Verifying GPU Kernels in Dafny

Arjun Vedantham
Alex Lee
ECE584 - Fall 2025

Background

- GPUs present a new programming paradigm
- Focus: memory access violations
  - Nvidia GPU memory is laid out in 3D
    - *Blocks* of *threads*, each with some amount of reserved memory
  - The actual indexing happens in 1D
    - Extremely liable to mistakes
- Many other potential bugs
  - E.g. atomicity violations, data races, etc.
- Can *we gain a degree of certainty that GPU code generated by LLMs is correct?*

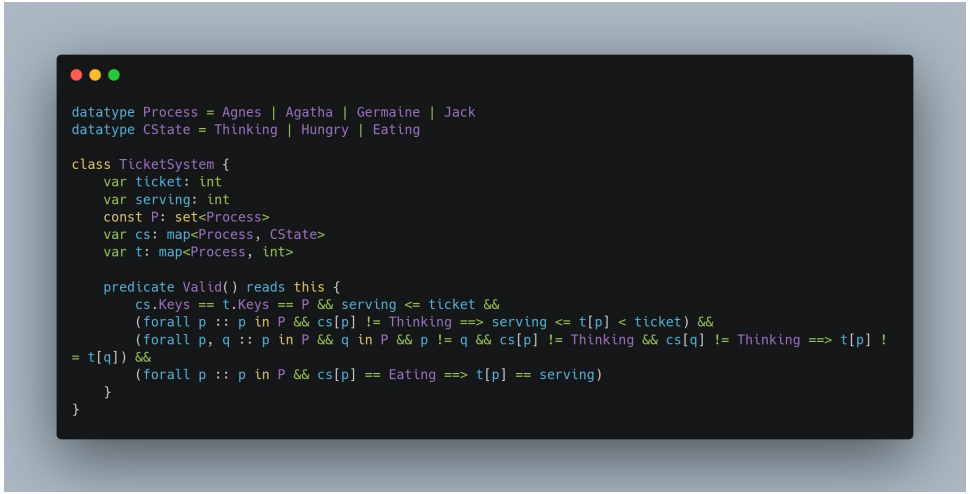
Background

Dafny

- We can write down *verifiable* programs
  - State invariants, leverage an SMT solver to discharge the proofs
- Contrast with Lean, Rocq:
  - Dafny relies on automated proving (via SMT)
- Allows us to do static analysis of GPU kernels
  - Contrasts with existing verification frameworks: compute-sanitizer (memcheck, initcheck)

Background

- Dafny is inherently sequential
  - No built in concurrency modeling
- Straightforward to model concurrent systems
  - First step: dining philosophers problem (similar to the Dijkstra token ring invariants)



```
datatype Process = Agnes | Agatha | Germaine | Jack
datatype CState = Thinking | Hungry | Eating

class TicketSystem {
    var ticket: int
    var serving: int
    const P: set<Process>
    var cs: map<Process, CState>
    var t: map<Process, int>

    predicate Valid() reads this {
        cs.Keys == t.Keys == P && serving <= ticket &&
        (forall p :: p in P && cs[p] != Thinking ==> serving <= t[p] < ticket) &&
        (forall p, q :: p in P && q in P && p != q && cs[p] != Thinking && cs[q] != Thinking ==> t[p] !
= t[q]) &&
        (forall p :: p in P && cs[p] == Eating ==> t[p] == serving)
    }
}
```

Related work

GPUVerify

- Focused on race detection / barrier divergence
- Restriction to canonical thread schedule
- If there is a data race in any execution (between barriers), one must occur in every execution (earliest race argument)
  - Restriction to two threads
- Modeling private vars / shared vars allows modeling two arbitrary threads
- Restriction on barrier semantics
- CUDA -> clang -> boogie

Problem statement

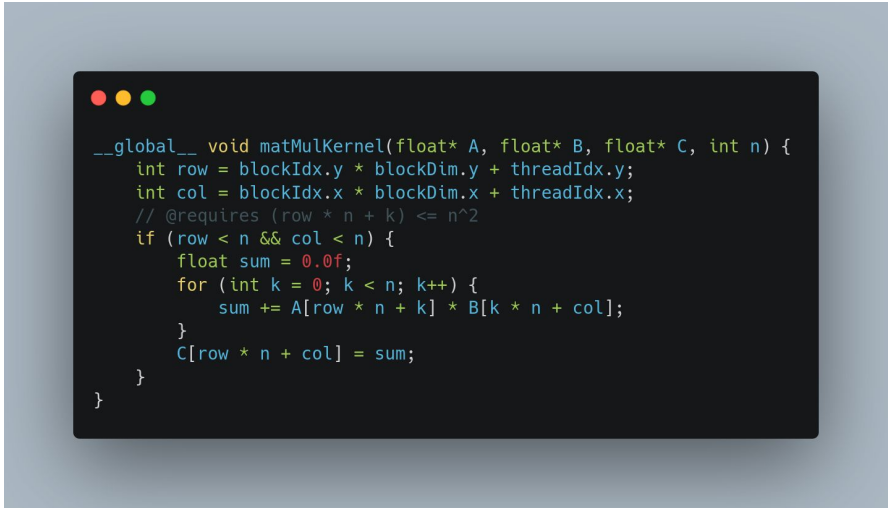
- Kernel K consisting of statements <s0…s_|K|>
  - Kernel execution formally consists of t threads in <s0_t…s_|K|_t> (may or may not be real)
  - Statements can be basic statements or a collection of statements
  - statements can describe control flow, operations
- Basic statement can describe local/global memory operations (eg, of type mem(arr,addr,val))
- Additionally threadId in CUDA is t, which is a local var defined for all threads

Problem statement

$\exists\ t : 0 <= t <= t_{max} \land \exists\ mem(addr) \in K_t$

$\exists\ mem(arr, addr, val)$

addr > |arr|



```
__global__ void matMulKernel(float* A, float* B, float* C, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    // @requires (row * n + k) <= n^2
    if (row < n && col < n) {
        float sum = 0.0f;
        for (int k = 0; k < n; k++) {
            sum += A[row * n + k] * B[k * n + col];
        }
        C[row * n + col] = sum;
    }
}
```

Evaluation

- Create proof of concept of extending Cuda -> Dafny manually to verify hard properties
  - Extract the dimensionality information of grid/blocks
  - Try to create a set of known memory addresses that are accessed
    - Prove the invariant holds

Evaluation

- Currently: matrix multiplication
  - Nvidia samples: benchmark source
  - Prompt GPT + try to reuse this proof infrastructure
- GPT is non-deterministic, so:
  - Try to insert deliberately bad statements (e.g. for *arr* of size *n,* attempt to access arr[2n])

Completed Work

- General model for blocks/threads organization
  - Define a validity predicate for a starting system
    - Must have blocks of the same dimension
    - Thread indices cannot exceed the block width
- Memory accesses: “bag of words” approach
  - Stripping the memory accesses of their program state context for now
- Invariant: every thread accesses memory addresses >= 0 and < block dimension
- Minor bits: index scrambling example

Early Takeaways

- Modeling concurrent systems is fairly straightforward (so far)
- Sample matrix multiplication kernel from GPT-4
- Issue is embedding Dafny programs with the kernels we want to verify
  - Right now: manual translation into a Dafny abstraction
  - Eventually: parse CUDA C version into Dafny version
    - SHARD: clang++ frontend to generate the invariants

Future Plans + Considerations

- Dafny/CUDA equivalence
  - Source level invariants make sense
  - Does the abstraction hold?
    - Currently have a good basic model, but there are some behaviors hidden from user's perspective
      - e.g. warp scheduling is not deterministic
- Additional bug classes
  - Atomicity/GPUVerify style data race detection