CUDA au Coq: A Framework for Machine-validating GPU Assembly Programs

Benjamin Ferrell
The University of Texas at Dallas
benjamin.ferrell@utdallas.edu

Jun Duan
The University of Texas at Dallas
jun.duan@utdallas.edu

Kevin W. Hamlen
The University of Texas at Dallas
hamlen@utdallas.edu

***Abstract*—A prototype framework for formal, machine-checked validation of GPU pseudo-assembly code algorithms using the Coq proof assistant is presented and discussed. The framework is the first to afford GPU programmers a reliable means of formally machine-validating high-assurance GPU computations without trusting any specific source-to-assembly compilation toolchain. A formal operational semantics for the PTX pseudo-assembly language is expressed as inductive, dependent Coq types, facilitating development of proofs and proof tactics that refer directly to the compiled PTX object code. Challenges modeling PTX's complex and highly parallelized computation model in Coq, with sufficient clarity and generality to tractably prove useful properties of realistic GPU programs, are discussed. Examples demonstrate how the prototype can already be used to validate some realistic programs.**

I. Introduction & Related Works

Machine-checked formal validation of software has become the gold standard for developing high-assurance algorithms and implementations. With the rise of program-proof co-development environments, particularly the Coq proof assistant [1], machine-validation of larger software systems has become both more feasible and more reliable. For instance, in recent years Coq has been used to verify correctness of a C compiler [2], prove update consistency of software-defined networks [3], model significant subsets of both the Intel x86 and the ARMv8 architectures [4], [5], validate a machine language certifier for Google's Native Client architecture [6], and develop the F* language for secure distributed programming [7], among many other successful applications.

The rise of GPU architectures as general-purpose computing platforms has made it increasingly desirable to make such validation approaches available for GPU algorithms and their implementations, toward developing high-assurance, GPU-based software. Many GPGPU programming tasks can benefit from such assurance; for example, GPUs are already being leveraged to more efficiently realize cryptography [8], scan for viruses [9], spot network intrusions [10] and so on. Unforeseen flaws in these computations could have severe ramifications for users and the general public. Since highly parallelized architectures are notoriously difficult for humans to reason about, it is especially important to develop machine-validation approaches for analyzing the correctness of such software.

Motivated by this need, we have developed the first encoding of a GPU pseudo-assembly language operational semantics in Coq, and used it to machine-validate some correctness properties of small GPU programs. Our prototype framework targets Nvidia's CUDA platform (though we anticipate that our general approach is extensible to other GPU architectures). In particular, our semantics model a Single Instruction, Multiple Threads (SIMT) architecture with warps, thread blocks, and grids. Our formalization hence supports all major standard forms of CUDA parallelism.

In order to minimize the *trusted computing base* (TCB) of our validation framework, and to offer developers maximum flexibility, theorems and proofs in our framework operate at the level of Parallel Thread eXecution (PTX) pseudo-assembly language computations rather than source code programs. This frees software developers to implement their high-assurance algorithms in any available source language, and use any compilation tools, provided that the tools ultimately yield PTX programs as output.

The downside of this choice, of course, is that proofs must reason at a significantly lower level than source code, and can therefore require extra effort to formulate. We believe this is nevertheless a wise design decision in the long term, because foundational support for PTX can eventually be scaled up to higher levels of abstraction via future development of Coq theories and tactics that modularize and automate much of the proof work for common source idioms. Similar trade-offs have motivated prior work on modeling ISAs of other low-level architectures in Coq (e.g., [4], [5], [11]).

As an example of this sort of scaling, a major success of our work is the formulation and validation of the first mechanized proof that CUDA's memory synchronization model ensures transparency of the thread scheduler. In particular, correctness of a computation under the assumption of a deterministic scheduler always implies correctness under a non-deterministic scheduler. This result greatly simplifies proofs of PTX code correctness by eliminating and abstracting away the non-deterministic scheduler from the sources of parallelism that proofs must consider.

Our work complements related works on GPU software debugging through unit testing [12], [13] or heuristic static analysis of source code [14] by offering a higher level of assurance than these traditional approaches can provide, but at the cost of (possibly significant) extra validation effort. That is, we anticipate that a typical development workflow for high-assurance GPU software should first employ these heuristic debugging techniques to identify and fix any demonstrable

w	$: \mathbb{N}$	(data types)
dty	$: \{\text{UI, SI, BD}\} \times \mathbb{N}$	
id	$: \{\text{Id}\} \times \mathbb{N}$	
bid	$: \mathbb{N} \times \mathbb{N} \times \mathbb{N}$	(memory)
ss	$: \{\text{Global, Const, Shared}\} \times bid$	
$addr$	$: ss \times \mathbb{N}$	
μ	$: (ss \times addr) \rightarrow (byte \times \mathbb{B})$	
reg	$: \{\text{UI}, \text{SI}\} \times \mathbb{N} \times \mathbb{N}$	(registers)
ρ	$: reg \rightarrow \mathbb{Z}$	
φ	$: \mathbb{N} \rightarrow \mathbb{B}$	(predicate state)
dim	$: \{\text{Dx, Dy, Dz}\}$	(special registers)
$sreg$	$: \{\text{T, B, NT, NB}\} \times dim$	
$sreg\_aux$	$: tid \rightarrow sreg \rightarrow \mathbb{N} := get\_sreg(tid)$	
op	$: reg \uplus sreg \uplus \mathbb{Z} \uplus reg \times \mathbb{Z}$	(operands)
θ	$: \mathbb{N} \times \rho \times \varphi$	(threads)
β	$: \vec{\omega}$	(blocks)

Table I
DEFINITION OF THE FORMAL PTX MODEL

flaws, and then proceed to apply formal methods machine-validation to obtain complete proofs of correctness. Our work is an alternative to runtime or hybrid fault detection approaches [12], [15], since it imposes no runtime overhead (all validation is strictly static), and yields *a priori* guarantees that span the universe of all execution traces for verified code.

It is beneficial for our model to be as small as possible in order to minimize the trusted computing base. Our prototype implementation in Coq part includes 350 SLOC for the PTX model, 300 SLOC for theorems, and 140 SLOC of Ltacs. Example theorems and proofs demonstrate how our prototype can be applied successfully to realistic GPU programs.

II. BACKGROUND

Our semantics are based on PTX, which is an intermediate assembly language either produced from compiling CUDA C/C++ code or manual encoding. It is the lowest intermediate representation that abstracts away from GPU-specific details that cannot be relied upon for forward compatibility.

At run-time, the intermediate assembly language is further compiled by the device driver down to the appropriate machine code, which is then executed on the device. Since our focus is on PTX, we are not concerned with the CUDA to PTX compilation process, but we do trust the final PTX to machine code compilation process. PTX is well documented so our goal is to formalize its execution semantics in Coq in a clean, succinct fashion conducive to writing tractable proofs.

At a high level, CUDA GPUs systematically execute millions of threads to accomplish a task. The number of threads spawned is defined by two user-configurable parameters, `grid_size` and `block_size`, which are 3-dimensional vectors. When a job is dispatched to the GPU, a `grid_size` array of *thread blocks* are spawned with each thread block containing a `block_size` array of threads.

Thread blocks are scheduled to execute on various *streaming multiprocessors* (SM). A GPU typically contains multiple SMs, which are composed of CUDA cores (ALUs), Load/Store units, and special function units. When a block is scheduled on an SM, threads are grouped into *warps* (sets of 32 threads) and scheduled for execution. Each time a warp executes an instruction, all threads do so in lock-step.

III. TECHNICAL APPROACH

Table I summarizes our formal PTX model. It includes data types dty, registers reg, special registers $sreg$, operands op, memory μ, threads θ, warps ω, blocks β, and grids.

1) Data Types: Data types are implemented in Coq as a sum type consisting of Unsigned Integers (**UI**), Signed Integers (**SI**), and Byte Data (**BD**), each parameterized by a bit width w. An id is a label to uniquely mark a storing unit or differentiate operational modules in the system.

2) Memory: GPU memory is made up of different sections, or *state-spaces* (ss), each of which have a specific purpose. Due to its accessibility for threads and some restrictions, some memory can be manipulated by only one specific thread, some can be shared by the threads inside the same block, some can be only readable, and the others have well-defined features for other purposes. We here focus on three types of state-spaces for memory: **Global**, **Const**, and **Shared**. To match the right type with an address, we define each in terms of its location and the accessibility type.

Many memory transactions are performed during execution without explicit order, possibly introducing memory synchronization errors. We therefore define memory μ to be a mapping from state-space and address to byte and boolean, where the boolean value specifies whether a byte is valid or could possibly still be in flight—similar to a valid bit for cache memory.

At the launch of a program, only **Global** and **Const** memory may have data, and their valid bits are set to `true`. During execution, global memory periodically updates as values are stored. Its valid bits are always `false`, since the hardware does not guarantee memory synchronization (excepting atomic instructions).

GPUs leverage data-independence to achieve high parallelism, so proper **Global** memory synchronization is often a prerequisite for code correctness. This is a perennial source of GPU algorithm bugs, so our memory formalization is designed to support formal verification of such properties.

Thread intercommunication is possible only in thread blocks with the use of **Shared** memory, which can be synchronized by barriers. A thread block performs some computation and threads begin to arrive at a barrier and wait. Once all threads have reached the barrier, the values stored in **Shared** memory during this time are guaranteed to be valid when execution resumes. To model this, our semantics initially set a value's valid bit to `false` and switch to `true` when the entire block has reached the barrier.

3) Registers: In this work we only consider unsigned (**UI**) and signed integers (**SI**), which can be extended to include all other data types of PTX codes. Threads are allocated a private register file which contains a set of PTX registers. A *register* (reg) holds temporary values during execution, and is uniquely identified by its data type, bit width, and index. We define the *register file* ρ to be a mapping from registers to integers.

During program execution, not all SIMT threads follow the same path. Threads maintain a set of predicate registers, which optionally prefix any instruction and indicate whether it should be executed or skipped. The case where the predicate is false is semantically equivalent to a `Nop`. In our implementation, we only consider branch instructions to optionally have prefixed predicates, so we introduce a pseudo-instruction to distinguish these from non-predicated branches. Coupled with the register file is a *predicate state* φ, which maps predicate indexes to boolean values.

4) Special Registers: *Special registers* contain static information about the grid configuration and a thread's location (i.e., index). They are unique to GPUs, and are useful when delegating work to each thread. There are four predominant special registers, each a 3-dimensional vector: the thread-index **T**, block-index **B**, block-size **NT**, and grid-size **NB**. Every thread has a unique combination of thread-index and block-index, but identical block-size and grid-size. To model this, our state model includes a *auxiliary function* for special registers (see Table I).

5) Operands: Operands are parameters to instructions and their types are statically known. We define 4 different types based on the origins of different state spaces: register, special register, immediate, and register & immediate.

6) Instructions: Instructions (`instr`) are drawn directly from the PTX language specification, with a Coq definition that enforces proper types of all parameters. A program running in CUDA can therefore be decompiled into PTX code and translated into our definition with no semantic gap. Programs `prg` are lists of PTX instructions and operand types.

7) Threads: Potentially millions of threads execute on a GPU, so we assign each an enumerated value for unique identification. Proofs do not exhaustively enumerate this potentially large identifier space; the identifiers typically take the form of universally quantified variables in proofs. The identifiers are passed to the auxiliary function in §III-4 to obtain the appropriate special registers. As mentioned in §III-3, threads maintain a set of private registers ρ and predicates φ. Therefore, we define a *thread* θ to be a tuple of these three components.

In the following semantics, vector $\vec{t}$ defines a thread's state, including its ID tid, memory state ρ, and predicate state φ.

8) Warps: A warp is defined as a set of 32 threads, and is the smallest level of granularity to execute on an SM. All threads in a common warp execute the same instruction at the same time (i.e., in lock-step), which is efficient for most instructions except for a predicated branch. With a predicated branch, there is the potential for part of the warp to take the branch and the rest stay behind to execute the next immediate instruction. A warp in this state is called *divergent*, and must now execute the two (or more) paths serially, thereby increasing run-time. At the formal level, we define a warp to be in one of two states: uniform execution of a set of threads (`Uni` (`pc` : `nat`) (`ts` : `list thread`)), or divergent execution of two warps (`Div` (`w1 w2` : `warp`)). Hence, warps may form a tree of divergences.

$$\frac{}{\texttt{Nop} \vdash \langle (pc, \vec{t}), \mu \rangle \rightarrow_1 \langle (pc+1, \vec{t}), \mu \rangle}\,(\text{nop})$$

$$\frac{\vec{t'} = \{(tid, \rho[r \mapsto op(a,b)], \varphi) \,|\, (tid,\rho,\varphi) \in \vec{t}\}}{\texttt{Bop}\ op\ r\ a\ b \vdash \langle (pc, \vec{t}), \mu \rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu \rangle}\,(\text{bop})$$

$$\frac{\vec{t'} = \{(tid, \rho[r \mapsto op(a,b,c)], \varphi) \,|\, (tid,\rho,\varphi) \in \vec{t}\}}{\texttt{Top}\ op\ r\ a\ b\ c \vdash \langle (pc, \vec{t}), \mu \rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu \rangle}\,(\text{top})$$

$$\frac{\vec{t'} = \{(tid, \rho[r \mapsto a], \varphi) \,|\, (tid,\rho,\varphi) \in \vec{t}\}}{\texttt{Mov}\ r\ a \vdash \langle (pc, \vec{t}), \mu \rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu \rangle}\,(\text{mov})$$

$$\frac{\vec{t'} = \{(tid, \rho[\mathrm{r} \mapsto \mu(ss,a)], \varphi) \,|\, (tid,\rho,\varphi) \in \vec{t}\}}{\texttt{Ld}\ ss\ r\ a \vdash \langle (pc, \vec{t}), \mu \rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu \rangle}\,(\text{ld})$$

$$\frac{\vec{v} = \{(ss, a, \rho(r)) \,|\, (tid,\rho,\varphi) \in \vec{t}\} \quad \mu' = \text{update}(\mu, \vec{v})}{\texttt{St}\ ss\ a\ r \vdash \langle (pc, \vec{t}), \mu \rangle \rightarrow_1 \langle (pc+1, \vec{t}), \mu' \rangle}\,(\text{st})$$

$$\frac{}{\texttt{Bra}\ tgt \vdash \langle (pc, \vec{t}), \mu \rangle \rightarrow_1 \langle (tgt, \vec{t}), \mu \rangle}\,(\text{bra})$$

$$\frac{\vec{t'} = \{(tid, \rho, \varphi[p \mapsto cmp(a,b)]) \,|\, (tid,\rho,\varphi) \in \vec{t}\}}{\texttt{Setp}\ cmp\ p\ a\ b \vdash \langle (pc, \vec{t}), \mu \rangle \rightarrow_1 \langle (pc+1, \vec{t'}), \mu \rangle}\,(\text{setp})$$

$$\frac{\begin{array}{c}\vec{t_1} = \{(tid,\rho,\varphi) \,|\, (tid,\rho,\varphi) \in \vec{t} \wedge \varphi_i(p)\} \\ \vec{t_2} = \{(tid,\rho,\varphi) \,|\, (tid,\rho,\varphi) \in \vec{t} \wedge \neg\varphi_i(p)\} \\ \omega' = \text{sync}((pc+1, \vec{t_2}), (tgt, \vec{t_1}))\end{array}}{\texttt{PBra}\ p\ tgt \vdash \langle (pc, \vec{t}), \mu \rangle \rightarrow_1 \langle \omega', \mu \rangle}\,(\text{pbra})$$

$$\frac{i \neq \texttt{Sync} \qquad i \vdash \langle \omega_1, \mu \rangle \rightarrow_1 \langle \omega_1', \mu' \rangle}{i \vdash \langle (\omega_1, \omega_2), \mu \rangle \rightarrow_1 \langle (\omega_1', \omega_2), \mu' \rangle}\,(\text{div})$$

$$\frac{}{\texttt{Sync} \vdash \langle \omega, \mu \rangle \rightarrow_1 \langle \text{sync}(\omega), \mu \rangle}\,(\text{sync})$$

Figure 1. Warp small-step semantics

$$\text{sync}(\omega) = \begin{cases} (pc+1, \vec{t}), & \text{if } \omega = (pc, \vec{t}) \\ \text{sync}(\omega_2), & \text{if } \omega = ((pc_1, \{\}), \omega_2) \\ \text{sync}(\omega_1), & \text{if } \omega = (\omega_1, (pc_2, \{\})) \\ (pc_1 + 1, \vec{t_1} \cup \vec{t_2}), & \text{if } \omega = ((pc_1, \vec{t_1}), (pc_2, \vec{t_2})) \\ & \quad \wedge\ pc_1 = pc_2 \\ (\omega_2, (pc_1, \vec{t_1})), & \text{if } \omega = ((pc_1, \vec{t_1}), \omega_2) \\ (\text{sync}(\omega_1), \omega_2), & \text{otherwise } \omega = (\omega_1, \omega_2) \end{cases}$$

Figure 2. Warp sync function

Figure 1 lists the small-step semantic rules for warps. The semantics are distinguished by instruction input, which transforms the given warp according to the operational semantics of the instruction. The first nine rules are fairly straightforward and only apply to uniform warps. Instructions `Bop` and `Top` are arithmetic operations on two and three inputs, respectively. If a warp is divergent and the instruction is not `Sync`, then the left-most warp is executed.

The final case is the most complex due to the synchronization operation. When a warp diverges, it should at some point converge back to a uniform warp through a sync operation (see Figure 2). Compilers enforce this because memory barriers can cause undefined execution. In some executions, a warp could diverge with some threads halting at a barrier while the others continue to execute and eventually exit. Since all threads must be at the memory barrier in order for it to lift, this situation creates a deadlock and the program hangs or (more likely) crashes. Careful analysis is required to establish that correct code always avoids this situation. Our operational semantic encodings facilitate such reasoning in Coq proofs.

9) Blocks: Thread blocks are typically defined as sets of threads, but because they are grouped into warps, we formalize them as sets β of warps.

Warps are selected by the scheduler to execute an instruction, but the details of the scheduling can vary between GPUs and other contextual factors. Proofs in our framework must

$$\frac{\exists \omega \in \beta \,.\, \pi(\omega_{pc}) \notin \{\text{Bar}, \text{Exit}\} \qquad \pi(\omega_{pc}) \vdash \langle \omega, \mu \rangle \rightarrow_1 \langle \omega', \mu' \rangle}{\pi \vdash \langle \beta, \mu \rangle \rightarrow_1 \langle \beta[\omega'/\omega], \mu' \rangle}(\text{exec}_b)$$

$$\frac{\forall \omega \in \beta \,.\, \pi(\omega_{pc}) = \text{Bar}}{\pi \vdash \langle \beta, \mu \rangle \rightarrow_1 \langle \text{incr\_pc}(\beta), \text{commit}(\mu) \rangle}(\text{lift-bar})$$

$$\text{complete}(\pi, \beta) \equiv (\forall \omega \in \beta \;.\; \pi(\omega_{pc}) = \text{ Exit})$$

$$\frac{\exists \beta \in \gamma \,.\, \neg\text{complete}(\pi, \beta) \qquad \pi \vdash \langle \beta, \mu \rangle \rightarrow_1 \langle \beta', \mu' \rangle}{\pi \vdash \langle \gamma, \mu \rangle \rightarrow_1 \langle \gamma[\beta'/\beta], \mu' \rangle}(\text{exec}_g)$$

Figure 3. The small-step semantics of thread block and grid

therefore establish correctness independently of the scheduling algorithm. Our semantics hence formalize the scheduler non-deterministically.

Looking at the first two rules of Figure 3, we have two possible scenarios. In the first scenario, there exists a warp that is not at a memory barrier or has not finished, so it executes its next instruction. Notation $\beta[\omega'/\omega]$ denotes the capture-avoiding substitution of ω with ω', in order to update the block state. The second scenario applies when all warps have reached a memory barrier. At this time all **Shared** memory is committed (i.e., all valid bits are set to `true`), and warp program counters are incremented to continue execution.

10) Grids: Grid execution is similar to thread block execution in the sense that thread blocks are non-deterministically chosen for execution. There is no hardware based memory synchronization at this level, so Grid small-semantics (last two rules of Figure 3) includes only one derivation rule. The rule chooses an unfinished thread block to execute and updates the grid with the new thread block state.

We define a thread block to be complete when all warps are at an `Exit` instruction. When a grid finishes executing, all thread blocks should be complete.

IV. Validation Results & Correctness

To demonstrate how our framework facilitates machine-checked validation of GPU software, we here walk through the validation of PTX code that sums two vectors. Listing 1 is the verbatim PTX code compiled from sources; our only modification is to rename the parameters in lines 9–12 to something more human-readable, for explanatory purposes.

Listing 2 shows our translation of the PTX code to corresponding Coq definitions. At the beginning of all PTX functions is a declaration of the types and quantities of needed registers. We do not necessarily need to translate this to Coq, but for the sake of readability we do so in lines 1–4. Since PTX instructions take operands (not just registers) as input, we use a wrapper to turn all registers into operands (line 6), and prefix the variable with an underscore (e.g. _r1) to make the distinction. Lines 9–12 load the function arguments into registers. Loads have semantics equivalent to Moves in our framework, so the Coq translation uses `Mov` instructions.

All threads execute the same code concurrently, but each receives a distinct thread index, which is computed in lines 14–17. Since each thread is tasked with adding elements from a specific index in the vector, at least `size` threads are spawned, where `size` is the length of the vector. It could be the case that more than `size` threads are spawned, so a bounds check is implemented in lines 19–20. As mentioned in §III-8, warps can diverge when multiple execution paths exist. In this example there are two paths with a divergence point at line 20. The matching `Sync` instruction is inserted at line 35 (index 18 in the Coq instruction list).

The translation of lines 22–33 omits the `cvta.to` instructions because they are implicit in our PTX formalization. Specifically, instruction `cvta.to` converts a generic address to a specified state-space, but our framework implicitly does this with the `ld` and `st` instructions, which both require a state-space parameter. The final PTX instruction is `ret`, which we translate to `Exit` for this example to end the validation at function completion.

In Coq we can formally prove and machine-check theorems about the program's behavior on arbitrary inputs, such as termination and correctness of output. Even though this is a simple example, Coq has a rich collection of supporting libraries and theory modules that can be applied to reason about much more complex mathematical properties of larger programs. For example, there is extensive prior work on leveraging Coq to prove properties of cryptographic algorithms [16]–[18]. For expository simplicity, we here limit our presentation to proving total correctness of the code segment.

Listing 3 shows how termination is defined and proved for our program. A program's execution is considered terminated (or completed) when all threads have reached the `Exit` instruction. Definition `terminated` in the listing checks whether all blocks are complete, which entails confirming whether all warps are complete. We then initialize the state (`kc`, `g`, and `mu`) to begin program execution. Finally we define our theorem `add_vector_terminates` by hypothesizing that after 19 small-steps of execution, the program terminates.

Proposition `n_apply` is inductively defined in Listing 4, which relates the number n of applications of a proposition $f : A \rightarrow A \rightarrow \text{Prop}$ to an input $a : A$ with an output $a' : A$. At its base case, the number of steps is zero, yielding an identity proposition. The inductive case performs one application of f and recursively applies it $n - 1$ more times.

Proving a theorem in Coq typically begins with introducing universally quantified variables (g' and μ') and hypotheses (`Happ`) into the proof context, as shown. Next, the `repeat` tactic repeatedly applies a given proof tactic until it fails. The given tactic in this case is `unroll_apply`, which is defined in Listing 4.

Our `unroll_apply` tactic can be thought of as a primitive symbolic execution engine for PTX. It directly applies the operational semantics of PTX to a proof hypothesis through `inversion` reasoning, which infers a derivation rule's prerequisites from its consequent. In the case of derivation rules that encode operational semantics, this infers the set of new program states that may result from each instruction's execution, thereby symbolically interpreting the instruction within the proof environment.

Listing 1
VECTOR SUM PTX ASSEMBLY

```
.reg .pred  %p<2>;
.reg .u32  %r<9>;
.reg .u64  %rd<11>;





ld.param.u64  %rd1, [arr_A];
ld.param.u64  %rd2, [arr_B];
ld.param.u64  %rd3, [arr_C];
ld.param.u32  %r2, [size];

mov.u32  %r3, %ntid.x;
mov.u32  %r4, %ctaid.x;
mov.u32  %r5, %tid.x;
mad.lo.s32  %r1,%r4,%r3,%r5;

setp.ge.s32 %p1, %r1, %r2;
@%p1 bra  BB0_2;

cvta.to.global.u64  %rd4, %rd1;
mul.wide.s32  %rd5, %r1, 4;
add.s64  %rd6, %rd4, %rd5;
cvta.to.global.u64  %rd7, %rd2;
add.s64  %rd8, %rd7, %rd5;
ld.global.u32  %r6, [%rd8];
ld.global.u32  %r7, [%rd6];

add.s32  %r8, %r6, %r7;
cvta.to.global.u64  %rd9, %rd3;
add.s64  %rd10, %rd9, %rd5;
st.global.u32  [%rd10], %r8;


BB0_2: ret;
```

Listing 2
COQ PTX ASSEMBLY

```
Definition r1 : reg := (UI 32, 1).
Definition r2 : reg := (UI 32, 2).
...
Definition rd1 : reg := (UI 64, 1).
...
Definition _r1 : op := Reg r1.
...
Definition add_vector : prg := [
Mov rd1 arr_A;
Mov rd2 arr_B;
Mov rd3 arr_C;
Mov r2 size;

Mov r3 ntid_x;
Mov r4 ctaid_x;
Mov r5 tid_x;
Top MADLO r1 _r4 _r3 _r5;

Setp GE p1 _r1 _r2;
PBra p1 18;


Bop MULWD rd5 _r1 (Imm 4);
Bop ADD rd6 _rd1 _rd5;

Bop ADD rd8 _rd2 _rd5;
Ld Global r6 _rd8;
Ld Global r7 _rd6;

Bop ADD r8 _r6 _r7;

Bop ADD rd10 _rd3 _rd5;
St Global _rd10 r8;

Sync;
Exit ].
```

Listing 3
PROVING TERMINATION OF VECTOR SUM

```
Definition warp_complete (pi : prg) (w : warp) : bool :=
  match pi (get_pc w) with
  | Some Exit => true
  | _ => false
end.

Definition block_complete (pi : prg) (b : block) : bool :=
  forallb (warp_complete pi) b.

Definition terminated (pi : prg) (g : grid) : Prop :=
  forallb (block_complete pi) g = true.

(* sample parameter configs *)
Definition kc : kconf := ((1,1,1),(32,1,1)).
Definition g : grid := generate_grid kc.
Definition mu : mem_f := ... (* initial memory state *)

Theorem add_vector_terminates :
    ∀ (g' : grid) (mu' : mem_f),
    n_apply 19 (grid_t add_vector kc) (g,mu) (g',mu') →
    terminated add_vector g'.
Proof.
  intros g' mu' Happ.
  repeat (unroll_apply Happ).
  compute. reflexivity.
Qed.
```

Our symbolic interpreter ultimately reveals the final states of g' and μ' and encodes them as hypotheses in the proof environment. The `compute` tactic reduces `terminated add_vector` g' to `true = true` and we finally apply `reflexivity` to complete the proof.

We then prove partial correctness of the vector sum—i.e., that the result is the sum of the two input vectors if it terminates. Coupled with Listing 3, this establishes total correctness of the computation. Vectors A and B come from the initial memory state μ and vector C is from the final memory state μ'. This therefore posits that $A + B = C$.

Both of these proofs demonstrate the power of our automation tactics to reliably facilitate formal reasoning about PTX operation. By expressing the symbolic interpreter as proof tactics, we afford users the ability to quickly and easily reduce computations to symbolic expressions within a Coq proof, and subsequently apply the full power of Coq's mathematical theories to reason about the resulting symbolic expressions. This power comes without any additions to the TCB of the framework, since the tactics merely automate the application of the operational semantics rules; they do not introduce new rules that must be checked for accuracy.

Listing 4
PROOF AUTOMATION TACTICS FOR SYMBOLIC EXECUTION OF PTX CODE

```
Inductive n_apply {A:Type}: nat → (A→A→Prop) → A → A → Prop :=
| AppZero (f: A→A→Prop) (a:A): n_apply 0 f a a
| AppNext (n:nat) (a a1 a':A) (f: A→A→Prop)
    (Hf: f a a1) (Happ: n_apply n f a1 a'): n_apply (S n) f a a'.

Ltac unroll_apply H :=
  match type of H with
  | n_apply ?n _ _ _ =>
    match n with
    | O => inversion H; subst; clear H
    | _ => let Hgrid := fresh "Hgrid" in
           let Happ := fresh "Happ" in
           inversion H as [ | ? ? ? ? ? Hgrid Happ];
           subst; clear H;
           step_grid Hgrid
    end
  | _ => fail
end.
```

Listing 5
NON-DETERMINISTIC MAP

```
Inductive nth_ri {A:Type}
: nat → list A → A → list A → Prop :=
| RI_O a t : nth_ri 0 (a::t) a t
| RI_S n t t' x a (Hn: nth_ri n t a t') :
    nth_ri (S n) (x::t) a (x::t').

Inductive nd_map {A B : Type}
: (A→B) → list A → list B → Prop :=
| NDNil (f : A → B) : nd_map f [] []
| NDCons (f : A → B) (l l1 : list A) (l2 l' : list B) (a : A) (n : nat)
    (Hl: nth_ri n l a l1) (Hmap: nd_map f l1 l2) (Hl': nth_ri n l' (f a) l2) :
    nd_map f l l'.
```

Non-deterministic Execution. One of the most difficult aspects of parallel and concurrent programs to reason about is the inter-thread order in which instructions are executed. Explicitly considering all possible executions within proofs is infeasible and unmanageable even for small programs. To ease this burden, we have successfully proved a general theorem showing that the result of a PTX computation is always independent of the order in which the threads of a warp execute. It therefore suffices to only consider a sequential thread execution order within proofs. We here outline this theorem and its proof.

Listing 5 first defines proposition `nth_ri`, which removes an element a at position n from a given list l, and returns a new list l'. We use this definition to create a non-deterministic map function `nd_map`, whose elements are processed in an arbitrary order. This captures all possible thread schedules for warps, which execute threads in lock-step but in an unspecified order. Some components on the SM, such as SFUs, are physically limited in quantity, making it impossible for all threads in a warp to execute on the same clock cycle. Therefore, we seek to prove that a non-deterministic execution is equivalent to a deterministic one.

Listing 6
Non-deterministic/Deterministic Equivalence

```
Theorem nd_map_eq :
  ∀ (A B : Type) (f : A→B) (l : list A) (l' : list B),
  nd_map f l l' ⟷ l' = map f l.
Proof.
  intros A B f l l'.
  split; intros H.
  (* nd_map → map *)
  induction H.
    (* Case: NDNil *) reflexivity.
    (* Case: NDCons *)
    inversion Hl'; subst; inversion Hl; subst; clear Hl Hl'.
      (* SCase: n = 0 *)
      reflexivity.
      (* SCase: n is in the middle somewhere *)
      inversion H3; subst; clear H3 H.
      simpl. apply list_hd_eq.
      generalize dependent t. induction Hn0; intros;
      inversion Hn; subst; clear Hn; auto.
      simpl. apply list_hd_eq. eapply IHHn0. exact Hn1.
  (* map → nd_map *)
  generalize dependent l'.
  induction l; intros; subst; simpl.
    (* Case l = [] *) apply NDNil.
    (* Case l = hd :: tl *)
    eapply NDCons; try apply RI_O.
    apply IHl. reflexivity.
Qed.
```

Listing 6 summarizes our equivalence proof. We induct on the definition of `nd_map`, which generates two cases: the base case `NDNil` and inductive case `NDCons`. The base case is straightforward, so we here focus on the inductive case. It divides into two sub-cases: an element is chosen from the head, or from within the tail. The bulk of the proof focuses on the second sub-case. It leverages dependent inductive reasoning [19], [20] to universally consider all thread orderings.

V. Conclusion

Our work shows promise in providing a feasible means to machine-verify GPU programs using a well-established, mature proof assistant—the Coq system. Coq's strong, dependent typing system facilitates a natural encoding of GPU assembly code operational semantics as inductive axioms, including modeling its traditionally challenging parallelism and thread scheduling properties. To streamline proofs, Coq's tactic language is leveraged to build a symbolic interpreter that automatically derives provably correct symbolic expressions for assembly code fragments within the proof environment.

Using the resulting framework, we construct the first machine-checked proof of CUDA thread scheduling transparency, as well as validating correctness properties of PTX programs. Our approach provides stronger guarantees and assurances than are possible with unit testing, since it does not rely upon comprehensiveness of test sets.

Acknowledgments

The research reported herein was supported in part by ONR award N00014-17-1-2995, NSF award #1513704, an NSF IUCRC award from Lockheed Martin, and a gift from the Eugene McDermott Foundation. Any opinions or recommendations expressed are those of the authors and not necessarily of the above supporters.

References

[1] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, ser. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[2] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond, "A formally-verified C compiler supporting floating-point arithmetic," in *Proceedings of the 21st IEEE International Symposium on Computer Arithmetic (ARITH)*, 2013, pp. 107–115.

[3] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012, pp. 323–334.

[4] A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand, "Coq: The world's best macro assembler?" in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2013, pp. 13–24.

[5] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the ARMv8 architecture, operationally: Concurrency and ISA," in *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

[6] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "RockSalt: Better, faster, stronger SFI for the x86," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 395–404.

[7] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2011, pp. 266–278.

[8] T. Yamanouchi, "AES encryption and decryption on the GPU," in *GPU Gems 3*, H. Nguyen, Ed. NVidia Corporation, 2007.

[9] E. Seamans and T. Alexander, "Fast virus signature matching on the GPU," in *GPU Gems 3*, H. Nguyen, Ed. NVidia Corporation, 2007.

[10] M. Alshawabkeh, B. Jang, and D. R. Kaeli, "Accelerating the local outlier factor algorithm on a GPU for intrusion detection systems," in *Proceedings of the 3rd Workshop on General Purpose Processing Using Graphics Processing Units (GPGPU)*, 2010, pp. 104–110.

[11] R. Atkey, "CoqJVM: An executable specification of the Java virtual machine using dependent types," in *Proceedings of the International Conference on Types for Proofs and Programs (TYPES)*, 2007, pp. 18–32.

[12] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "GRace: A low-overhead mechanism for detecting data races in GPU programs," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011, pp. 135–146.

[13] A. Holey, V. Mekkat, and A. Zhail, "HAccRG: Hardware-accelerated data race detection in GPUs," in *Proceedings of the 42nd International Conference on Parallel Processing (ICPP)*, 2013, pp. 60–69.

[14] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira, Jr., "Divergence analysis and optimizations," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 320–329.

[15] P. Li, C. Ding, X. Hu, and T. Soyata, "LDetector: A low overhead race detector for GPU programs," in *Proceedings of the 5th International Conference on Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2014.

[16] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin, "Probabilistic relational reasoning for differential privacy," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 35, no. 3, 2013.

[17] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin, "Computer-aided security proofs for the working cryptographer," in *Proceedings of the 31st Annual Conference on Advances in Cryptology (CRYPTO)*, 2011, pp. 71–90.

[18] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *Proceedings of the 4th International Conference on Principles of Security and Trust (POST)*, 2015, pp. 11–18.

[19] C. McBride, "Elimination with a motive," in *Proceedings of the International Conference on Types for Proofs and Programs (TYPES)*, 2002, pp. 197–216.

[20] C. Cornes and D. Terrasse, "Automating inversion of inductive predicates in Coq," in *Proceedings of the International Conference on Types for Proofs and Programs (TYPES)*, 1995, pp. 85–104.