**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. [6]

**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).[6]

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.[7]

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 [7]

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.[7]**

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.[7]

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.[7]

Summary:

After understanding what the GPU’s and wrap specialised kernels are all about, the Introductory part of the paper is summarised as follows:

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](https://en.wikipedia.org/wiki/Semantics_(computer_science)) in which certain desired properties of a program, such as correctness, safety or security, are [verified](https://en.wikipedia.org/wiki/Formal_verification) by constructing proofs from logical statements about its execution and procedures, rather than by attaching mathematical meanings to its terms ([denotational semantics](https://en.wikipedia.org/wiki/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.