Static Analysis based Bug Detection for CUDA Kernels

Introduction to Program Analysis and Optimization

TA: Soumik Kumar Basu (cs21resch11004)

Deadline: May 7, 2025

April 9, 2025

1 Background

Synchronization bugs are critical issues in GPU-accelerated applications that can lead to unpredictable and incorrect program behaviour. These bugs typically arise due to improper synchronization between threads, which are the fundamental execution units in CUDA kernel functions. The three primary types of synchronization bugs are data races, redundant barriers, and barrier divergence. Data races occur when multiple threads access shared memory simultaneously in an unsafe manner, leading to inconsistent results. Redundant barriers involve unnecessary synchronization points that degrade performance without preventing any concurrency issues. Barrier divergence happens when some threads in a block reach a synchronization point while others do not, causing undefined behaviour and inefficient execution. We will now discuss each type of these bugs in detail.

1.1 Data Race

The code in Figure 1 demonstrates an example of a data race bug in a CUDA kernel function. A data race occurs when two or more threads concurrently access the same memory location, and at least one thread performs a write operation, leading to undefined behaviour due to a lack of synchronization. Here, at Lines 4 and 6 the threads write to the shared memory array `f_val2reduce[tid]` based on the condition `(y > 0 && a < C)`. Subsequently, at Line 8, the function `get_block_min` tries to write the same array `f_val2reduce` on an array index other than index `tid`. While threads within a single warp operate in lock-step fashion, no implicit synchronization exists between different warps within a thread block. Consequently, certain threads may execute the write operation at Line 8 before others complete their writes at Line 4 or Line 6, resulting in a race condition. The appropriate solution requires the inclusion of a block-level synchronization barrier using the `__syncthreads` primitive immediately before Line 8. This ensures all threads within the block complete their initial write operations before any thread proceeds to the subsequent memory access, thereby maintaining data integrity and deterministic program behaviour.

```
1  tid = threadIdx.x ;
2  ....
3  if ( y > 0 && a < C )
4    f_val2reduce [ tid ] = f ;
5  else
6    f_val2reduce [ tid ] = INFINITY ;
7  // get_block_min will write data to f_val2reduce
8  get_block_min ( f_val2reduce, f_idx2reduce );
9  ....
```

Figure 1: Example of Data Race Bug

1.2 Redundant Barrier

The code segment in Figure 2 demonstrates a common synchronization inefficiency known as a redundant barrier bug in CUDA parallel programming. This inefficiency manifests when a barrier synchronization primitive

(__syncthreads) is placed at a position where thread synchronization is unnecessary, resulting in suboptimal performance.

```
1  const unsigned tid = threadIdx .x ;
2  s_median [ tid ] = FLT_MAX ;
3  s_idx [ tid ] = 0;
4  __syncthreads () ;  //redundant synchronization
5  if ( i < iterations ) {
6     ...
7     s_idx [ tid ] = I ;
8     s_median [ tid ] = m ;
9  }
10 ....
```

Figure 2: Example of Redundant Barrier Bug

In Figure 2, the barrier synchronization (__syncthreads) is positioned at Line 4, between the initialization of shared memory arrays (s_median[tid] and s_idx[tid]) and their subsequent modification within the conditional block. This synchronization point is redundant because the initialization operations are performed independently by each thread on its own designated memory location (tid), with no interdependencies requiring synchronization at this stage. Furthermore, the subsequent conditional block (Lines 6 - 8) potentially modifies the same thread-specific memory locations without creating any data hazards that would necessitate synchronization. The performance impact of this redundant barrier is significant, as it forces all threads within a block to wait unnecessarily before proceeding, thereby introducing execution latency without computational benefit. The correct implementation would eliminate this barrier or reposition it to locations where actual thread coordination is required, such as after shared memory updates that precede operations dependent on values written by other threads.

1.3 Barrier Divergence

The code in Figure 3 illustrates a critical synchronization defect known as a barrier divergence bug in CUDA parallel programming. This error occurs when a thread synchronization primitive (__syncthreads) is positioned within a conditionally executed code block, resulting in undefined behaviour and potential program failure.

```
1  if(tid>N){
2     s_dist [ sid ] = dist ;
3     s_idx [ sid ] = s_idx [ sid + I ];
4     __syncthreads () ;  //Remove the barrier from this location
5  }
6  // fix by moving the barrier out .
7  ...
8  }
9  ....
```

Figure 3: Example of Barrier Divergence Bug

In Figure 3, the __syncthreads directive is placed inside a conditional block at Line 4. This configuration creates a scenario where some threads within a block may execute the barrier while others bypass it, depending on their evaluation of the conditional expression. According to CUDA's execution model, all threads within a block must encounter the same __syncthreads calls; otherwise, threads that reach the barrier will wait indefinitely for threads that never arrive, resulting in deadlock.

The code comment at Line 6 correctly identifies the appropriate solution: relocating the synchronization barrier outside the conditional scope (as shown in the "fix by moving the barrier out" implementation). This modification ensures that all threads within the block, regardless of their conditional execution paths, will encounter the barrier synchronization point, thereby maintaining the fundamental invariant of CUDA's synchronization mechanism. This restructuring preserves the intended thread coordination while eliminating the deadlock condition, resulting in correct and deterministic parallel execution.

Related Work. For a more detailed understanding of these bugs and a comprehensive runtime solution that automatically addresses them, refer to the paper by Wu et al. [1].

2 Project Task

1. **Design of the Static Analysis Tool**: Design a static analysis tool capable of detecting synchronization bugs (mentioned in the previous section) in CUDA kernel functions. The tool should analyze

kernel code to identify potential synchronization issues without executing the program. Clearly describe the methodology by which your tool will identify synchronization bugs statically. We understand the time constraints, and we do not expect a fully functional tool. However, we expect a well-structured, logically sound, and error-free design that demonstrates a clear approach to detecting synchronization bugs.

2. **False Positives and False Negatives**: Address the conditions under which the tool may produce false positives and explain the rationale behind it. Your analysis should aim to be conservative—while false positives are acceptable, false negatives should be avoided. Any known instances where false positives or false negatives may occur should be clearly documented, with examples provided in the report.

3. **Documentation and Report:** Write a comprehensive and well-organized report detailing the design (see Pt. 1), implementation and functionality of your tool. The quality of your report (along with viva) will be the primary basis for evaluating your work, so ensure it is clear, precise, and well-documented for review by the teaching assistants.

References

[1] Wu, M., Ouyang, Y., Zhou, H., Zhang, L., Liu, C., and Zhang, Y. Simulee: detecting cuda synchronization bugs via memory-access modeling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (New York, NY, USA, 2020), ICSE '20, Association for Computing Machinery, p. 937–948.