau_2)(c) \neq 0$.*



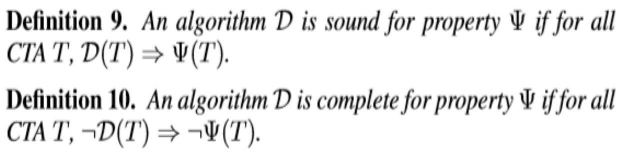
Definition 7. *A configuration (s, T) is well-synchronized if for any two traces τ_1 and τ_2 that start from (s, T), for all synchronization commands c, we have $Gen(\tau_1)(c) = Gen(\tau_2)(c) \neq 0$.*

The non-zero check takes care that no trace of a well-synchronized configuration can deadlock or go to an error. The check also considers safe barrier recycling and deadlock freedom. It also ensures that same commands in traces synchronize together.

Race freedom:


Definition 8. *A well-synchronized CTA T is data race free if for all traces τ_1 and τ_2 starting from (I, T), if $t(\tau_1, \eta_1) < t(\tau_1, \eta_2)$, $t(\tau_2, \eta_1) > t(\tau_2, \eta_2)$, and c_{η_1} and c_{η_2} access the same shared variable g, then c_{η_1} and c_{η_2} are both* `read`.

If Two commands access the same shared variable g and they do not have a happens before relationship between them , then both should be reads .If there is any write , they get into data races.


Definition 9. *An algorithm $\mathcal{D}$ is sound for property Ψ if for all CTA T, $\mathcal{D}(T) \Rightarrow \Psi(T)$.*

Definition 10. *An algorithm $\mathcal{D}$ is complete for property Ψ if for all CTA T, $\neg\mathcal{D}(T) \Rightarrow \neg\Psi(T)$.*

4.2 Property checking.

The main idea behind checking the well synchronized property of the CTA is to statically generate one trace of execution of the CTA including many warps, assign the generations to each command of the warp that makes use of the thread barrier. Run the verification algorithm that generated the happens before relation followed by which the transitive closure of the relation is found. In the transitive closure of the relation, the property that every pair of commands has the happens before relation is checked. If the happens before relation between the two commands in the pair exists and all the traces of execution of the CTA (that can be generated statically) assign the same generations to the commands in the trace, then the well synchronized property holds else the well synchronized property is not satisfied.

Formally, for the given trace they first generate the Gen(τ) to construct the static happens before relation R. Initially, the relation R is empty. The commands that are guaranteed to execute sequentially in a given warp are added to the R. For example, if c1 and c2 are two commands such that they belong to the same warp and c1 is a command executed at program time step t1 and c2 executed at time step t2 and t2 occurs after t1 then pair (c1, c2) is added to R. Next, they consider arrive command on barrier b c1 in warp 1, and command on barrier b having same generation as that in c1, c2 in warp 2 such that c1 occurs before c2 then pair (c1, c2) is added to R. If c1 is a sync on barrier b and c2 is a sync on barrier b in warp 2 both having same generations then both the pairs (c1, c2) and (c2, c1) are to be added to R. Finally, they compute the transitive closure of R. And they check for each such command c1 ≡ sync b n with Gen(τ) = k and each c2 with barrier b and Gen(τ) = k+1 and c3, if the pair (c1, c3) does not belong to R then well synchronized property is said to be violated.

5 Verification using WEFT

5.1 Emulation of GPU Program in WEFT

WEFT takes as an input pseudo assembly program PTX for GPU and starts converting it into formal language discussed in section 3. It does so by emulating a thread in CTA until it finds that thread terminates, deadlocks or errors in some state. It concurrently emulates the threads so as to model the blocking. It continues to emulate a thread until the thread blocks on some barrier and then resumes on arrival of all the threads on that barriers (Recycle of the barrier). Another mode of emulation is also provided to simulate the lock step execution of the threads in a given warp but the users have to explicitly opt to this mode. In this way weft validates the kernels that rely on the warp synchronous execution.

Weft maintains a separated program for each thread in the CTA. As soon as the PTX instruction corresponding to the formal language command is encountered it is noted down. WEFT is capable of generating the straight line code by examining the static branch conditions. It is also capable of tracking data exchange through shared memory.

WEFT terminates in one of the following ways:

- Successfully being able to emulate all the instructions and no instruction left for further emulation.
- Encountering deadlock which is said to occur when the program cannot proceed forward (without being able to acquire the resource for example). In this case it also reports the how the deadlock occurred.
- Unable to emulate the required instruction because it doesn't have access to the necessary inputs to the instruction and cannot dynamically generate the inputs though it is capable of doing it.

5.2 Optimized WEFT Verification Algorithm

WEFT verification process:

After programs for each of the threads are generated, WEFT checks if the programs are well synchronised and shared memory is race free.

Steps:

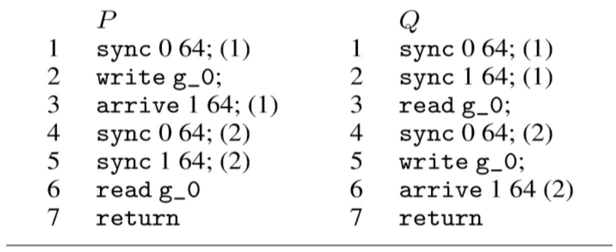
Synchronization check:

1. WEFT omits all read and write commands from the thread programs and simulates one execution of the CTA with only the synchronization commands.
2. WEFT uses the algorithm in Figure 4 to determine if the CTA is well-synchronized.
3. Memory accesses which do not affect synchronization are omitted. Hence, reducing the total number of commands. This makes the computation tractable.

Establishing happens before relationship between the generations of barriers by checking if the barrier dependence graph is a directed acyclic graph

1. Each completed barrier is converted to a single node in the graph and is assigned a generation. These nodes are called dynamic barriers.
2. $C^P{}_i$ denotes the ith command corresponding to the synchronization commands of Pth program.
3. Example: Consider the following program for constructing the barrier dependence graph
4. The nodes of the graph are: n_1 has $c^P{}_1$, $c_1{}^Q$, generation 1; n_2 has $c_3{}^P$, $c_2{}^Q$, generation 1; n_3 has $c_3{}^P$, $c_4{}^Q$, generation 2; n_4 has $c_5{}^P$, $c_6{}^Q$, generation 2

Listing 2. CUDA snippet for the working example.



```
    P                          Q
1   sync 0 64; (1)         1   sync 0 64; (1)
2   write g_0;             2   sync 1 64; (1)
3   arrive 1 64; (1)       3   read g_0;
4   sync 0 64; (2)         4   sync 0 64; (2)
5   sync 1 64; (2)         5   write g_0;
6   read g_0               6   arrive 1 64 (2)
7   return                 7   return
```

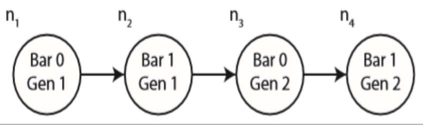


Figure 7. Barrier dependence graph for Figure 5.

After checking for well synchronization and barrier dependence, a check for data races between the shared memory is made.

Presence of hundreds to thousands of millions of commands in the programs, makes computation of transitive closure expensive. Hence, WEFT performs constant time race tests by computing the latest happens-before and earliest happens-after points reachable in every thread program from every program point.

To explain latest happens-before and earliest happens-after points, consider the command $c_2{}^P$. The latest happens-before command is $c_1{}^Q$ and the earliest happens after command is $c_3{}^Q$. All commands between $c_1{}^Q$ and $c_3{}^Q$ can execute in parallel with command $c_2{}^P$.

WEFT first computes the earliest/latest happens-after/before points reachable in every thread program from each node in the barrier dependence graph. Each dynamic barrier initializes its reachability points with its participants and sets the remaining values to $\perp$. The transitive closure of reachability points is then computed over the barrier dependence graph.

WEFT uses topological sorting to compute this relationship for every barrier node. This computation is in order $O(D^2)$ where D is number of nodes in the graph. D is much smaller than the total number of commands hence reduces the time complexity of computation.

This computation produces the barrier intervals for the synchronization commands. For example, the generated barrier interval of $c_1{}^P \in n1$ in thread Q is $(c_0{}^Q, c_2{}^Q)$, where the 0th command is just a placeholder for the beginning of a program. Similarly the barrier interval for $c_2^Q \in n2$ is (c_1^Q, c_3^Q).

5.3 Experiments:

WEFT was evaluated on 13 warp-specialized kernels that use the CudaDMA library and 13 warp-specialized kernels that were emitted by the Singe compiler.

CudaDMA : 65-1561 lines of CUDA Singe compiler: 1684-13245 lines of CUDA

All experiments were run on an Intel Xeon X5680 Sandy Bridge processor clocked at 3.33 GHz with 48 GB of memory divided across two NUMA domains.

All experiments using WEFT were run with four threads (two per NUMA domain).

WEFT reports statistics on each of the kernels including the number of dynamic barriers, total shared memory addresses that are used, total commands in the formal language from Section 3 across all threads, the number of thread programs, and the total race tests that were performed.

Performance is measured by the time required to verify each kernel and the total memory required to perform the verification.

WEFT reports whether the kernel is well-synchronized and race-free.

In the future, the authors anticipate WEFT will be especially valuable in giving feedback for debugging warp specialized kernels that either deadlock or crash due to a violation of the well-synchronized property during development.

WEFT is able to verify most kernels in a few minutes or less making it practical for use by real developers.

Conclusion:

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. The named barriers available on NVIDIA GPUs make warp specialization possible, but require more complex code with potentially difficult to discern bugs. WEFT, a verifier for kernels using named barriers that is sound, complete, and efficient. Soundness ensures that developers do not have to write any tests to check for synchronization errors. Completeness ensures that all violations reported by WEFT are actual bugs; there are no false alarms.

WEFT runs on large, complex, warp-specialized kernels requiring upto billions of checks for race conditions and completes in a few minutes. Using WEFT, 26 kernels were validated and non-trivial bugs were identified.

Appendix 1:

SUMMARY OF REFERENCES:

Wrap specialised kernels: Optimizing the performance of a GPU kernel is primarily about managing constrained resources such as registers, shared memory, instruction issue slots, and memory bandwidth. Warp specialization allows subsets of threads within a **CTA** to have their behaviour tuned for a particular purpose which enables more efficient consumption of constrained resources.

In addition to these explicit techniques, warp specialization improves performance by allowing the compiler to do a better job of instruction scheduling and resource allocation. Warp specialization separates memory and compute operations into two different instruction streams. The compiler's job is greatly simplified by only having to optimize a few metrics in independent instruction streams, rather than multiple performance metrics across a single, mixed stream leading to better machine code. [1]

CTA: CUDA is a general purpose programming language for programming GPUs. **Each CUDA-enabled GPU consists of a collection of streaming multiprocessors (SMs)**. A SM possesses an on-chip register file, as well as an on-chip scratchpad memory that can be shared between threads executing on the same SM. DRAM memory is off-chip, but is visible to all SMs. The CUDA programming model targets this GPU architecture using a hierarchy of threads. Threads are grouped together into thread blocks, also known as cooperative thread arrays (CTAs).[1]

Instead of relying on traditional GPU programming models that emphasize data-parallel computations, warp specialization allows compilers like Singe to partition computations into sub-computations which are then assigned to different warps within a thread block. Fine-grain synchronization between warps is performed efficiently in hardware using producer-consumer named barriers.[2]

Current GPU programming models, such as OpenCL[10] and CUDA[1], support data-parallel computations where all threads execute the same instruction stream on arrays of data. However, the expansion of GPUs into general purpose computing has uncovered many applications which exhibit properties which make them challenging to map onto traditional data-parallel GPU programming models [2]

Places where the traditional data parallel model fails:

Large working sets: In data-parallel model these working sets commonly exceed the small on-chip memory capacity allotted to each thread, resulting in register spilling, low occupancy, and under-utilization of math units.

Irregular computations: Because of irregular computations the data parallel model goes to serial execution instead of parallel computation hence underutilizing resources.

Irregular data accesses: Under many circumstances it is impossible for a data-parallel model to avoid memory divergence and shared memory bank conflicts.[7]

warp specialization can be used as an alternative programming model for mapping irregular and large working set applications onto GPUs.[2]

In the data parallel model, a collection of threads within a thread block all execute the same program over independent elements from arrays of input data. On the hardware, however, a thread block is broken into warps consisting of (typically) 32 threads which serve as the unit of scheduling. Warp specialization exploits the division of a thread block into warps to partition computations into sub computations such that each sub-computation is executed by a different warp within a thread block. Carefully structured programs can handle irregularity by grouping threads into warps such that threads within a warp have good data-parallel behaviour, even if threads in different warps do not.[2]

While there are several APIs for programming GPUs, they all implement variations of the same programming model. We use CUDA as a proxy for the standard GPU programming model as it is the only interface that currently supports the fine-grain synchronization primitives necessary for warp specialization. **CUDA launches grids of thread blocks or cooperative thread arrays (referred to as CTAs for the remainder of the paper) on the GPU.** This abstraction gives the hardware considerable flexibility when executing a CUDA application. In current GPUs, **the threads within a CTA are broken into groups of 32 threads called warps. All threads within a CTA (and therefore also within a warp) execute the same program.** If the threads within a warp diverge on a branch instruction, the streaming multiprocessor (SM) on which the warp is executing first executes the warp with all the threads not taking the branch masked off. After the taken branch is handled, the warp is re-executed for the not-taken branch with the complementary set of threads masked off from executing. **Divergence is severely detrimental to program performance because it serializes potentially parallel thread execution within a warp.** The crucial insight for warp specialization is that while control divergence within a warp results in performance degradation, divergence between warps does not. A warp-specialized kernel is one in which dynamic branches, dependent on each thread's warp ID, create explicit inter-warp control divergence for the purpose of executing different code in each warp. As long as all threads within a warp execute the same instruction stream then the only execution overhead is the cost of the warp-specific branch instructions.[2]

Summary:

The code written for GPU's is verified traditionally using the kernels written within data parallel programming model. No work has been done to verify code written for GPU's using kernels written with wrap-specialised kernels. This paper presents the operational semantics,

(**Operational semantics** is a category of formal programming language semantics in which certain desired properties of a program, such as correctness, safety or security, are verified by constructing proofs from logical statements about its execution and procedures, rather than by attaching mathematical meanings to its terms (denotational semantics) wiki), For the producer consumer named barriers (explained in the coming paragraphs), and explains correctness for a wrap specialised kernel.

They try to give algorithms to prove correctness for wrap-specialised kernels for the most general wrap-specialised programs.

Application which have a high demand for computation resources and memory bandwidth use GPU as an energy efficient and high-performance tool. But due to the complexity of the GPU's memory hierarchy, caches and threads it is very likely that a code written for the GPU will have race conditions and other bugs. Hence writing code for GPU's is a challenging task.

Many attempts have been made to check the correctness of kernels for the GPU's all of them have assumed the kernels written within data parallel programming languages. Here the kernels execute code based on a streaming paradigm: load data on chip, perform a computation on data, and write the results on the chip. To ensure synchronization among threads in a particular block a barrier is used.

In Wrap specialised kernels, the kernel distributes the computation among wraps (division within a thread block(32 threads each)) to achieve performance goals. To achieve the synchronisation between wraps, producer consumer named barriers are used. Named barriers are implemented directly in hardware and support a richer set of synchronization patterns than can be achieved with the standard thread block-wide barrier used by CUDA and OpenCL. Specifically, named barriers allow warp-specialized kernels to encode producer-consumer relationships between arbitrary subsets of warps in a thread block. Importantly, producers do not block on a named barrier, and can continue executing after arriving on a named barrier.

Specifically, 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.

Thread Divergence:

If there are no branches in the thread program then all threads can execute in a lockstep. But if there exists a branch, as mentioned in above paragraphs, threads may diverge and introduce some serialisation, hence reducing the performance.

For example consider the FIGURE 1, where first warp handles if case and the second warp handles the else case. No divergence.But in FIGURE 2: Some threads handle the if other handle the else, hence there is a branch divergence. Here due to presence of one if else there is two way branch divergence.

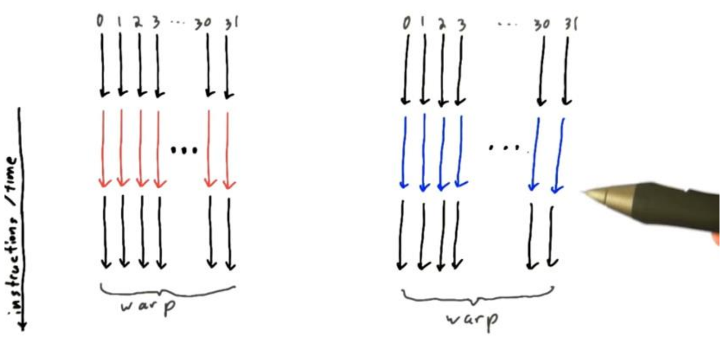


FIGURE 1

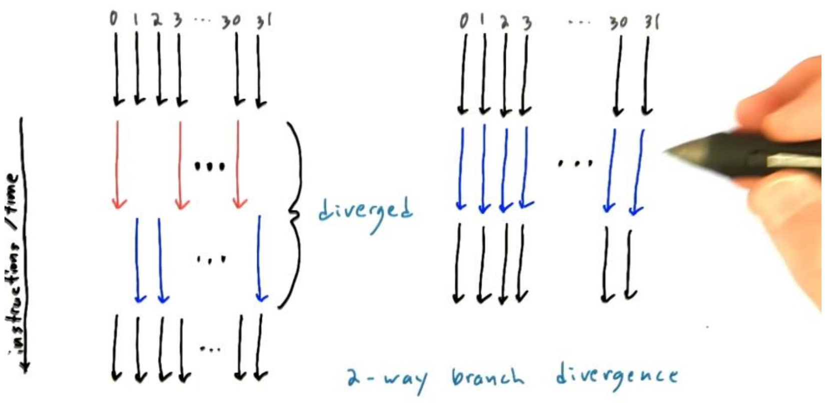


FIGURE 2

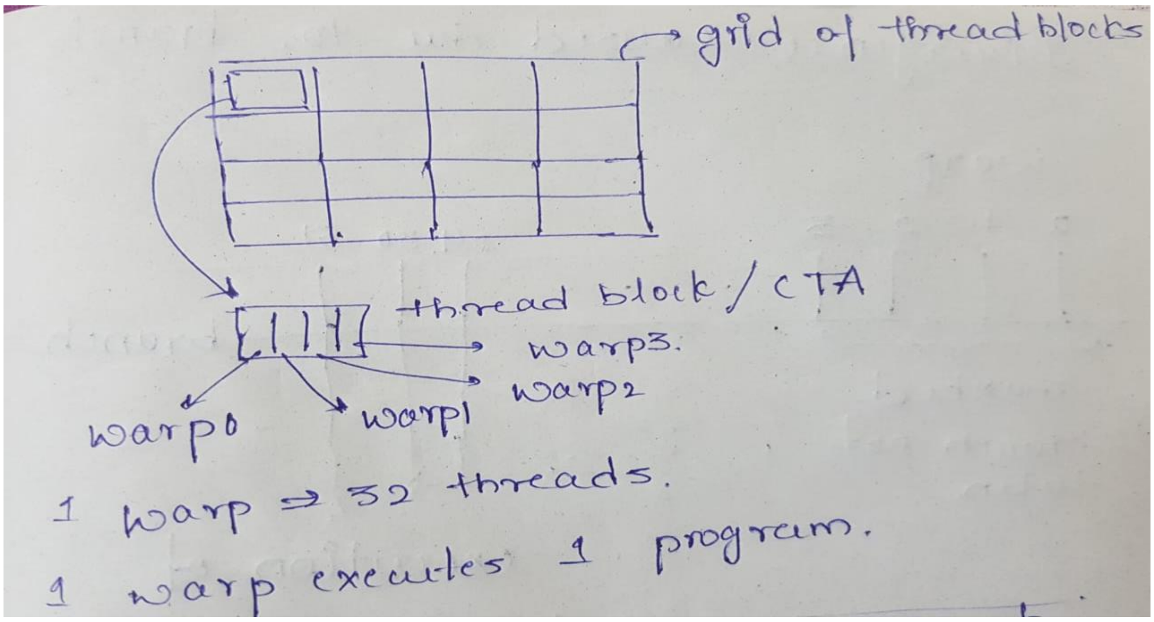
Big step and Small step Operational semantics:

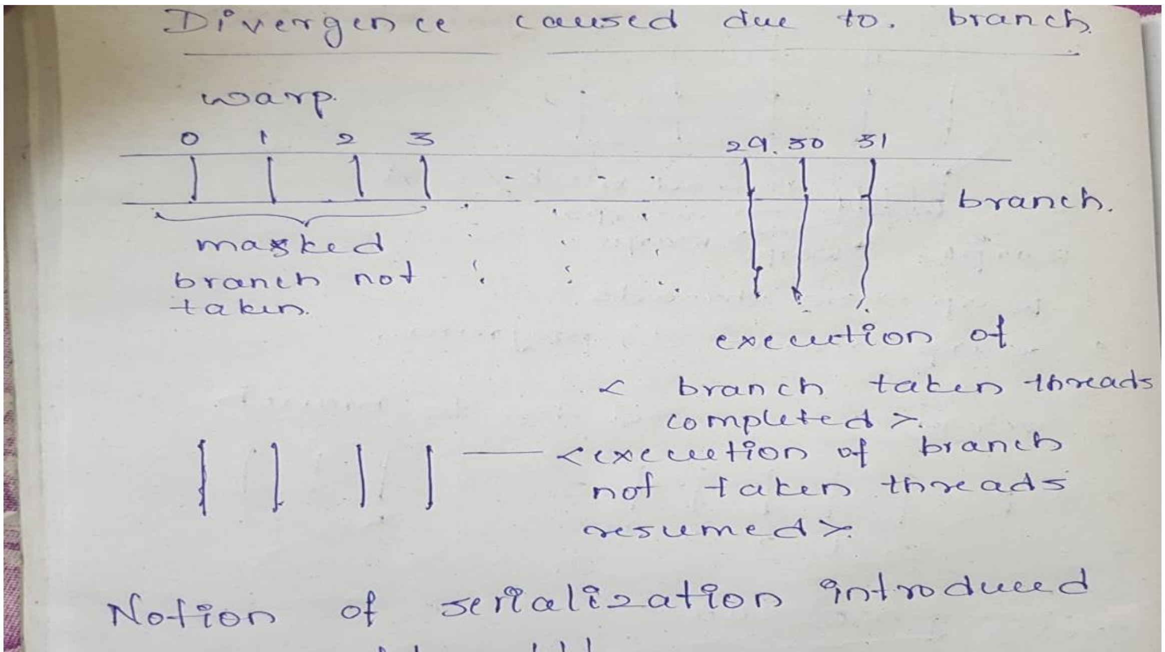
Operational semantics is a category of formal programming language semantics in which certain desired properties of a program, such as correctness, safety or security, are verified by constructing proofs from logical statements about its execution and procedures, rather than by attaching mathematical meanings to its terms (denotational semantics). Operational semantics are classified in two categories: **structural operational semantics** (or **small-step semantics**) formally describe how the *individual steps* of a computation take place in a computer-based system; by opposition **natural semantics** (or **big-step semantics**) describe how the *overall results.*

The Paper gives small step operational semantics for the formal language.

Brief introduction about warp specialization model in CUDA:

Cuda issues grids of thread blocks that can communicate within the grid, each block is made up of threads grouped into warps of 32 threads. The thread blocks are also known as Cooperative Thread Arrays. All the threads in a warp execute same program in parallel. Each warp executes on a single multiprocessor. When threads in a warp encounter a branch instruction, the SM executes the warp with all the threads that take the branch by masking the ones that don't followed by executing the warp by masking off the threads that took the branch. This essentially brings serialization into picture degrading the Performance of the GPU as one set of the threads in the warp take one control path and other set takes other path after the first set has finished executing. The warp specialization requires a powerful synchronization mechanism to work efficiently and correctly. One of the options is to use named barriers. Two barriers, one non-blocking known as *arrive* barrier and other blocking known as *sync* barrier can be used.




Divergence caused due to. branch
warp.
0 1 2 3
29. 30 31
branch.
masked
branch not taken.
execution of
< branch taken threads completed >.
<execution of branch not taken threads resumed>.
Notion of serialization introduced
→ problem !!!

Power Point Presentation for the topic can be found at:

https://github.com/Akanksha-Tonne/IOS_2019_Assignment/tree/master/PPT

This word document along with weekly summaries can be found at:

https://github.com/Akanksha-Tonne/IOS_2019_Assignment

REFERENCES

[1] M. Bauer, H. Cook, and B. Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. SC '11, 2011.

[2] M.Bauer,S.Treichler,andA.Aiken. Singe:Leveragingwarpspecialization for high performance on GPUs. PPoPP '14, 2014.

Thread Divergence:

https://www.youtube.com/watch?v=oylKKetzucg&t=1s

Big Step and small step operational semantics:

https://en.wikipedia.org/wiki/Operational_semantics

Online Slides:

https://stanford-ppl.github.io/website/papers/sc11-bauer-slides.pdf

https://cs.stanford.edu/~sjt/pubs/ppopp14.pdf