Operational semantics:

Here we are making an symmetrical assumption that all CTA's execute the same program and therefore we only need to model and verify a single CTA to establish correctness,also the validation is limited to the accesses to a GPU's shared memory(which has a well defined semantics due to the software-managed nature of the cache**).Global memory is not used for communication in any of the warp-specialized kernels of which already we are familiar with,and also weak semantics.So the data is passed between threads through global memory.**

Now our aim is to formally check the deadlock freedom,proper named barrier recycling and shared memory data race freedom.

[mentioned these definations already in my last summary

deadlock freedom: making sure that use of named barriers does not result in deadlocks

named barrier recycling : making sure that Ids of named barriers are properly re-used

shared memory data race freedom:making sure that shared memory accesses synchronized by named barriers are race free.

]

Inter-CTA synchronization: synchronization among threads can be done implicitely by launching kernel.Use the kernel launch itself. Before the kernel launch, or after it completes, all blocks (in the launched kernel) are synchronized to a known state.

Uber-kernels: Just a kernel version.

To understand Intra and Inter CTA synchronization
this link is helpful: https://devblogs.nvidia.com/cooperative-groups/
It is just these synchronization patterns both within and across CUDA thread blocks.

Atomic primitives: atomic primitiveness is the set of read-modify-write (RMW) operations where the value in some memory location is read, modified in some way, and then written back, with the guarantee that no other write will occur to that location between the read and the write.

Syntax: Now we are diving into our domain that is our formal language that is syntax.Here we need to make few assumptions about grammars.A thread program is a sequence of commands.
Thread->P
CTA->T
CTA has N threads that can synchronize for access to shared memory
Variables in shared memory locations->g
i--> to range over thread identifiers
B-->Named Barriers which is provided by hardware.
b-->specific barrier name or ID(first argument in the code)
n-->expected number of threads to register at the perticular generation of barrier(second argument in the code).
We denote a thread by P and a CTA by T. We use P1 ||P2 || . . . || PN to denote a CTA with threads P1, P2, . . . , PN .

assume that all variables occupy 64 bits. We consider abstract thread programs for each thread. Each thread program has a separate thread identifier denoted by id.

P ::= return | c; P //thread i terminated using return.
c ::= read g | write g // c is command in which a thread can either read/write(treated as no-ops)from shared memory location,in other words perform synchronization operation.Used for detecting data races and they play no role in the semantics of named barriers.

| arrive b n | sync b n //sync is for blocking synchronization operations and arrive is for non-blocking synchronization operations.

// the standard barrier syncthread is expressed as sync 0 N: a sync across all threads in a CTA on barrier 0. Program points are defined in the standard manner: before and after each command c. Each program point is uniquely identified by the command just before it.
//We say that a program point η and command cη correspond if η is the program point just after cη. The first program point of thread i is denoted by η i I and the last by η i F .
// Warp-synchronous execution is the assumption that all threads within a warp will execute in lock-step. This assumption has traditionally held for

past GPU architectures, but it is not standardized and may be invalidated by future designs.
//modelling warp-synchronous execution in our language by inserting a sync command across all threads in a warp after every original command in a program, thereby ensuring that the threads within a warp execute in lockstep.

State:
s-->state of a CTA.

//An enabled map E that maps thread identifiers to booleans signifying whether the thread is enabled or not. Threads are disabled when they block on a barrier.
// A barrier map B that maps 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.

-->Thread count is configured by the first thread that reaches a barrier.
--->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, A[x/y] --> the map that agrees with A on all inputs except y and maps y to x.
-->A[x/Y] -->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 $\forall i.E(i) = \text{true}$ and $\forall b.B(b) = ([], [], \perp)$.
-->making sure everyting whether or not
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.

Semantics:
Let us assume that T(CTA's small step operational semantic) with threads P1,P2,.....P_N executing in parallel.We know the state s has two components E and B.There are two kind of rules.

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

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

$$\frac{\forall b.\Big(\mathcal{B}(b) = ([],[],\bot) \vee \big(\mathcal{B}(b) = (\mathcal{I},\mathcal{A},n) \wedge |\mathcal{I}| + |\mathcal{A}| < n\big)\Big) \quad \mathcal{E},\mathcal{B},i,P_i \rightsquigarrow \mathcal{E}',\mathcal{B}',i,P_i'}{\mathcal{E},\mathcal{B},P_1||\ldots||P_i||\ldots||P_N \rightsquigarrow \mathcal{E}',\mathcal{B}',P_1||\ldots||P_i'||\ldots||P_N}$$

A CTA/Thread program executing in some state takes a single step.
Which means it is resulting in a new CTA/Thread.
So in this above step, explaination goes like this..
for all b that is for all barriers,either b is unconfigured or will have to register more threads.
Non-deterministic--->even for the same inputs show different outputs or varying behaviours on different runs.
So if we run N thread programs executing concurrently,any one of these can make an evaluation step from Pi to Pi' with some side effects on the state because of it's Non-deterministic behaviour.

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

If the correct number of threads(n) have registered by barrier at b,then enabling or waking up all the threads blocked at the barrier ,recycling here means changing b to an unconfigured state,and update the control of all threads in initial state I(This rule recycles a barrier by returning it to an unconfigured state).

//This terminates the threads when they execute return.

$$\frac{}{\mathcal{E},\mathcal{B},\mathbf{return} \parallel \ldots \parallel \mathbf{return} \rightsquigarrow \mathcal{E},\mathcal{B},done}$$

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

This is the first synchronization operation that configures the barrier and find out the number of threads required.

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

Programs with more than maximum threads count permitted in CTA's can be rejected by a preprocessing phase before execution.

sync's and arrive's semantics are similar,If the sync is first thread to register with a barrier.Map is enabled and updated by thread,and is added to the list of blocked threads.

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

On executing the Non-blocking arrive,control and map are updated.

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

Here,the operation is slightly different from arrive:the thread is disabled and gets added to the list of blocked threads.After this,control remains unchanged.

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

The expected number of prticipants must match for each thread program. for each generation of a named barrier.

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

Denominator is transitioning to the error state,if there are invalid number of threads that is trying to reach barrier.

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

If there is a **mismatch** on thread count then it goes to the error state.

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

Whenever there is arrive synchronization command,error productions are identical.

Finally,if any thread produces an error then the execution of CTA ends up in error stae.
No other outcome is possible for error productions except reaches done,error and deadlock states.