Kuiper: verified and efficient GPU programming

Anonymous Author(s)

Abstract

Kuiper is a language for safe and verified GPU programming based on dependent types and concurrent separation logic. Kuiper models the intricacies of the GPU programming model, including the memory hierarchy, kernel launches, and synchronization within a single comprehensive framework. Memory bugs and data races are ruled out for any well-typed program. Optionally, full functional-correctness proofs can be performed. Kuiper programs can be heavily polymorphic (over types, operations, memory layout, and more) and extract to efficient, specialized CUDA code. We use Kuiper to implement several common GPU kernels (matrix multiplication, reductions, softmax). Kuiper is built over the F* [Swamy et al. 2016] ecosystem.

GPU kernels are delicate pieces of code, where possibilities for errors (memory bugs, data races, synchronization bugs, etc) abound. At the same time, performance is paramount, and compromises between performance and ease of programming will normally favor performance. There is a clear need for safe, performant languages in this space.

Kuiper is a language aiming to provide both. It is embedded in Pulse, a dependently-typed programming language based on concurrent separation logic [Ebner et al. 2025], allowing a high-degree of expresiveness and verification. All Kuiper programs are free of memory bugs and data races by construction. Kuiper also allows the programmer to write heavily-optimized code, not imposing any significant constraints over CUDA. Optionally, Kuiper programs can be verified for functional correctness.

Kuiper is a work in progress.

1 Kuiper Basics

Kuiper is built on top of Pulse, which is a dependently-typed programming language based on concurrent separation logic [Ebner et al. 2025]. Program specifications include pre- and postconditions stating precise ownership of resources. For instance, the following program copies a reference s into another one t:

```
fn copy (t s : ref α)
  preserves s ↦ Frac 'f 'vs
  requires t ↦ 'vt
  ensures  t ↦ 'vs {
  let v = !s;
  t ⟵ v;
}
```

The function has several annotation clauses: **preserves**, **requires**, and **ensures**. Multiple clauses of the same kind may be provided, in which case they are combined using $\star$, the separating conjunction. The **preserves** clause is just a shortcut for resources that appear unchanged in both the pre- and postcondition. The $\mapsto$ symbols read "points to" states that a given reference is live and set to some particular value. The precondition states that this functions requires ownership over (some fraction of) s and p (fully). Requiring only a fraction of "s" allows other threads to own other fractions of "s" at the same time, and allows for read-only access to it. The postcondition states that the value pointed to by s has remained unchanged, and that ‘t‘ now points to the same value as ‘s‘.

1.1 Modes for distinguishing CPU and GPU code

To model device and host functions, Kuiper introduces "mode resources". Every host function (e.g. the main function) requires a **cpu** resource, and every device function (e.g. kernels and functions called by them) requires a **gpu** resource. Both of these resources are kept abstract, so nothing can be done with them, a priori, except for passing them around. Every function returns their mode resource back to the caller. Kuiper can be used both to write whole programs and mixed CPU/GPU libraries. In the first case, a program starts owning the **cpu** resource.

```
val cpu : slprop (* abstract *)
val gpu : slprop (* abstract *)
```

Here is the Kuiper signature of a function that allocates an uninitialized array in GPU memory. It can only be called from the host.

```
fn gpu_array_alloc0
  (t : Type) {| sized t |} (len : nat)
  preserves cpu
  returns a : gpu_array t len
  ensures ∃* s. ga ↦ s
```

The **preserves** here indicates that this function may only be called when "owning" the CPU mode, which is returned back to the caller. The function expects an element type t and a length len. The {| sized t |} fragment is a typeclass constraint: arrays can only be created for types for which Kuiper statically knows the size, to request the proper amount of bytes to cudaMalloc. The function returns a new gpu_array of the proper type and length. The postcondition states that this array points to something, but makes no further guarantees about the contents. Kuiper provides types for device memory data (gpu_ref, gpu_array, etc) in tandem with the native Pulse counterparts (ref, array).

1.2 Kernel Descriptions and Kernel Calls

So far, the use of **cpu** and **gpu** mode resources prevents any kind of calls between different modes. Kuiper provides functions that allow calling ("launching") kernels from CPU code, provided the proper conditions are met. The simplest version is the following, which spawns one single thread in the GPU executing some function k.

```
fn launch_kernel₁
  (k : unit → unit (requires gpu ★ 'pre)
                   (ensures  gpu ★ 'post))
  preserves cpu  requires 'pre  ensures 'post
```

The $\texttt{launch\_kernel}_1$ function can be called on any GPU function, provided it returns unit. The caller needs to provide k's precondition, and obtains back k's postcondition. The **preserves** guarantees this function can only be called from the CPU, where the call can be properly configured (in the case of CUDA, this would be something like k<<<1,1>>>).

For more realistic, parallel kernel calls, Kuiper provides a record type for kernel descriptions (we elide the definition). A value of this type specifies the kernel function to run, the grid configuration, and any intermediate pre- and postconditions.

```
type kernel_desc (pre post : slprop) : Type
```

Once a record of this type is constructed, it can be launched simply by calling the function below.

```
fn launch_kernel (k : kernel_desc 'pre 'post)
  preserves cpu  requires 'pre  ensures 'post
```

Since the grid configuration is baked into the record, mismatches at the call are impossible.

For a given functionality in mind (e.g. multiplying two matrices A and B into C), there will be a function to create a kernel_desc that is specialized to exactly this taks which can then be launched as-is, without providing extra arguments. Roughly:

```
val mk_matmul_kdesc
  (a : gpu_matrix f32 m k)
  (b : gpu_matrix f32 k n)
  (c : gpu_matrix f32 m n)
  (#va #vb #vc : _)
  : kernel_desc (a ↦ va ★ b ↦ vb ★ c ↦ vc)
                (a ↦ va ★ b ↦ vb ★ c ↦ va × vb)
```

where f32 represent the float type, and $\times$ is the (mathematical, specification-level) matrix multiplication. We have implemented several (verified) versions of such multiplcations. We elide discussing them for size constraints, and instead focus on polymorphism and specialization aspects.

This kernel can then be launched as-is, without specifying a grid as it as already baked into the description. The code generation process will partially evaluate the program, removing all these indirections and ending up with a simple kernel call, without overhead.

1.3 Polymorphism: values, operations, layouts

Kuiper inherits from F★ and Pulse the support for polymorphism and typeclasses. The matrix multiplication above can be defined generically over any "scalar" type. The scalar typeclass is defined as such:

```
class scalar (t : Type) = {
  [@@@superclass] is_sized : sized t;
  zero : t;              one : t;
  add  : t → t → t;    mul : t → t → t;
}
```

That is, a scalar type is one that has a staticly-known size, a distinguished zero and one, and supports addition, multiplication. Notably we do not require any properties over these operations.

Also, one is usually interested in GEMMs (computing $C \leftarrow \alpha AB + \beta C$ for some scalars α and β). To not duplicate this implementation, we instead generalize it over some comb function that indicates how to combine the previous value in c with the result of the multiplication. Choosing comb old new = new gives the standard matrix multiplication, while comb old new = alpha * new + beta * old gives a GEMM.

As a further generalization, we can also abstract over the in-memory representation of the matrices. The algorithm is independent of concrete positions in memory and only accesses individual cells of the matrix. We call *layout* the way in which matrix cells map into the concrete array positions (essentially a bijection between $\mathbb{N}_M \times \mathbb{N}_N$ and $\mathbb{N}_{M\times N}$). The layouts encode how to index into the underlying array, essentially providing accessor and setter functions. Given any layout for each matrix of the matrices (which can be different), the algorithms works the same, so we abstract over it too.

The most generic type for this matmul becomes:

```
val mk_matmul_kdesc
  (#et : Type) {| scalar et |}
  (#la : mlayout et 'm 'k)
  (#lb : mlayout et 'k 'n)
  (#lc : mlayout et 'n 'k)
  (comb : old:et → net:et → et)
  (a : gpu_matrix et la)
  (b : gpu_matrix et lb)
  (c : gpu_matrix et lc)
  (#va #vb #vc : _)
  : kernel_desc (a ↦ va ★ b ↦ vb ★ c ↦ vc)
                (a ↦ va ★ b ↦ vb ★ c ↦ va × vb)
```

This kernel can be specialized to any scalar type, any comb function, and any combination of layouts (e.g. row-major inputs and column-major output). All the abstraction inlines away and the generated code is fully specialized, without any induced overhead. The verification is performed only once at the most general level.

References

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. 2025. To appear in PLDI 2025.

N. Swamy, C. Hriţcu, 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.