

Kuiper: Correct and Efficient GPU Programming with Dependent Types and Separation Logic

GUIDO MARTÍNEZ, Microsoft Research, USA

BASTIAN KÖPCKE, TU Berlin, Germany

JONÁŠ FIALA, ETH Zurich, Switzerland

GABRIEL EBNER, Microsoft Research, USA

TAHINA RAMANANANDRO, Microsoft Research, USA

MICHEL STEUWER, TU Berlin, Germany

TYLER SORENSEN, Microsoft Research, USA

NIKHIL SWAMY, Microsoft Research, USA

We introduce KUIPER, a language for safe and verified efficient CPU/GPU programming embedded as an extensible library within the F^* dependently typed language. We rely on F^* 's support for dependent types and its associated Pulse concurrent separation logic to develop a program logic in which to prove CPU/GPU programs safe, data-race free, and functionally correct. Our model of the GPU includes several intricacies, including the memory hierarchy, kernel launches, and synchronization within a single comprehensive framework. To do so, we extend the Pulse program logic with a novel notion of *located resources* and a new connective to structure reasoning about massively parallel programs, and present new proof rules to lift the per-thread view of GPU kernels to an end-to-end correctness specification.

We have used KUIPER to program and prove correct a variety of GPU kernels, including full functional correctness proofs of an optimized matrix multiplication using two levels of block tiling and tensor cores. In doing so, we have developed a range of libraries to enable programs and proofs at a high-level of abstraction but without imposing any runtime overhead. These allow KUIPER programs to be polymorphic (over types, operations, memory layout, and more) and compile to efficient, specialized CUDA code, while enabling a novel form of verified auto-tuning. Our experimental evaluation confirms that KUIPER programs match the performance of their handwritten CUDA counterparts and are competitive with closed source, state-of-the-art kernels in cuBLAS.

1 Introduction

With the vast compute resources being spent training and serving large language models (LLMs) on GPUs, the incentives to develop optimized GPU programs are higher than ever. However, mainstream GPU programming languages, such as CUDA and HIP, are low-level and unsafe, relying on the programmer to do proper memory management, indexing, and synchronization between thousands of threads accessing shared resources in various parts of the GPU's memory hierarchy. The low-level aspect is appreciated by expert GPU programmers, as it allows for fine-grained control to extract maximum performance, particularly as small details can account for large performance differences. However, as the devil is also in the details, writing correct programs remains a challenge, and even experts spend a significant portion of their development effort debugging memory bugs, data races, and functional correctness issues.

As an example, consider a matrix multiplication in CUDA shown in Figure 1, in its most basic form.

Authors' Contact Information: Guido Martínez, Microsoft Research, Redmond, USA; Bastian Köpcke, TU Berlin, Berlin, Germany; Jonáš Fiala, ETH Zurich, Zurich, Switzerland; Gabriel Ebner, Microsoft Research, Redmond, USA; Tahina Ramananandro, Microsoft Research, Redmond, USA; Michel Steuwer, TU Berlin, Berlin, Germany; Tyler Sorensen, Microsoft Research, Redmond, USA; Nikhil Swamy, Microsoft Research, Redmond, USA.



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

This *kernel* (a function to be executed in parallel on the GPU) multiplies two matrices A and B (of sizes $M \times K$ and $K \times N$, respectively), storing the result in C. This is done by spawning enough threads to cover all the cells of C, where each thread computes one cell by iterating over the K dimension. The kernel function `ker` executes in parallel over a “grid” of $(M \times N / 1024)$ “blocks” of 1024 threads each (assuming 1024 divides $M \times N$). In line 2, every thread computes its global ID (`gid`), from which it derives the row and column indices (`i` and `j`), and then computes the relevant dot-product (lines 4 & 5) and stores it in C. The `matmul` function is the host-side function (executing on the CPU) that configures and launches the kernel in line 10, with appropriate parameters and grid configuration.

There are several subtleties for even this simplest kernel to work correctly. First the grid configuration (number of blocks and threads) must cover all the cells of C exactly, otherwise some cells will not be computed or some threads will access memory out of bounds. To avoid data races, every thread must write to a distinct cell of C and the input matrices must not be modified while the kernel is running. During kernel execution, conceptually, the threads share read-only access to A and B, and each thread has exclusive write access to a single cell of C. Finally, the accesses to the arrays must be in bounds, and at the correct offset according to the matrix representation (in this case, row-major). As kernels become more complicated (involving synchronization, matrix tiling, block-wide shared memory, tensor core operations, etc), these challenges are exacerbated.

```
__global__ void ker(int K, int N, float *A,
                  float *B, float *C) {
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    int i = gid / N; int j = gid % N; float acc = 0.0;
    for (int l = 0; l < K; l++)
        acc += A[i * K + l] * B[l * N + j];
    C[i * N + j] = acc;
}

void matmul(int M, int K, int N, float* A,
            float* B, float* C) {
    int nblks = M * N / 1024;
    ker<<<nblks, 1024>>>(K, N, A, B, C);
}
```

Fig. 1. Naive matrix multiplication in CUDA

Safe GPU Programming. Making GPU programming safer and easier is an active area of research, with two main themes of work. First, a variety of analysis tools have been developed to check GPU programs for memory safety and race conditions, e.g., PUG [Li and Gopalakrishnan 2010], GKLEE [Li et al. 2012], GPUVerify [Betts et al. 2012], and Faial [Liew et al. 2024]. These tools offer a high level of automation for checking certain classes of bugs, but they can also time out, report false positives, and are hard to extend to handle the latest features of the rapidly-evolving GPU hardware and programming model. They also offer no guidance to programmers on how to write correct GPU code, only flagging errors after the fact.

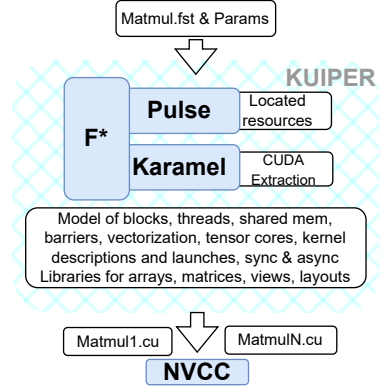
Along another axis, there are many domain-specific languages that aim to raise the level of abstraction for GPU programming, both to simplify programming and to improve safety. Prominent on the industrial side are languages like Triton [Tillet et al. 2021] and Cutlass/CuTe [Nvidia, 2025], which offer higher level languages embedded within Python or C++, aiming to simplify GPU programming but without trying to catch all potential errors at compile time. From the research community, languages like Halide [Ragan-Kelley et al. 2013], Lift [Steuwer et al. 2017], RISE [Hagedorn et al. 2020], Descend [Köpcke et al. 2024], and Futhark [Henriksen et al. 2017] offer new programming languages to raise the level of abstraction and improve safety, but they often sacrifice low-level control and performance, and are hard to extend with new GPU features. Besides, these existing languages only ensure basic safety properties, and do not extend to proofs of functional correctness.

1.1 KUIPER: An Extensible Framework for Correct & Efficient GPU Programming

We seek an extensible framework for GPU programming that offers high-level abstractions but with full control over low-level details, allowing programmers to eke out performance, while still providing strong static guarantees early in the development process, ranging from memory safety and data race freedom to full functional correctness. Our idea for how to achieve this is to embed support for CPU/GPU heterogeneous programming as library constructs (a so-called "shallow embedding") within the F* dependently typed programming language [Swamy et al. 2016]. As such, we present KUIPER, a framework whose architecture is shown alongside. KUIPER programs are written in the Pulse sub-language of F*, a C-like imperative, concurrent, higher-order language with full low-level control of memory management and layout. KUIPER libraries in Pulse model a reference GPU programming model, including for blocks, warps, threads, tensor cores, barriers, shared memory, and vectorized memory access. With GPU features described just as libraries, one can easily adjust or extend them to accommodate new features as the hardware evolves. On top of this model, we use Pulse's program logic to build abstract reasoning principles, so called kernel descriptions, that relate CUDA's per-thread programming model to a global view of the entire kernel. We also offer a variety of libraries to structure programming and reasoning, including for arrays, matrices, views, and matrix layouts.

To express all of this safely, we extended the Pulse program logic with a novel notion of *located resources*, allowing us to encode the accessibility of resources from different devices and threads. We also made a small number of additions to Karamel, an existing F* to C transpiler [Protzenko et al. 2017], to emit code that maps cleanly to CUDA.

A Naive Matrix Multiplication in KUIPER. Figure 2 shows the CUDA matrix multiplication kernel shown earlier programmed in KUIPER, slightly simplified to fit in the paper. The parameters of `kf` include the input matrices, constrained to have the right dimensions, while the block ID `bid` and thread ID `tid` is of C type `size_t`, but refined to ensure they are in bounds for the grid configuration. The pre- and postconditions in lines 3 and 4 are in Pulse's separation logic, and state that the function preserves read-only access to the input matrices `a` and `b` and does not modify their contents, `va` and `vb` respectively. Additionally, it requires exclusive write access to the cell `(row, col)` of the



```

1 fn kf {| scalar et |} (bid : szlt nblk) (tid : szlt nthr) (a:gpu_matrix et m k layoutA)
2   (b:gpu_matrix et k n layoutB) (c:gpu_matrix et m n layoutC)
3   preserves a ↦ Frac f va * b ↦ Frac g vb requires c[row, col] ↦ _
4   ensures c[row, col] ↦ mm_dotprod va vb row col {
5     let gid = 1024sz * bid + tid; let row, col = gid / k, gid % k;
6     let mut sum : et = zero; let mut l : SZ.t = 0sz;
7     while (!l < k) invariant live l * pure (!l ≤ k) * sum ↦ mm_part va vb row col !l {
8       sum ← !sum + a[row,l]*b[l,col];
9       l ← l + 1sz;
10    };
11    c[row, col] ← sum; }

```

Fig. 2. Naive matrix multiplication in KUIPER: kernel function

output matrix c (which cannot alias a or b). The postcondition further states that this cell of c now contains the correct result of the matrix multiplication at this position. The loop in line 9 is decorated with a loop invariant stating that at each iteration $!l$, the variable sum contains the partial dot-product of the row and column up to $!l$. Of course, the postcondition of this function alone is not enough to guarantee that the entire matrix is multiplied correctly—a key contribution of our work is showing how to modularly lift per-thread specifications (such as this one) to global specifications of the entire kernel.

Generic kernels, enabling verified auto-tuning. While the code itself is relatively close to the CUDA version, we already see here some of the higher level abstractions of KUIPER at work. This `kf` function is parametric in a type class for any scalar type `et`. Further, although the matrices a , b , and c are just flat arrays in memory, we access them using *views* that support indexing with tuples (e.g., `a[row, l]`), and the kernel is also polymorphic in the *layouts* of the matrices, so the same code can be used for say, row-major or column-major layouts, or even more exotic layouts, separately for each matrix, without changing the code or the proof. In KUIPER, we verify a generic kernel once and provide many instantiations across different parameterizations, generating CUDA variants that can be compiled with `nvcc` or integrated into existing frameworks. This enables a single specification to accommodate diverse matrix layouts; for example, `llama.cpp` [ggml.org] fixes the second matrix in a transposed form, while PyTorch [Ansel et al.] defaults to row-major storage yet permits arbitrary strides. In doing so, KUIPER avoids the tedious and error-prone task of rewriting kernels for each setting, yielding a single flexible and reusable implementation. We report on the performance implications of different verified instantiations in §5 to find high-performance variants of kernels specialized to specific hardware types, layouts, and matrix dimensions.

Concretely, in introducing KUIPER, we offer the following contributions:

- We extend the PulseCore program logic with a novel notion of locations and located resources, useful for modelling heterogeneous memory models, and introduce a new connective ($\forall^+$) for universal separating conjunction with refinement types over possibly infinite domains. (§3.1, §3.2)
- A model of CPU/GPU programming as a shallow embedding in Pulse, with specifications using dependent types and separation logic accounting for the intricacies of the heterogeneous programming model, the GPU’s memory hierarchy, synchronization, vectorized loads & stores, tensor cores, and both synchronous and asynchronous kernel launches. (§3.3)
- Libraries for data abstraction, including a formally-verified “view” framework, which can be used to write *layout-polymorphic* algorithms, reducing verification effort and increasing code reuse. This view framework also provides a generic account of matrix *tiling*, a common ingredient of optimized matrix algorithms, enabling verified auto-tuning of kernels. (§4.1, §4.4)
- A specification & proof technique to generically verify the functional correctness of algorithms over scalar types, obtaining precise results for types without numerical error (e.g., integers) and approximate results (ignoring numerical error) for floating-point numbers. (§4.2)
- We show this approach can produce realistic CUDA code by implementing several optimized GEMMs, providing the first GPU kernels with machine-checked proofs of functional correctness, including for those using features such as 2D block-tiling and tensor cores. We also show that the performance of our verified kernels is comparable to hand-written code, and is competitive with state-of-the-art libraries such as cuBLAS. (§5)

Limitations. Despite our desire to provide formal guarantees about the safety and correctness of CPU/GPU programs, we note that our results are far from foundational. To start with, an official semantics of CUDA or the GPU hardware does not exist, so the model we provide in Pulse via a shallow embedding is our best effort at capturing the semantics, based on our understanding of the documentation and prior work. Some aspects of the documentation are especially subtle and

difficult to model accurately in a logic, especially syntactic aspects, e.g., the requirements kernel threads must synchronize on the same barrier *instruction*. Also, Pulse so far only offers partial correctness—proving total correctness, including deadlock freedom, is interesting future work. We discuss these limitations and offer some possible directions for addressing them, though we believe KUIPER is already a significant step forward in providing strong guarantees for heterogeneous programming with CPUs and GPUs.

2 Overview

We begin by presenting an informal overview of KUIPER, starting with primers on GPU programming and separation logic. As we will see, the basic ideas of separation logic are well-suited to reason modularly about massively parallel GPU kernels, though many subtleties arise in accounting for the GPU’s memory hierarchy and synchronization model—subtleties that we account for in KUIPER using a newly enhanced version of the Pulse logic.

2.1 A Primer on GPU Programming

GPUs are massively parallel programmable accelerators. GPU execution and memory management is coordinated by the CPU, referred to as the *host*, while the GPU is called the *device*. The two have distinct memory spaces, and device memory must be explicitly managed by the host through the CUDA runtime, e.g., using `cudaMalloc` to allocate memory and `cudaMemcpy` to transfer data between device and host memory regions. Although the host cannot directly read or write device memory (and vice versa), both sides may exchange opaque pointers, which can obscure ownership and lead to subtle correctness issues.

GPU computation on the device begins at a *kernel*, which can be *launched* by the host (i.e., the CPU). Kernels are annotated functions written in CUDA, a dialect of C++. When launched, this function will be executed by many GPU threads in parallel. These threads are organized hierarchically, closely mirroring the GPU hardware, which allows to organize a kernel in ways that maximize memory locality at the varying levels. At the lowest level, a *thread* executes a sequential stream of instructions, much like a CPU thread. Threads are grouped into warps (32 threads for NVIDIA GPUs), which execute in a Single Instruction, Multiple Threads (SIMT) fashion: that is, threads in a warp issue instructions together. Warps are further collected into *thread blocks* (or just *blocks*), which are programmer-defined units containing up to 1024 threads and scheduled onto a single *streaming multiprocessor*. Different blocks run concurrently, without synchronizing. A kernel launch (from the host) specifies the number of thread blocks, which can be very large, and the size of the thread blocks. This configuration is known as the *grid*.

Threads can query identifiers at each level of this hierarchy, such as their thread ID within a block and their block ID within the grid. This allows them to index disjoint regions of input data and take divergent control-flow paths when necessary.

Threads that share lower levels of the hierarchy can exploit features provided by their hardware spatial locality. At the warp level, threads execute in close coordination, and modern GPUs expose specialized warp-level features that can significantly improve performance. The most prominent are tensor cores, which perform small matrix multiplications. These operations can be composed across warps to implement large-scale linear algebra operations. The interface provides warp-wide collective operations that load memory tiles and execute fixed-size matrix multiplications. These operations require explicit specifications of memory layout, strides, base pointers, and matrix dimensions. At the block level, threads can cooperate through efficient *shared memory*, a software-managed cache. Threads within a block can synchronize using the `__syncthreads()` barrier call.

High-performance kernels must explicitly manage this hierarchy and carefully compose optimizations at each level. Doing so requires precise reasoning about memory visibility, ownership, and synchronization across thousands of concurrently executing threads. Figure 3 shows the different locations in a heterogeneous CPU/GPU program, and the visibility of references allocated in each location. The CPU process can access CPU memory, but not GPU memory. Each GPU has its own memory space, accessible by all its blocks and threads. However, shared memory is only visible to threads within the same block. Ensuring correctness under these constraints is notoriously difficult and motivates the need for verification techniques.

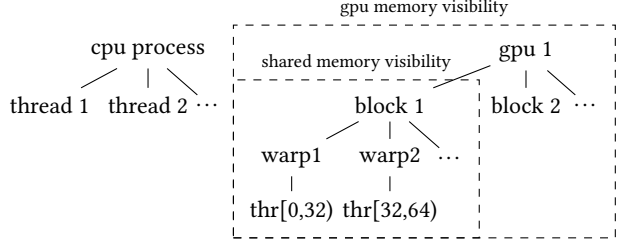


Fig. 3. Memory locations and visibility of references

2.2 A Primer on Separation Logic

Separation logic [Reynolds 2002], a variant of Hoare logic, was initially developed by John Reynolds as a way to reason modularly about programs that manipulate pointers and heap-allocated data structures. Later, Peter O’Hearn observed that the logic applies naturally to reasoning about concurrent programs as well [O’Hearn 2007], leading to the development of concurrent separation logic, of which there are a number of sophisticated modern variants, of which Pulse is one.

The key idea of a separation logic is to describe program properties with assertions that express both *ownership* of heap fragments as well as properties about the contents of those fragments. These assertions can be combined using various connectives, most notably the *separating conjunction*. For example, the assertion $x \mapsto 5$ states that the current thread owns the heap fragment containing memory address x (no other thread can read or write it), and further that that memory location stores the value 5. Additionally, the assertion $x \mapsto 5 * y \mapsto 10$ states that the current thread owns two disjoint heap fragments, one where x points to 5 and another where y points to 10.

O’Hearn observed that separation logic yields simple proof rules to reason modularly about concurrent programs. For example, the rule below to reason about the parallel composition of two programs c_1 and c_2 states that, if c_1 can be shown to satisfy a Hoare triple with precondition P_1 and postcondition Q_1 , and similarly for c_2 with P_2 and Q_2 , then the parallel composition $c_1 || c_2$ satisfies a Hoare triple with precondition $P_1 * P_2$ and postcondition $Q_1 * Q_2$.

$$\frac{\{P_1\} c_1 \{Q_1\} \quad \{P_2\} c_2 \{Q_2\}}{\{P_1 * P_2\} c_1 || c_2 \{Q_1 * Q_2\}}$$

This should make intuitive sense: the two threads operate on disjoint parts of the heap, so they cannot interfere with each other, and thus we can reason about them independently. Standard generalizations of this rule allow the two threads to share read-only access to overlapping parts of the heap, so long as they only write to disjoint parts, enforcing data-race freedom.

The same idea applies to the vast number of GPU threads: as long as each thread only writes to a disjoint part of memory, we should be able to reason about each thread independently. Further, since every thread is executing the same code with different parameters and data, we should be able to prove the correctness of a single thread for all values of its parameters and data, and then lift that to a correctness proof of the entire kernel. A naïve first attempt at a proof rule reflecting this reasoning principle follows: if the pre- and postconditions can be split according to the grid

of blocks and threads, then so long as each thread can be proven to consume its fragment of the precondition and produce its fragment of the postcondition, they can all be combined.

$$\frac{\forall i, j. \{P_{i,j}\} \text{ kf}(i, j) \{Q_{i,j}\} \quad 0 \leq i < B \quad 0 \leq j < T}{\{\ast_{i,j} P_{i,j}\} \text{ launch}_{\langle B, T \rangle}(\text{kf}) \{\ast_{i,j} Q_{i,j}\}}$$

However, this does not quite work as stated, for several reasons. First, the GPU memory model is more complicated than a simple shared heap: there are multiple kinds of memory (CPU memory, global, shared, local), with different visibility and lifetime properties, some of which get logically allocated “on the fly” to each block at the time the kernel is launched. For example, referring back to our simple matrix multiplication kernel, the input GPU arrays passed to the kernels are not even accessible from the CPU where the launch command is issued; so we cannot assert ownership of them in the precondition of the launch command. Second, the GPU programming model includes synchronization primitives (barriers, atomics) that allow threads to communicate and share data, which complicates the reasoning principles. Finally, the kernel launch itself can be asynchronous, meaning the host code continues executing while the kernel runs on the GPU, which complicates reasoning about the overall program.

2.3 Located Resources and a Proof Rule for Launching Kernels

Our first innovation is to extend Pulse with a notion of *located resources*. We introduce a type `loc_id`, which represents the memory location where resources can be located in the hierarchy of blocks, warps, and threads in the grid. A new predicate, `on l p` asserts ownership of the resource `p` located at `l`. For example, `on gpu0 (a ↦ va)` could be asserted on any thread (e.g., on CPU threads) to indicate ownership of the GPU array `a` containing values `va` in GPU0’s memory. Unlike `a ↦ va`, the on-guarded resource cannot directly be accessed by the CPU thread: it merely asserts that the resource exists on the GPU with its stated value. Further, the predicate `loc l` indicates that the current thread of execution is at location `l`, where `loc l * on l p` is sufficient to deduce `p`, i.e., a thread can access its own resources. By default, resources like `loc l` or `a ↦ v` cannot be transferred between locations, whereas `on l p` is suitably guarded and can be freely shared across locations. Further, we write `cpu` to assert that the current thread is running on the CPU.

This allows us to state a better version of the proof rule for launching kernels, also making use of $\forall^+$, a new form of universal separation conjunction (over possibly infinite domains) using refinement types as opposed to the iterated, finite separating conjunction.

$$\frac{C = \text{launch}_{\langle B, T \rangle}(\text{kf}) \quad \forall i, j. \{P_{i,j}\} \text{ kf}(i, j) \{Q_{i,j}\}}{\{\text{cpu} * \forall^+(ij: \mathbb{N}\{i < B \wedge j < T\}). \text{ on } (\text{tid } ij) P_{i,j}\} C \{\forall^+(ij: \mathbb{N}\{i < B \wedge j < T\}). \text{ on } (\text{tid } ij) Q_{i,j}\}}$$

This rule is sound in our model, in that it requires proving on the CPU thread that the resources exist at each thread’s location before the launch, that these resources can be granted to each thread before it runs, and their postconditions resources can be guarded and transferred back to the CPU. However, this exposes to the CPU the grid’s memory hierarchy (with assertions speaking about resources accessible at each thread). We would like instead to abstract some of the details and describe only, say, resources in GPU global memory accessible to all threads.

Located resources allow expressing a notion of *visibility* of resources from different locations in the hierarchical GPU memory model as well as the CPU host. For example, a resource located in GPU global memory is visible to all threads on that GPU, whereas a resource located in block shared memory is only visible to threads within that block—both locations are inaccessible to CPU threads. We write `is_send_across vis p` to indicate that the resource `on l1 p` can be transferred to `on l2 p`, so long as `vis l1 == vis l2`. With this, we can state a better version of the rule, yielding

a specification that only mentions resources in GPU global memory:

$$\frac{C = \text{launch}_{\langle B, T \rangle}(\text{kf}) \quad \forall i, j. \text{is_send_across_gpu_of } P_{i,j} \quad \{P_{i,j}\} \text{ kf}(i, j) \{Q_{i,j}\}}{\{\text{cpu} * \text{on_gpu0} (\bigvee^+ (ij : \mathbb{N} \{i < B \wedge j < T\}). P_{i,j})\} C \{\text{on_gpu0} (\bigvee^+ (ij : \mathbb{N} \{i < B \wedge j < T\}). Q_{i,j})\}}$$

This too is not the full picture, since it does not account for GPU shared memory, barriers, and asynchrony, but this rule is useful to reason about simple kernels launches, such as for the naïve matrix multiplication shown in Figure 2. In KUIPER, one can write the following code to launch the kernel with 1024 threads on a single block:

```
fn matmul' (a b c:gpu_matrix et 1024 1024)
  requires cpu preserves on gpu0 (V+ i j. a ↦ Frac (1/1024) va * b ↦ Frac (1/1024) vb)
  requires on gpu0 (V+ i j. c[i,j] ↦ _)
  ensures on gpu0 (V+ i j. c[i,j] ↦ mm_dotprod a b va vb i j)
{ launch_sync1_n 1024 kdesc; }
```

Here, we require to be on a **cpu** thread, and for the input matrices to be split into read-only permissions for all the threads, while the output matrix is split into per-cell write permissions. But this too is not the ideal top-level specification. To enable packaging kernel calls into useful top-level specifications, KUIPER offers *kernel descriptions*: a library-level abstraction that packages a kernel function along with its grid configuration together with proof steps to make it easier to apply the core proof rules.

In the case of our matmul kernel, we can build the following kernel description, packaging a proof step setup and teardown ghost functions to precede and follow the kernel launch, where we write *pre* for the precondition of *matmul'*, *q* for its postcondition, and $m \rightsquigarrow n$ for a proof step that transforms the assertion *m* into *n* (i.e., a *shift* in Pulse, a form of separation logic implication):

```
val setup : (on gpu0 (a ↦ va * b ↦ vb * c ↦ vc)) ~> p
val teardown : q ~> (on gpu0 (a ↦ va * b ↦ vb * c ↦ a `matmul` b))
let kdesc : kernel_desc = { kf; nblk=1; nthr=1024; setup; teardown }
```

Using this kernel description to launch our matmul kernel, we can give the canonical top-level specification for matrix multiplication:

```
fn matmul (a b c:gpu_matrix et 1024 1024)
  requires cpu preserves on gpu0 (a ↦ va * b ↦ vb)
  requires on gpu0 (c ↦ _) ensures on gpu0 (c ↦ a `matmul` b)
{ launch kdesc; }
```

2.4 Shared Memory, Barriers, Vector Operations, and Tensor Cores

We now turn to our model of some of the more advanced features of GPU programming.

Shared memory. When launching a kernel in CUDA, the programmer can specify the amount of shared memory to allocate for each block. Concretely, in CUDA syntax, a kernel call with shared memory looks like `kernel<<<nblk, nthr, shmem_size>>>(...)`, where *shmem_size* is the number of bytes of shared memory to allocate per block. Within a kernel, one simply declares a local array such as the one below to obtain a pointer into the shared memory for that block:

```
extern __shared__ float shared_array[256];
```

This, of course, opens the possibility for various kinds of mistakes. For instance, the size of the `shared_array` in the kernel must not exceed the `shmem_size` specified at launch time. Further, threads within a block must carefully coordinate their accesses to shared memory, ensuring that there are no data races on accesses to a block's shared memory, typically done using barriers.

Additionally, as explained by the visibility rules from Figure 3, attempting to read shared memory from another block is illegal, e.g., a kernel that mistakenly stashes a pointer to shared memory in global memory and then attempts to read it from another block would lead to undefined behavior.

KUIPER offers a higher level, typed API for working with shared memory safely. When launching a kernel with shared memory, the kernel description does not directly specify the number of bytes of shared memory to allocate, but rather a provides a typed, polymorphic *shared memory descriptor*. For instance, for a polymorphic kernel working on matrices of a scalar type `et`, one can write the following shared memory descriptor to allocate space for two shared memory arrays, the first holding an array of `et` elements whose size is determined by the type parameter `tile_size`, and the second holding an array of 32-bit unsigned integers whose size is the number of threads in a block.

```
[SHArray et (tile_size * tile_size); SHArray u32 nthr]
```

KUIPER computes the bytes to allocate and the offsets into shared memory from the descriptor, ensuring that the kernel never exceeds the allocated shared memory or attempts to address it at an incorrect offset (accounting both for alignment and size). The type of a pointer to shared memory is `shmem_array t sz is equal to x:gpu_array t sz{visibility_of x == block_of}`, an array type like any other array, but with a refinement that ensures its points-to assertion $a \mapsto va$ only satisfies a more limited visibility rule, i.e., `is_send_across block_of (a  $\mapsto$  va)` holds, meaning that it can only be sent to other threads within the same block. For a descriptor `sh`, `refs_of sh` is a tuple of pointers to the shared memory arrays (e.g., `shmem_array et tile_size2 & shmem_array u32 nthr`) and `live_shmem (ptrs:refs_of sh)` is a predicate (possibly fractional) describing the ownership of the shared memory (e.g., `fst ptrs  $\mapsto$  _ * snd ptrs  $\mapsto$  _`).

Barriers and __syncthreads(). To synchronize access to shared memory in CUDA, one uses the `__syncthreads()` barrier call, which blocks until all threads in the block have reached the barrier. This primitive can be used to shuffle access to parts of shared memory between threads. For example, at iteration `i` of a loop, thread `j` may write to shared memory location $(i+j) \bmod k$, and so requires write access to that cell. By waiting at the barrier, all threads in a block waiting until the writes of a given iteration are done, and then take over write access to the next cell from the next thread in the block. In practice, the pattern of shuffling access to shared memory between threads can be quite complex and getting it wrong leads quickly to data races and corruption.

KUIPER models the `__syncthreads()` primitives as a barrier validating the following Hoare triple [Hobor and Gherghina 2011]. The assertion `barrier_token(p, q, i, it)` describes a barrier to shuffle access to resources based on the thread id `i` and iteration count `it`. Informally, at iteration `it`, when blocking at a `__syncthreads()`, the barrier requires thread `i` to *give up* ownership of resource `p(i, it)` and gain `q(i, it)` for use in the next iteration.

$$\left\{ \begin{array}{c} \text{thread_id } i \\ \text{barrier_token}(p, q, i, it) \\ p(i, it) \end{array} \right\} \text{syncthreads}() \left\{ \begin{array}{c} \text{thread_id } i \\ \text{barrier_token}(p, q, i, it + 1) \\ q(i, it) \end{array} \right\}$$

Of course, to create a barrier token one has to prove `admissible(p, q)`; that is, for all iterations `it`, $\forall^+ i. p(i, it) \rightsquigarrow \forall^+ i. q(i, it)$, so that the resources given up by all threads at iteration `it` can be re-distributed to all threads for the next iteration.

Kernel launches, generally. With shared memory descriptors and barrier tokens, we can now give the full proof rule for launching kernels in KUIPER, where in addition to blocks `B` and threads `T`, the launch also specifies the shared memory descriptor `sh` and the predicates `p` and `q` to use for `syncthreads()` barriers.

$$\begin{array}{c}
\text{admissible}(p, q) \quad \forall i, j. \text{is_send_across } \text{gpu_of } P_i, Q_i \quad \text{is_send_across } \text{block_of}(R_{i,j}, S_{i,j}) \\
(\forall^+ i. P_i * \text{live_shm}(\text{refs_of } sh) * \text{barrier_tok}(p, q, j, 0)) \leadsto (\forall^+ i, j. R_{i,j}) \\
\{R_{i,j}\} \text{kf}(i, j) \{S_{i,j}\} \quad (\forall^+ i, j. S_{i,j}) \leadsto (\forall^+ i. Q_i) \\
\hline
\{\text{cpu} * \text{on } \text{gpu} (\forall^+ i. P_i)\} \text{launch} \langle B, T, sh, p, q \rangle \langle \text{kf} \rangle \{\text{on } \text{gpu} (\forall^+ i. Q_i)\}
\end{array}$$

First, we require p and q to be admissible for use in barriers and for P_i and Q_i to be sendable across the entire GPU. $R_{i,j}$ and $S_{i,j}$ are the per-thread pre- and postconditions, and these need to be sendable across the block. Then, we require that the precondition P_i together with ownership of the shared memory and a barrier token for block i and thread j in iteration 0 can be shifted to the per-thread preconditions $R_{i,j}$. Then, we prove the Hoare triple for each thread $\text{kf}(i, j)$, starting with its precondition $R_{i,j}$ and producing its postcondition $S_{i,j}$. Finally, one must prove that the per-thread postconditions $S_{i,j}$ can be combined and shifted to the per-block postconditions Q_i , and used in the conclusion of the rule. Throughout, we implicitly bound i by the number of blocks B and j by the number of threads T . Finally, as mentioned in §2.3, we package all these proof steps into the kernel description to make it easier to apply the rule. Note, our implementation also supports asynchronous launches and a corresponding proof rule, which we omit here for lack of space—pleasantly, the same kernel description can be used for both synchronous and asynchronous launches, only the postcondition of the launch invocation changes to account for asynchrony.

Vectorized Operations. CUDA provides vectorized copies of 16 bytes that are more efficient than regular array accesses. However, vectorized accesses come with alignment requirements. We model this in KUIPER with refinement types and type classes: `gpu_array_vec_cpy` `dst dst_off src src_off` is parametric in a type `et` that has a static size and satisfies the `has_vec_cpy` type class (e.g., the types `f16`, `f32`, etc. support vectorized copy), requires the offsets to be in bounds, and for the offsets from the base addresses of the arrays to be aligned to 16 bytes. The pre- and postconditions specify that a chunk of data of size `chunk et` is copied from `src` to `dst` at the specified offsets.

```

fn gpu_array_vec_cpy { | sized et, has_vec_cpy et | }
  (dst : gpu_array et dst_sz) (dst_off : SZ.t)
  (src : gpu_array et src_sz) (src_off : SZ.t)
  { m ≤ dst_off ∧ dst_off + chunk et ≤ n ∧ i ≤ src_off ∧ src_off + chunk et ≤ j ∧
    aligned 16 src src_off ∧ aligned 16 dst dst_off }
preserves src[i,j] ↦f ss requires dst[m,n] ↦ ds
ensures   dst[m,n] ↦ blit ds (dst_off - m) ss (src_off - i) (chunk et)

```

3 Semantic Foundations

We now present the semantic model on which we develop KUIPER, including our theory of located resources from §3.1 and the new $\forall^+$ connective—both of these are independent of GPU programming, and are of general use. We then present briefly our operational model of CUDA kernel launches, with blocks, threads, and shared memory—these are the reference semantics on which we prove the soundness of the proof rules presented previously.

3.1 A Theory of Located Resources

As explained previously, for KUIPER, we need to restrict the transfer of resources between different locations in the GPU memory hierarchy, e.g., $a \mapsto va$ to an array allocated in shared memory must not be transferred to another block or even the CPU. However in concurrent separation logic *invariants* usually allow any resource to be exchanged between threads.

To restrict the sharing of resources, with inspiration from prior logics, notably RustBelt [Jung et al. 2018a], we enhance PulseCore (Pulse’s core logic) with a notion of *located resources*. We did this by changing PulseCore’s definition of separation logic predicates from $\text{mem} \rightarrow \text{prop}$ to $\text{mem} \times \text{loc_id} \rightarrow \text{prop}$. Then we can define the predicates $\text{loc } l = \lambda (m, t) \rightarrow \text{pure } (l == t)$ and $\text{on } l \ p = \lambda (m, t) \rightarrow p \ m \ l$. This is similar to RustBelt’s modeling of the thread-aware ownership semantics of a type where their type $\text{TID} \times \text{List}(\text{Val}) \rightarrow \text{iProp}$, adds an extra explicit thread ID argument. We have opted to make the location of a resource implicit. This has the advantage that single-threaded code does not require any changes for use with located resources.

We then define $\text{is_send_across } vis \ p$ to indicate that the resource p can be sent from l to any location l' such that $vis \ l == vis \ l'$, as mentioned in §2.3. A special case of this are the *placeless* resources that can be sent to any location, i.e., $\text{placeless } p = \text{is_send_across } (\lambda _ \rightarrow ()) \ p$. Any predicate can be made placeless by specifying its location with $\text{on } l \ p$, meaning that the resource p is available at location l . RustBelt’s semantics for the Rust Send trait has almost exactly the same definition as our *placeless* (though RustBelt uses a magic wand while Pulse uses a ghost function). Our definition of is_send_across is more general, as it allows specifying different visibility regions, not just the universal one.

The only basic predicates that are not placeless are $\text{loc } l$ and heap arrays $a \mapsto va$. The location predicate $\text{loc } l$ cannot be sent to any other thread (as that thread already owns $\text{loc } l'$ for a different location l'). Heap arrays can be sent across a configurable *visibility region* specified at allocation time: arrays allocated on the CPU can be sent to any other thread in the same process, arrays allocated in the shared memory can be sent to any thread in the same block, etc. PulseCore exposes a primitive that eliminates $\text{loc } l * \text{on } l' \ (a \mapsto va)$ to $a \mapsto va$ if l and l' are in the same visibility region. This visibility region is stored in the pointer in our model, and can be accessed with the vis_of function. KUIPER’s **cpu** resource is defined as $\exists^* l. \text{loc } l * \text{pure } (\text{cpu_of } l == \text{cpu_loc})$, saying that the current thread is on the CPU, and also an analogously for the **gpu** resource. We have also adapted PulseCore’s notion of impredicative invariants to work smoothly with located resources, though a full description of this is out of scope of this paper.

3.2 A New Connective: Universal Separating Conjunction with Refinement Types

In KUIPER, we often need to represent families of resources: for example, the ownership of an array cell for every matrix index, the ownership of some shared memory for every thread index, etc. The most basic way to model this is with a “big” conjunction operator $\bigstar_{i=0}^n p_i$ defined by recursion on n (i.e., $\bigstar_{i=0}^{n+1} p_i = p_{n+1} * \bigstar_{i=0}^n p_i$). However such a combinator is surprisingly cumbersome to use in practice. First, it requires us to choose a concrete indexing scheme since we typically want to index over other types than the first n natural numbers. Second, an extremely common operation is removing elements from this list. Removing a single element is already fairly verbose, resulting in two $\bigstar$ s: $\bigstar_{i=0}^n p_i = (\bigstar_{i=0}^{k-1} p_i) * p_k * (\bigstar_{i=k+1}^n p_i)$. Removing *two* elements $k \neq l$ is a formidable task if we don’t know whether $k < l$ or $l < k$. And third, just in order to state $\bigstar$ we need to prove not just the finiteness of the index domain, but also compute its precise size. These issues make $\bigstar$ extremely tedious to use in practice, particularly as we want to state properties over much richer index domains.

We introduce a $\forall^+$ quantifier that solves all of these problems: $\forall^+(i : \alpha). p_i$ is defined for any type α . Removing an element results in only a single $\forall^+$ quantifier: $(\forall^+(i : \alpha). p_i) = p_k * (\forall^+(i : \alpha\{i \neq k\}). p_i)$. (The type $(i : \alpha\{i \neq k\})$ is a *refinement type* consisting of all elements i of α that satisfy $i \neq k$.) Removing two elements $k \neq l$ is just as easy: $(\forall^+(i : \alpha). p_i) = p_k * p_l * (\forall^+(i : \alpha\{i \neq k \wedge i \neq l\}). p_i)$. Supporting infinite domains allows us to easily state properties like $\forall^+(x : \mathbb{N}). \forall^+(y : \mathbb{N}\{x + y \leq 10\}). p_{xy}$ without worrying about finiteness.

The basic API for $\forall^+$ allows us to introduce $(\forall^+(x : \alpha). \text{emp}) = \text{emp}$, $\text{split } (\forall^+(x : \alpha). p_x) = (\forall^+(x : \alpha\{f(x)\}). p_x) * (\forall^+(x : \alpha\{\neg f(x)\}). p_x)$, add singletons $(\forall^+(x : \alpha\{x = y\}). p_x) = p_y$, as well as a `trade_map` function allowing us to map ghost operations over a $\forall^+$.

F^* supports refinement types natively and provides built-in automation; making this API more ergonomic than lists or sets. For example, $x : \alpha\{\text{True}\}$ and α unify and are definitionally equal; similarly we have $x : (x : \alpha\{f(x)\})\{g(x)\} = x : \alpha\{f(x) \wedge g(x)\}$. Also, we can usually treat an element $x : \alpha$ transparently as an element of $y : \alpha\{f(y)\}$ with Z3 automatically discharging the proof obligation $f(y)$.

Iris defines a comparable combinator on lists (and using that, on finite sets, finite multisets, and finite maps). Defining $*$ on finite sets also allows for easily removing elements. Compared to our approach, Iris' combinator also supports elements occurring more than once in a multiset; we have not found a need for that in KUIPER.

3.3 A Reference Semantics for Kernel Launches

Using these two new ingredients, located resources and $\forall^+$ we can now present our reference semantics for kernel launches in KUIPER. Our goal is to show that the kernel launch proof rule presented in §2.3 has a model in our extended version of Pulse, and is a reasonably realistic if highly simplified model of CUDA kernel launches.

Pulse already has a model of concurrency based on spawning threads in a shared-memory setting. Using located resources this is now enhanced to support spawning threads at different locations, as described by the rule below.

$$\frac{\text{is_send_across } \text{vis } (P_1, Q_1) \quad \{ \text{loc } l * P_1 \} f_1 \{ \text{loc } l * Q_1 \} \quad \{ Q_1 \} f_2 \{ Q_2 \} \quad \text{vis } l_0 = \text{vis } l}{\{ \text{loc } l_0 * P_1 * P_2 \} \text{ par } \text{vis } l_0 \quad l \quad f_1 \quad f_2 \{ Q_1 * Q_2 \}}$$

We use this to implement kernel launches from a kernel description by recursively spawning threads for each block. The block-level thread allocates shared memory according to the shared memory descriptor and then spawns threads for each thread in the block. Each thread runs the kernel function with its own thread id, and the pre- and postconditions of the kernel function are combined using $\forall^+$ over the thread ids. The pre- and postconditions of the block-level thread are then combined using $\forall^+$ over the block ids. The sendability conditions in the kernel description ensure that all resources are sent and retrieved from the right locations.

Of course, this reference semantics of a kernel launching is very much part of our trusted computing base (which also includes our verification tools). It provides the formal model against which to judge the theorems we prove about our code. Whether or not this model is an accurate representation of the actual behavior of CUDA kernels is outside the scope of this paper, but we believe it is a reasonable first approximation. Note, we have not tried to model all primitives, e.g., we assume `mma_sync` as a primitive rather than modeling a tensor core within Pulse. One direction for the future could be provide executable models for all these intrinsics and to run our reference semantics with CPU threads to confirm empirically that it matches the behavior of actual CUDA kernels, though even this would have limitations, and would be quite a large undertaking.

4 Proof-oriented Libraries for KUIPER

We now discuss libraries in KUIPER for programming and proving CPU/GPU code, with abstractions to ease some common tasks and their associated proofs.

4.1 Verified Data Abstractions: Views, Matrices, and Tiling

GPU kernels have strict requirements on memory layouts and access patterns to achieve good performance. The choice of storing a matrix in row-major or column-major order can have a huge

significant impact on performance. Usually, GPU programmers manually work with strides, offsets, and pointer arithmetic to achieve the desired results. This is error-prone, and makes changing a layout after the fact difficult. However, while crucial for performance, layout choices are usually orthogonal to the high-level algorithm being implemented.

With a full-fledged proof-oriented language at our disposal, we can build a matrix library that abstracts issues related to strides and offsets, and enable *layout-polymorphic* programming, where algorithms are verified over an abstract layout, and then easily instantiated repeatedly for various specific layout choices. Crucially, this matrix type also permits *tiling*, i.e., viewing a matrix as a matrix of sub-matrices (tiles), a common technique to improve locality in memory accesses. We sketch main ideas of this construction, referring the reader to the supplement for details.

We construct our matrix library from the ground up, starting from Pulse’s array library. There, an array in Pulse is a pointer to a contiguous memory region, and is related (via the $\mapsto$ predicate) to a sequence (list) of values of the element type. Reading from the array, at some integer index, returns the corresponding element from the sequence, and writing to the array updates the sequence accordingly. Other operations include slicing an array into two sub-arrays, modeling taking a pointer to an offset within an array, splitting it into its prefix and suffix. There are three main problems to solve here: (1) the flat indexing, (2) the flat view as a sequence, and (3) the requirement for contiguity (and order) in memory. Concretely, we want to be able to index into a matrix with a row and a column and specify its contents as a mathematical matrix for arbitrary memory layouts, which may be non-contiguous (a necessity for tiles).

To achieve these goals, we begin by introducing a type of “virtual arrays”, `varray`. A `varray et vw` is an array of element type `et` which is accessed via some *view* `vw`. The view contains a type of indices `vw.idx`, which are used to refer to positions of the array. There is no restriction on this type (e.g. pairs of integers for matrices) so long as one can also define an injective map from indices into $\mathbb{N}_{< n}$, where n is the length of the underlying array. The view also contains a specification type `vw.st`, used to denote the contents of the array, which must be in bijection with sequences of type `et` and length n . The points-to predicate for an `varray et vw` relates it to a value of type `vw.st`. Reading and writing, on an index of type `vw.idx`, has the expected behavior via the bijection. Importantly, a `varray` permission can be blown up into permissions over its cells, and vice versa, using the $\forall^+$ connective described earlier.

We are now ready to define our matrix type. We first define the notion of a matrix layout, which is an array length together with an injection from row-column pairs into flat indices.

```
type mlayout (rows cols :  $\mathbb{N}$ ) = { len :  $\mathbb{N}$ ; map :  $\mathbb{N}_{< \text{rows}} \ \& \ \mathbb{N}_{< \text{cols}} \hookrightarrow \mathbb{N}_{< \text{len}}$ ; }
```

Any matrix layout defines a view where the indices are row-column index pairs, the index map is the one above, and the specification type is `matrix et rows cols`, i.e., mathematical matrices of the appropriate dimension.

```
val view_from_layout (l : mlayout 'rows 'cols) : view et (matrix et rows cols)
```

We define our matrix type `gpu_matrix et l` with `l : mlayout rows cols`, as an appropriate `varray`.

```
type gpu_matrix (et:Type) (l : mlayout rows cols) = varray et (view_from_layout l)
```

Many operations for matrices are directly inherited from the virtual array layer, such as reading and writing at a given row and column, exploding or imploding a matrix into its cells, etc. We can also define matrix-specific operations, such as tiling. For a matrix whose dimensions are multiples of some tile size, we can define the layout of a tile by a simple restriction of the outer layout. We also provide operations to turn ownership of a matrix into separate ownership of its cells, where `a /? b` denotes that `b` is a multiple of `a`.

```

val tile_layout (l : mlayout rows cols) (tr tc :  $\mathbb{N}$  {tr /? rows  $\wedge$  tc /? cols})
  (i :  $\mathbb{N}_{<}$  (rows / tr)) (j :  $\mathbb{N}_{<}$  (cols / tc)) : mlayout tr tc
val tile (m : gpu_matrix et l) (tr tc :  $\mathbb{N}$  {tr /? rows  $\wedge$  tc /? cols})
  (i :  $\mathbb{N}_{<}$  (rows / tr)) (j :  $\mathbb{N}_{<}$  (cols / tc)) : gpu_matrix et (tile_layout l tr tc i j)
ghost fn matrix_tile (l : mlayout rows cols) (m : gpu_matrix et l)
  (tr tc :  $\mathbb{N}$  {tr /? rows  $\wedge$  tc /? cols}) requires m  $\mapsto$  em
  ensures  $\forall^+ (i : \mathbb{N}_{<} (rows / tr)) (j : \mathbb{N}_{<} (cols / tc)).$ 
    tile m tr tc i j  $\mapsto$  matrix_tile em tr tc i j

```

Notably, the tiles are themselves of type `gpu_matrix et l'`, where all that changes is the layout. They can still be accessed with row and column indices (within the proper, smaller, bounds) directly, and the specification type is still a mathematical matrix. There is no need to compute offsets or strides, as is commonly done in CUDA. This is all done behind the scenes by the injection function of the sublayout for each tile, and relieves the user from these low-level details. As an example, assume we have a matrix `m` of type `gpu_matrix et (row_major 100 100)`, then we can tile it, extract a single tile `m1`, and access it as follows:

```
matrix_tile _ m 10 10; let m1 = tile m 10 10 4 5; let x = m1[2sz, 3sz];
```

This would generate the following access to the underlying array: `m[(10*4+2) * 100 + (10*5+3)]`. The expression comes from composing the row-major mapping with the restriction relevant for the tile. In practice, constant folding kicks in to simplify these expressions.

With the layout being a parameter of the `gpu_matrix` type, and not affecting the abstract view as a mathematical matrix, it becomes extremely simple to write layout-polymorphic algorithms. There is in fact nothing special to do: just abstract the layout as another parameter of the kernel. Later, these parameters can be instantiated to specific layouts (e.g. row-major, column-major) without needing to reverify the kernel.

Sometimes, particular properties of the layouts are required. For instance, CUDA provides vectorized copies of 16 bytes that are more efficient than regular array accesses. If we want to read a row of a matrix with such vectorized loads, we need to ensure that the layout maps a row into a contiguous sequence. Further, these operations usually require alignment of the starting address. We model all these constraints with refinement types on the layouts. In fact, we provide verified implementations for vectorized accesses over families of layouts that satisfy the necessary properties, and are used in our GEMM implementations.

This development provides a verified account of built-in view techniques used in other languages, for instance in Descend [Köpcke et al. 2024]. However, KUIPER’s views are also extensible: a user can define a new view by choosing their own index type and index mapping, and providing the necessary proofs. They are also fully verified, contrary to the built-in status of views in Descend.

We have omitted some details from this section. First, all of these predicates, and the `explode/implode` functions, work over any fraction of ownership. There is nothing special about full ownership. Second, for most of these constructs there are “abstract” versions and “concrete” versions. The abstract version usually works over (mathematical) natural numbers, and need not concern itself with overflow. However, for code generation, we need concrete implementations, e.g., the view’s index mapping must produce machine integers, that must be shown to not overflow. Hence, a concrete view is defined as, a concrete function that mimics the abstract view mapping. This is a common pattern in proof-oriented programming.

4.2 Approximate Reasoning for Floating Point, and Tensor Cores

Floating-point numbers are a well-known challenge when verifying numerical algorithms. In general, floating point operations are not exact, and lack basic properties such as associativity. This

means that the sum of an array depends on the order that we sum up its elements. For instance, we could give the reduce (sum) function a specification as follows (where `seq_fold_left zero add s` computes the sum $((0 + s_0) + s_1) + \dots + s_n$).

```
fn reduce #et {| scalar et |} (len:ℕ) (a : larray et len) (#s : seq et)
  preserves a ↦ s returns res : et ensures pure (res == seq_fold_left zero add s)
```

For algebraically well-behaved types like integers, this specification is correct regardless of how exactly the numbers are added up. But for floating point types, this specification is claiming the result is *exactly* the same as adding the elements in sequence order, basically enforcing exactly that implementation. Other summation orders, e.g. in a hierarchical way, do not satisfy this specification. We could adjust the specification to replace `seq_fold_left` for a pure function that mimics the actual order of operations performed by the implementation, but this makes the specification harder to understand, and less compositional.

In practice, CUDA programmers are usually very liberal about these sorts of numerical imprecisions, and if performance can be gained by changing the order of operations, they do so. For a reduction like above, any order of operations is acceptable, and an implementation would be considered correct as long as it does add up all of the array elements. We would like to ignore numerical error as well, but nevertheless verify some approximate functional correctness of numerical floating-point algorithms.

Most algorithms in KUIPER are polymorphic in the type of scalars and can be used for both integer and floating-point types. We want to give a specification for these algorithms that is on one hand fully precise for integer types but on the other hand also provable for floating-point types while giving some amount of confidence in the correctness. To this end, we introduce the `real_like` type class for scalars that have an interpretation as real numbers:

```
class real_like (t : Type) {| scalar t |} = {
  ( ≈ ) : t → real → prop;           (* an approximation relation *)
  to_real : x:t → y:real{x ≈ y};      (* a canonical real for every t *)
  (* compatibility between t operations and real operations *)
  zero_compat : unit → (zero ≈ 0.0);  one_compat  : unit → (one  ≈ 1.0);
  add_compat  : (x1 x2:t) → (y1 y2:real) → (x1 ≈ y1 ∧ x2 ≈ y2) → (x1+x2 ≈ y1+y2);
  mul_compat  : (x1 x2:t) → (y1 y2:real) → (x1 ≈ y1 ∧ x2 ≈ y2) → (x1*x2 ≈ y1*y2); }
```

The class defines an *approximation* relation between the scalar type and reals in a way that is compatible with the algebraic operations, such that every scalar is related to some real number. We also define related classes for additional operations like exponentiation. The $(\approx)$ operator can also be used over sequences and matrices, where it is applied pointwise. When using this class in specifications, we usually add a precondition stating that the inputs approximate some real values, and a postcondition stating that the output approximates the corresponding real result.

For integer types, we provide instances that allow for precise verification: `to_real` is the obvious map sending the integer to the corresponding real, and $(\approx)$ is equality modulo 2^n , where n is the bitwidth of the integer type. This makes $(\approx)$ injective: $m \approx x$ and $n \approx x$ imply $m = n$. This allows us to recover precise specifications when algorithms are used with integer types. That is, when the reduce function's postcondition claims that the results approximates the sum, we can derive that the result is actually equal to the sum.

For floating-point types, the provided instance is less precise. The `to_real` function still sends a floating-point number to the corresponding real, but $(\approx)$ is no longer injective. For example, both $3.0f/10 - 2.0f/10 - 1.0f/10 \approx 0.0$ and $0.0f \approx 0.0$ even though the two floating point numbers are different. (It is perfectly sound to use the universal relation for $(\approx)$.)

Nevertheless, this API gives useful verification guarantees even in the floating-point case. First, for polymorphic algorithms we have parametricity: the algorithm is going to execute the same operations for integers (where we have precise verification) as it does for floating-point numbers. (We do not expose equality or comparison operators on floating-point numbers.) This provides a guarantee against any kind of bug that can be observed on in the integer version: indexing errors, multiplying the wrong elements, forgetting part of a sum, etc.

Second, we can instantiate the scalar type with nonexecutable exact reals. The `real_like` instance for exact real numbers is just as precise as it is for integers. By parametricity, this instantiation performs the same operations as the floating-point version. This gives us the verification result that the result is correct if there were no rounding errors.

Third, we only expose a limited interface for floating-point numbers. The correctness proofs do not see the implementation of the floating-point numbers or their `real_like` class. As far as user code is concerned, `f32` could be defined as a pair `machine_f32 × erased real`, giving the same verification guarantees as verification with exact real numbers.

With the approximation class, the reduction function can now be specified as:

```
fn reduce #et { | real_like et | } (len:ℕ) (a:larray et len) (#s:seq et)
  preserves a ↦ s    requires pure (s ≈ 'r)
  returns  res : et   ensures  pure (res ≈ seq_fold_left 0.0 (+) 'r)
```

The verification of such a function will internally use the class methods to construct the proof that the result approximates the real sum. This extra precondition does not restrict the input: one can always choose `r` to be `map to_real s`, trivially satisfying the relation.

We use this to model the semantics of tensor core operations as well, as we discuss next.

4.3 Modeling Tensor Cores and Proving the Correctness of a 2D Blocktiling Matmul

Tensor cores are specialized hardware units on NVIDIA GPUs that perform small matrix multiplications (e.g., 16×16) very efficiently. These are exposed to CUDA through a warp-level intrinsics and for a matrix multiply-add (MMA), which computes $D \leftarrow A \times B + C$ for small matrices A , B , C , and D . The API requires loading small enough tiles of matrices from GPU or block-shared memory into dedicated "fragment" arrays (warp-level registers) using specific `load` and `fill` operations; then calling `mma_sync` to perform the MMA operation; and then finally using `store` functions to copy the results back from the fragments to global or shared memory. The operations require specifying the data types, the dimensions, and the layouts of the matrices involved, either row or column major for the input matrices, or a unspecified layout for an "accumulator" fragment.

We model this in KUIPER by introducing a type for fragments and fragment descriptors, ensuring that the operations are only called with well-typed fragments. The `fragment et` type is also equipped with a `points-to` relation, with a specification matrix suited to the layout of the fragment. The tensor core operations are available only for certain combinations of types and matrix dimensions, e.g., 16×16 matrices of half-precision floats (`f16`) for A and B , and half- or single-precision floats (`f32`) for C and D . By asserting $f \mapsto m$ for a fragment, we ensure that m is a mathematical matrix in an layout appropriate for f , and further that the appropriate combinations of types and dimensions are used. Since the dimensions of all three matrices are constrained together, the type of the fragment specifies all three dimensions (m, n, k) at once. For example, the operation to load a fragment in column-major layout is specified as follows:

```
fn mma_loadA_cm (fr : fragment et FragA m n k FragLCM) { | strided_col_major 1 | }
  (gm : gpu_matrix et 1)
preserves gm ↦ Frac f m0 requires fr ↦ f0 ensures fr ↦ m0
```

The main compute operation is `mma_sync`, which when called with fragments of different floating point types (usually C and D are of a larger type than A and B), will implicitly cast the elements of the matrices as needed. The tensor core documentation also does not specify the exact order in which the operations are performed, or where exactly the cast happens. This makes it impossible to provide a precise specification of the operation, especially one that accounts for floating point error. Our approximation technique comes in handy for describing such underspecified behavior.

An important limitation is that we do not model the *collective* warp aspect of the MMA operations. All threads in a warp must call the MMA operation together, and the behavior is undefined if some threads do not participate (e.g., if they call the `mma_sync` in a thread-specific control-dependent way). We do not yet model this aspect, though we expect we could adapt the same technique we use for barriers to enforce this. We leave this to future work.

Using Tensor Cores in a Verified 2D Block-tiled Matrix Multiplication. We have implemented and verified an algorithm that uses tensor cores to perform matrix multiplication, leveraging two levels of tiling and shared memory to achieve high performance. For the first level, tiles of size $bm \times bm$ are loaded from matrices A and B , respectively, into shared memory. This operation is performed collectively by all threads in each block, using barriers to synchronize access to shared memory. For the second level, every warp cooperates to compute a $wm * tm \times wm * tm$ subtile of the output tile. This subtile is computed via tensor core operations, which can perform fused MMA operations on small fragments of size $tm \times tm$, looping over the bm/tm fragments in the $bm \times bm$ tile, loading them into warp-local fragment registers, and calling the tensor core multiply-accumulate operations to compute the output fragments. Finally, the $wm * wm$ computed fragments are stored back to global memory. As expected, the matrix tiling API is heavily used in this implementation.

We have implemented this algorithm in KUIPER, verifying its correctness end-to-end. The implementation is general over the eight metaparameters mentioned above, and captures all required constraints between them. This implementation can later be instantiated to any concrete set of metaparameters satisfying the constraints. We report on the performance of this kernel in § 5.

4.4 Polymorphism, Parameter Selection & Extraction to CUDA

We’ve already shown polymorphism over scalar types (both exact and approximate), and matrix layouts. The beauty of dependently typed programming is that these are all instance of the same notion: dependent products. There is no distinction in KUIPER between type parameters, value parameters, or meta parameters. All of them can be abstracted over, and later instantiated when needed. This makes it trivial to provide kernels where the tuning parameters (say, the size of the shared memory tiles) are dynamic instead of static, though local arrays must have a statically-known size, or `nvcc` will reject it. Similarly, specializing the matrix sizes to a constant is also trivial.

Custom scalar operations. F^* ’s typeclasses are implemented via dictionary-passing through implicit arguments, and have no requirement of canonicity. Hence, by defining different dictionaries for the scalar constraint, we can also specialize GEMMs to use different operations over any given type. For instance, it is an easy exercise to define a type for integer distances in graphs (interpreting zero as infinity), defining addition and multiplication as minimum and addition respectively. Multiplying a distance matrix with itself computes the optimal two-step distance between nodes. Iterating this while repeatedly adding back the result computes all-pairs shortest paths.

Autotuning. A significant advantage of capturing all the constraints of metaparameters is that we know that every instantiation of the parameters that typechecks is valid. This enables *autotuning*, where we can generate many different instantiations of a kernel with different parameters, benchmark them, and pick the best-performing one for a given hardware and input size. While in

theory it should be possible to predict the best set of parameters, this is intractable in practice, and GPU programmers resort to this parameter search process routinely. Having confidence that all implementations in the search space are correct makes this process significantly easier.

A word on extraction. KUIPER uses F*'s existing extraction pipeline in order to eliminate all higher-order functions in the kernel (e.g., those inside the scalar typeclass, layouts, etc). We have also added some new passes to Karamel to perform more inlining and peephole optimizations. In particular, some of these optimizations are needed since nvcc (the NVIDIA CUDA compiler) misses some opportunities to optimize. One notorious example occurred in a function traversing a row-major matrix by rows via the layout framework. The generated C code had an array access of the form $A[(i/n)*n + (i\%n)]$. Even though C and C++ (and hence CUDA) guarantee the division algorithm identity (C standard, §6.5.5.6), this was not optimized away to i by nvcc. Adding a peephole optimization for this property increased the performance of some kernels significantly.

5 Evaluation

Using KUIPER, we have implemented several GPU kernels such as various flavors of GEMMs (a generalized variant of matrix multiplication), hierarchical parallel reductions, a stencil kernel, and the softmax function. All of these kernels have their functional correctness verified in KUIPER. We include these implementations in the supplementary material. Some of them, like simple variants of matrix multiplication and the stencil, have an exact functional specification, as the order in which they perform the floating point operations matches the mathematical definition. For the more sophisticated kernels like optimized GEMMs, hierarchical reduction, and softmax, the specification is approximate in the sense of § 4.2. While a fully-precise specification could be given for the latter two, it is a less compositional result and therefore less useful for client code.

In total, we have written around 36k lines of F* and Pulse code for the KUIPER DSL, its supporting libraries, and the GPU kernels. This generates around 31k lines of CUDA code after extraction, though this number is somewhat inflated by the several different specialized instantiations of the kernels (e.g. for different tile sizes and layouts). We show more detailed numbers in Table 1 for the main kernels¹ Nevertheless, it shows how KUIPER allows us to generate large amounts of verified CUDA code with a reasonable amount of effort, as the proof-to-code ratio is close to 1:1.

We also evaluate the performance of two matrix multiplication implementations: TensorCore2D and BlockTiling2D. The first one we described in § 4.3, while the second one is a conceptually similar implementation, but using regular scalar operations over the matrix elements instead of the tensor core intrinsics. Both of these algorithms are parametric over a large set of metaparameters, such as the dimensions of tiles on different hierarchy levels. For each GPU,

Kernel	F*/Pulse	CUDA
Matrix Multiplication (Naive)	386	41
Matrix Multiplication (BlockTiling2D)	930*	107
Matrix Multiplication (TensorCore2D)	2206	137
Tree Reduction	464	26
Dot product	277	40
Stencil	324	33
Softmax	224	68
Supporting Libraries	24,724	
Total	37,050	31,880

Table 1. Lines of code for verified GPU kernels in F*/Pulse and generated CUDA. The lines of CUDA are for a single instantiation of the relevant kernel, except for the total, which includes all our specialized kernels.

¹The proof for BlockTiling2D is not yet complete: only memory safety of the kernel function is proven. We plan to complete this proof in the very short term. It is conceptually simpler than that of TensorCore2D.

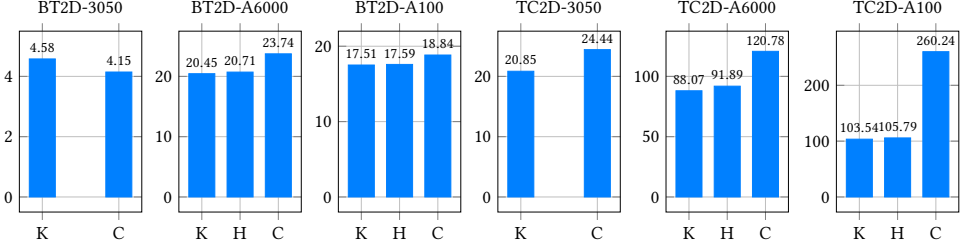


Fig. 4. Performance comparison of GEMMs with BlockTiling2D (BT2D) and TensorCore2D (TC2D), in TFLOPS (higher is better). The letters in the X axis stand for a **K**uiper kernel, compared against a **H**andwritten implementation of the same algorithm, and **C**uBLAS. The handwritten version uses exactly the same tuning parameters as the KUIPER version. We have no visibility into what cuBLAS does. Due to transient technical issues, we could not run the handwritten implementations on the RTX 3050.

we tune our implementation as described in § 4.4 to find the best performing configuration, and then benchmark the implementations over matrices of sizes 4096×4096 . We first compare the performance against the NVIDIA cuBLAS library, showing that these kernels are close to (but usually do not reach) the performance of this highly-optimized library. Not reaching the same performance is expected, as we have not (yet) implemented all the possible optimizations that cuBLAS has. However, being close shows that this approach is viable, and we are confident we can close the gap with more engineering effort. We also compare against a handwritten implementation of the same algorithm, with the same tuning parameters. This comparison is there to make sure that the particular shape of the KUIPER-generated code (which is usually non-idiomatic, particularly for tiled accesses) does not incur a performance penalty. We run all benchmarks on three different NVIDIA GPUs: an RTX 3050, an A6000, and an A100.

The results are shown in Figure 4. For BlockTiling2D, the non-tensor-core GEMM, the KUIPER kernel reaches between 80% and 110% of the performance of cuBLAS, and matches the handwritten implementation (within a few percent of noise). It is remarkable that the KUIPER implementation outperforms the cuBLAS implementation on the RTX 3050, though we do not know why (cuBLAS is closed source). One possible explanation is that cuBLAS is not usually meant for consumer GPUs, so its algorithms may not be as optimized for this hardware. For TensorCore2D, the performance is about 85% of cuBLAS on the RTX 3050, and 72% on the A6000. These gaps are larger, but within what we expect for our current level of optimizations. In the A100, the performance drops to about 40%. This is a signal that more complex features are required in order to fully take advantage of this hardware. One such missing feature is *asynchronous copies* from global to shared memory, which is supported in the A100 and can improve performance significantly. We plan to add support for this in the near future, and believe it should be straightforward. Nevertheless, we remark that KUIPER still performs within the noise of a handwritten implementation of the same algorithm, hence it does not introduce any significant overhead.

We remark that tuning is extremely important for performance. Figure 5 shows a glimpse of the performance distribution over the different configurations.

6 Related Work

GPU Verification Frameworks and Safe Languages. Given the difficulty of GPU programming and the importance of correctness, prior work has approached verification both by analyzing existing GPU code and by designing safe GPU languages.

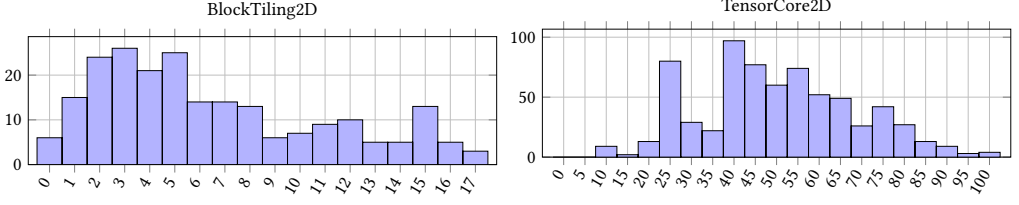


Fig. 5. Histogram of KUIPER kernel performance over tuning space. The X axis is performance in TFLOPS. The Y axis is the number of configurations that achieved that performance. For BlockTiling2D, the top-performant configuration (out of 288) is 3.14x faster than the median and 39.7x faster than the worst. For TensorCore2D there are 716 configurations tested, and the speedups are 1.96x and 9.23x respectively.

Tools such as PUG [Li and Gopalakrishnan 2010], GKLEE [Li et al. 2012], and GPUVerify [Betts et al. 2012] verify CUDA or OpenCL kernels, typically checking for errors such as data races. To scale to GPU-level parallelism, they rely on two insights: under a widely used notion of data-race freedom, only a single representative schedule must be explored, allowing sequentialization between synchronization barriers; and the behaviors of many threads can be captured by treating thread identifiers symbolically. Later extensions add limited reasoning about fine-grained sharing via RMW operations, either by selectively exploring relevant interleavings [Chiang et al. 2013] or conservatively overapproximating their effects [Bardsley and Donaldson 2014]. More recent tools, such as FAIAL [Liew et al. 2024], exploit common structural regularities in real kernels—simple data-parallel loops and predictable synchronization—to improve scalability and precision.

A complementary direction develops GPU languages with built-in safety guarantees. Futhark [Henriksen et al. 2017] enforces data-race freedom in a functional, data-parallel style while ensuring memory safety statically or via inserted checks. Descend [Köpcke et al. 2024] adopts Rust-like principles to provide low-level control with static guarantees of memory safety and data-race freedom, while permitting isolated unsafe regions. Higher-level GPU languages also offer partial correctness guarantees through input restrictions, including scheduling languages such as Halide [Ragan-Kelley et al. 2013] and TVM [Chen et al. 2018], and meta-languages like Exo [Ikarashi et al. 2025] that incorporate custom operations once validated for safety.

Verification efforts further depend on precise GPU semantics, yet many aspects of the model remain underspecified. Early work on memory consistency exposed misunderstandings about ordering guarantees [Alglave et al. 2015] and led to official formalizations [Lustig et al. 2019] later incorporated into a model-checking tool [Tong et al. 2025]. Other semantic questions remain open, such as forward progress between thread blocks (particularly on non-NVIDIA platforms) [Sorensen et al. 2021] and numerical properties of tensor core execution [Valpey et al. 2025].

KUIPER draws heavily on the extensive literature on concurrent separation logic for its program logic, with PulseCore [Ebner et al. 2025] itself drawing heavily on Iris [Jung et al. 2018b], and of course on the classic works of Reynolds [2002] and O’Hearn [2007]. There is limited work on integrating separation logic with GPU programming: VerCors supports OpenCL as a frontend language [Amighi et al. 2015; Blom et al. 2014], modeling barriers and blocks. Their logic also has a distinction between block-local memory and global memory like KUIPER, but they rely on syntactic checks of the source program, while we give a semantic model. Asakura et al. [2016] give an operational semantics and separation logic for a small GPU language and prove its correctness in Coq; their proof rule for kernel launches and barrier specification closely resembles ours, however they do not model blocks, shared memory, or distinctions between CPU and GPU global memory. Compared to existing approaches KUIPER enjoys the high expressivity of its host language F^* ,

supporting dependent types, monomorphization, and higher-order combinators. On the practical side, we prove the effectiveness and scalability of KUIPER by developing large, competitive kernels in it. We also focus on modeling bespoke hardware features such as tensor cores to achieve maximum performances, instead of proving extensive results about an idealized subset.

7 Conclusions

GPU programming desperately needs better, safe abstractions to manage its complexity, but these abstractions cannot come at the cost of performance, since the whole endeavor of GPU programming is to extract maximum performance from the hardware. Further, given the complexity of GPU programming, functional correctness is also desirable, particularly as testing some kernels can be challenging (e.g. in machine learning).

We’ve presented KUIPER, a safe CPU/GPU programming language which captures the intricacies of GPU programming in a dependently-typed concurrent separation logic. KUIPER programs are written from the point of view of a single thread, just like CUDA, and advanced features that sometimes elude other languages can be expressed in KUIPER in a modular way. While the thread-centric view allows the use of low-level features, abstractions (such as the matrix library) allow writing generic high-level code that is easier to verify. We have used KUIPER to write several realistic kernels, with a proof of their functional correctness, in a highly generic style, and with reasonable performance, showing that this approach is viable in practice. We believe KUIPER is a significant step forward in the quest for safe and efficient GPU programming.

References

- J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. [GPU concurrency: Weak behaviours and programming assumptions](#). In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015.
- A. Amighi, S. Darabi, S. Blom, and M. Huisman. [Specification and verification of atomic operations in GPGPU programs](#). In R. Calinescu and B. Rumpe, editors, *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*. 2015.
- J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, and S. Chintala. [Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation](#). In *ASPLOS ’24*.
- I. Asakura, H. Masuhara, and T. Aotani. [Proof of soundness of concurrent separation logic for GPGPU in coq](#). *J. Inf. Process.*, 24(1):132–140, 2016.
- E. Bardsley and A. F. Donaldson. [Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels](#). In *Proceedings of the 6th International Symposium on NASA Formal Methods - Volume 8430*. 2014.
- A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. [GPUVerify: a verifier for GPU kernels](#). In *OOPSLA ’12*. 2012.
- S. Blom, M. Huisman, and M. Mihelcic. [Specification and verification of GPGPU programs](#). *Sci. Comput. Program.*, 95: 376–388, 2014.
- T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, 2018.
- W.-F. Chiang, G. Gopalakrishnan, G. Li, and Z. Rakamarić. Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In *NASA Formal Methods*. 2013.
- G. Ebner, G. Martínez, A. Rastogi, T. Dardinier, M. Frisella, T. Ramananandro, and N. Swamy. [PulseCore: An impredicative concurrent separation logic for dependently typed programs](#). *Proc. ACM Program. Lang.*, 9(PLDI):1516–1539, 2025.
- ggml.org. llama-cpp. <https://github.com/ggml-org/llama.cpp>.
- B. Hagedorn, J. Lenfers, T. Koehler, X. Qin, S. Gorlatch, and M. Steuwer. [Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies](#). *Proc. ACM Program. Lang.*, 4(ICFP): 92:1–92:29, 2020.

- T. Henriksen, N. G. W. Serup, M. Elsmann, F. Henglein, and C. E. Oancea. [Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates](#). In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017.
- A. Hobor and C. Gherghina. Barriers in concurrent separation logic. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*. 2011.
- Y. Ikarashi, K. Qian, S. Droubi, A. Reinking, G. L. Bernstein, and J. Ragan-Kelley. [Exo 2: Growing a scheduling language](#). In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 2025.
- R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer. [RustBelt: securing the foundations of the Rust programming language](#). *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018a.
- R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. [Iris from the ground up: A modular foundation for higher-order concurrent separation logic](#). *J. Funct. Program.*, 28:e20, 2018b.
- B. Köpcke, S. Gorlatch, and M. Steuwer. [Descend: A safe gpu systems programming language](#). *Proc. ACM Program. Lang.*, 8(PLDI), 2024.
- G. Li and G. Gopalakrishnan. [Scalable SMT-based verification of GPU kernel functions](#). In *FSE*. 2010.
- G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. [GKLEE: concolic verification and test generation for GPUs](#). *SIGPLAN Not.*, 2012.
- D. Liew, T. Cogumbreiro, and J. Lange. [Sound and partially-complete static analysis of data-races in GPU programs](#). *Proc. ACM Program. Lang.*, (OOPSLA2), 2024.
- D. Lustig, S. Sahasrabudhe, and O. Giroux. [A formal analysis of the NVIDIA PTX memory consistency model](#). In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019.
- Nvidia. Getting started with CuTe. https://docs.nvidia.com/cutlass/media/docs/cpp/cute/00_quickstart.html. Accessed: 2025-09-10.
- Nvidia. [Nvidia cutlass](#), 2025.
- P. W. O'Hearn. [Resources, concurrency, and local reasoning](#). *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
- J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hrițcu, K. Bhargavan, C. Fournet, and N. Swamy. [Verified low-level programming embedded in F*](#). *PACMPL*, 1(ICFP):17:1–17:29, 2017.
- J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. [Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines](#). In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.
- J. C. Reynolds. [Separation logic: A logic for shared mutable data structures](#). *LICS*. 2002.
- T. Sorensen, L. F. Salvador, H. Raval, H. Evrard, J. Wickerson, M. Martonosi, and A. F. Donaldson. [Specifying and testing GPU workgroup progress models](#). *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021.
- M. Steuwer, T. Rummel, and C. Dubach. [LIFT: A functional data-parallel IR for high-performance GPU code generation](#). In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017.
- N. Swamy, C. Hrițcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. [Dependent types and multi-monadic effects in F*](#). *POPL*. 2016.
- P. Tillet, H. T. Kung, and D. Cox. [Triton: An intermediate language and compiler for tiled neural network computations](#), 2021.
- H. Tong, N. Gavrilenco, H. Ponce de Leon, and K. Heljanko. [Towards unified analysis of GPU consistency](#). In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 2025.
- B. Valpey, X. Li, S. Pai, and G. Gopalakrishnan. [An SMT formalization of mixed-precision matrix multiplication](#). In *NASA Formal Methods*. 2025.