Taming GPU programming with safe Rust

Manuel S. Drehwald
LLVM Developers' Meeting, 2025

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-PRES-2012993.

Collaborators

Manuel Drehwald
University of Toronto / LLNL

Marcelo Domínguez
Universidad Rey Juan Carlos

Kevin Sala
LLNL

Johannes Doerfert
LLNL

Challenges for safe GPU programming

- Every reference in Rust has alias guarantees[(1)]:

```rust
fn foo(x: &f64, y: &mut f64)
```

```llvm
define void @foo(ptr noalias noundef readonly align 8 captures(none) dereferencable(8) %x,
                 ptr noalias noundef align 8 captures(none) dereferencable(8) %y) {
```

(1) Excluding references to an UnsafeCell)

Challenges for safe GPU programming

- Every reference in Rust has alias guarantees[(1)]:

```rust
fn foo(x: &f64, y: &mut f64)
```

```
define void @foo(ptr noalias noundef readonly align 8 captures(none) dereferencable(8) %x,
                 ptr noalias noundef align 8 captures(none) dereferencable(8) %y) {
```

- Raw pointers have no alias guarantees, not even strict-aliasing (unlike C++):

```rust
unsafe fn bar(x: *const f64, y: *mut f64)
```

```
define void @bar(ptr noundef readonly captures(none) %x,
                 ptr noundef captures(none) %y) {
```

(1) Excluding references to an UnsafeCell)

Challenges for safe GPU programming

A CUDA vector addition[1]

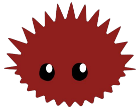
```
__global__ void vectorAdd(const float *A,
        const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements) {
        C[i] = A[i] + B[i] + 0.0f;
    }
}
```

Now let's imagine it in safe Rust

```
fn vector_add(A: &[f32], B: &[f32], C: &mut [f32]) {
    let idx: usize = unsafe {_thread_idx_x()} as usize;
    if idx < C.len() {
        C[idx] = A[idx] + B[idx];
    }
}
```

Data can be immutably borrowed any number of times [..]
On the other hand, only *one* mutable borrow is allowed at a time.[2]

Assuming > 1 thread, C causes UB!



(1) https://github.com/NVIDIA/cuda-samples

(2) https://doc.rust-lang.org/rust-by-example/scope/borrow/alias.html

Current GPU infrastructure in rustc

- **rustc –print target-list:**
  - nvptx64-nvidia-cuda (Tier 2 without Host Tools, “it compiles”),
  - amdgcn-amd-amdhsa (Tier 3, “it exists”)

- ***gpu-kernel*** ABI: lowered to ***ptx_kernel*** or ***amdgpu_kernel*** based on target

- **core::arch::nvptx:** A wrapper around ~30 basic, Nvidia specific gpu intrinsics.

Our wishlist for a gpu feature

- Safe & convenient by default. This includes automatic memory transfer
- Allow unsafe escape hatch for better control or performance
- Support "almost all" Rust functions and types. (see KernelAbstractions.jl)
- Support multiple vendors

Why offload?

- Able to support multiple vendors
- Already tested via OpenMP in C++ & Fortran
- Already provides helpful abstractions, but also supports “native” types.
- LLVM based, it works with std::autodiff (Enzyme), gpu-libc, and others
- Tested Rust support for AMD & NVIDIA (as of 30 minutes ago)

- Only AMD & NVIDIA support (More to come?)

Our current interface

- A CPU function called via our offload intrinsic
- Our intrinsic generates the needed calls to LLVM’s offload library
- Hard-coded Kernel Dimensions (for now)
- Single codebase compiled twice, for Host and Device
- Enable support for wrappers around libraries like CuBLAS

```
pub fn kernel(x: &mut [f32; 256]) {
    core::intrinsics::offload(_kernel, x)
}
```

```
pub fn _kernel(x: &mut [f32; 256]) {
    for i in 0..256 { usize
        x[i] = 21.0;
    }
}
```

Our compilation pipeline

- cargo +offload build -r -v // Compile for the host
- rustc +offload src/lib.rs -C lto=fat [..]--emit=llvm-bc,llvm-ir -Zoffload=Enable -Zunstable-options
- RUSTFLAGS="-Ctarget-cpu=gfx90a --emit=llvm-bc,llvm-ir" cargo +offload build -Zunstable-options -r -v --target amdgcn-amd-amdhsa -Zbuild-std=core // Compile for the device
- clang-offload-packager -o host.out --image=file=device.bc,triple=amdgcn-amd-amdhsa,arch=gfx90a,kind=openmp
- [...]
- clang-linker-wrapper --should-extract=gfx90a --device-compiler=amdgcn-amd-amdhsa=-g --device-linker=amdgcn-amd-amdhsa=-lompdevice --host-triple=x86_64-unknown-linux-gnu

Looking at the IR

Each `core::intrinsics::offload` invocation currently generates three calls:

```
call void @__tgt_target_data_begin_mapper(ptr @1, i64 -1, i32 1, ptr %5, ptr %6, ptr %7, ptr @.offload_maptypes.1, ptr null, ptr null)
; [..]
%21 = call i32 @__tgt_target_kernel(ptr @1, i64 -1, i32 2097152, i32 256, ptr @.kernel_1.region_id, ptr %kernel_args)
; [..]
call void @__tgt_target_data_end_mapper(ptr @1, i64 -1, i32 1, ptr %22, ptr %23, ptr %24, ptr @.offload_maptypes.1, ptr null, ptr null)
```

Tackling the safety challenge

Looking again at our unsound example

```
fn vector_add(A: &[f32], B: &[f32], C: &mut [f32]) {
    let idx: usize = unsafe {_thread_idx_x()} as usize;
    if idx < C.len() {
        C[idx] = A[idx] + B[idx];
    }
}
```

We can avoid such overlapping slices by going through their raw parts (no-op).

Tackling the safety challenge

Looking again at our unsound example

```
fn vector_add(A: &[f32], B: &[f32], C: &mut [f32]) {
    let idx: usize = unsafe {_thread_idx_x()} as usize;
    if idx < C.len() {
        C[idx] = A[idx] + B[idx];
    }
}
```

We can avoid such overlapping slices by going through their raw parts (no-op).

But now in sound Rust code

```
fn vector_add2(A: &[f32], B: &[f32],
               C: *mut f32, Csize: usize) {
    let idx: usize = unsafe {_thread_idx_x()} as usize;
    if idx >= Csize {
        return;
    }
    let C: &mut f32 = unsafe {
        &mut *C.add(idx)
    };
    *C = A[idx] + B[idx];
```

We "pre-divide" mutable output slices for users.

```
fn vector_add_batched(A: &[f32], B: &[f32],
               C: *mut f32, Csize: usize) {
    let idx: usize = unsafe {_thread_idx_x()} as usize;
    let chunk_size: usize = Csize / _block_dim_x();
    let c_offset: usize = idx * chunk_size;
    let C: &mut[f32] = unsafe {
        core::slice::from_raw_parts_mut::<f32>(
            C.add(c_offset), chunk_size,
        )
    };
    // Use C
}
```

Tackling the safety challenge

Looking again at our unsound example

```
fn vector_add(A: &[f32], B: &[f32], C: &mut [f32]) {
    let idx: usize = unsafe {_thread_idx_x()} as usize;
    if idx < C.len() {
        C[idx] = A[idx] + B[idx];
    }
}
```

We can avoid such overlapping slices by going through their raw parts (no-op).

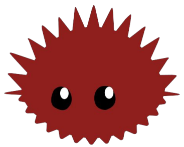
But now in sound Rust code

```
fn vector_add2(A: &[f32], B: &[f32],
               C: *mut f32, Csize: usize) {
    let idx: usize = unsafe {_thread_idx_x()} as usize;
    if idx >= Csize {
        return;
    }
    let C: &mut f32 = unsafe {
        &mut *C.add(idx)
    };
    *C = A[idx] + B[idx];
```

Can be auto-generated

We "pre-divide" mutable output slices for users.

```
fn vector_add_batched(A: &[f32], B: &[f32],
                      C: *mut f32, Csize: usize) {
    let idx: usize = unsafe {_thread_idx_x()} as usize;
    let chunk_size: usize = Csize / _block_dim_x();
    let c_offset: usize = idx * chunk_size;
    let C: &mut[f32] = unsafe {
        core::slice::from_raw_parts_mut::<f32>(
            C.add(c_offset), chunk_size,
        )
    };
    // Use C
}
```



Tackling the safety challenge

- The *"right way"* to split a mutable argument depends on the type and context
- We won't predict all ways, but ~60% of the RajaPerf benchmarks we looked at follow simple index patterns which we can cover.

- We provide a safe set of options (scalar, batched, …) where possible
- We provide an unsafe interface for user provided splitting otherwise

- By separating the (safe) Kernel code from the unsafe splitting, we can introduce safe abstractions

Looking at RajaPerf Benchmarks

Scalar output(s)	Shuffled scalar outputs	Advanced index patterns
VOL3D	MATVEC_3D_STENCIL	MASS3DPA
NODAL_ACCUMULATION_3D (including atomics)	DEL_DOT_VEC_2D	MASS3DEA
FIR	ZONAL_ACCUMULATION_3D	LTIMES
ENERGY		LTIMES_NOVIEW

With minimal or no adjustments, we should be able to cover 4 (7) of the 11 benchmarks.

A design for advanced output indexing

// SAFETY: For two different thread indices, a helper
// may not return the same reference (injective)

```
// MATVEC_3D_STENCIL reference implementation:

for (Index_type ii = ibegin; ii < iend; ++ii ) {
  Index_type i = real_zones[ii];

  b[i] = dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + …
}
```

```
// LTIMES_NOVIEW reference implementation:

for (Index_type z = 0; z < num_z; ++z ) {
  for (Index_type g = 0; g < num_g; ++g ) {
    for (Index_type m = 0; m < num_m; ++m ) {
      for (Index_type d = 0; d < num_d; ++d ) {

        phi[m+ (g * num_m) + (z * num_m * num_g)] +=
          ell[d+ (m * num_d)] * …;
```

A simple shuffle of indices (left) vs. multi-dimensional indexing (right)

https://github.com/LLNL/RAJAPerf

Supporting unsafe Rust types

- All 3 mayor Rust linear algebra libraries (faer, nalgebra, ndarray) use raw pointers for matrix types.
- Can we *really* not support automatic data movement for them?

```
pub struct ArrayBase<S, D>
where S: RawData
{
    /// Data buffer / ownership information. (If owned, contains the data
    /// buffer; if borrowed, contains the lifetime and mutability.)
    data: S,
    /// A non-null pointer into the buffer held by `data`; may point anywhere
    /// in its range. If `S: Data`, this pointer must be aligned.
    ptr: std::ptr::NonNull<S::Elem>,
    /// The lengths of the axes.
    dim: D,
    /// The element count stride per axis. To be parsed as `isize`.
    strides: D,
}
```

https://docs.rs/ndarray/0.16.1/src/ndarray/lib.rs.html#1280-1293

Supporting unsafe Rust types

- Many unsafe types implement the *clone* trait:
  A common trait that allows explicit creation of a duplicate value.[1]

```
pub trait Clone: Sized {
    // Required method
    fn clone(&self) -> Self;

    // Provided method
    fn clone_from(&mut self, source: &Self) { ... }
}
```

What if we just replace every `memcpy` with a `CopyHostToDevice`?

(1) https://doc.rust-lang.org/std/clone/trait.Clone.html

Optimizations (the fun part)

1. Only copy in the needed direction (const slices are not copied back)
2. Allocate a variable directly on the device, when possible.
3. Leave data on the device between kernels if unchanged/unused
4. Shared memory
5. Fusing kernels

Summary

- We need some unsafe code for splitting args, but look for safe abstractions

- We can handle 4/11 Benchmarks safely by hiding unsafety in the compiler
- We can handle additional 3/11 Benchmarks with trivial unsafe code.
- We need more advanced indexing logic for 4/11 Benchmarks

- We hope for more safe wrappers shared through user crates (libraries)

Questions?