GPU Offload in Rust: Available, Portable and Fast

Marcelo Domínguez
2026 European LLVM Developers' Meeting

Collaborators

Manuel Drehwald
University of Toronto / LLNL

Marcelo Domínguez
Universidad Rey Juan Carlos

Kevin Sala
LLNL

Johannes Doerfert
LLNL

Use Case A: Write GPU code in pure Rust

Built on top of LLVM Offload, which works on several different architectures such as NVPTX, AMDGPU or x86_64.

This makes Rust Offload portable!

```rust
pub unsafe extern "gpu-kernel" fn vector_add(A: &[f32], B: &[f32], C: &mut [f32]) {
  let i = unsafe { thread_idx_x() as usize };

  if let i < C.len() {
    C[i] = A[i] + B[i];
  }
}

fn main() {
  ...
  core::intrinsics::offload(
    vector_add,              // kernel
    [256, 1, 1],             // grid dims
    [32, 1, 1],              // workgroup dims
    (&A, &B, &mut C),        // kernel args
  );
}
```

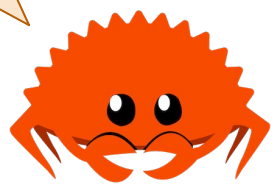
Use Case B: Call GPU vendor libraries from the Host

```
pub fn rocblas_sgemv_wrapper(A: &[f32; 6], x: &[f32; 3], y: &mut [f32; 2]) -> () {
  ...
  let trans = rocblas_operation::rocblas_operation_none;
  let mut handle: rocblas_handle = core::ptr::null_mut();
  let st_res = rocblas_sgemv(
    handle,
    trans,
    ...
  );
  ...
}

fn main {
  ...
  core::intrinsics::offload_args(
    rocblas_sgemv_wrapper,
    [256, 1, 1],
    [32, 1, 1],
    (&A, &x, &mut y),
  );
}
```

Adding a new offload_args intrinsic, which only maps arguments #150683

PR under review



Benchmarking: RAJA Performance Suite

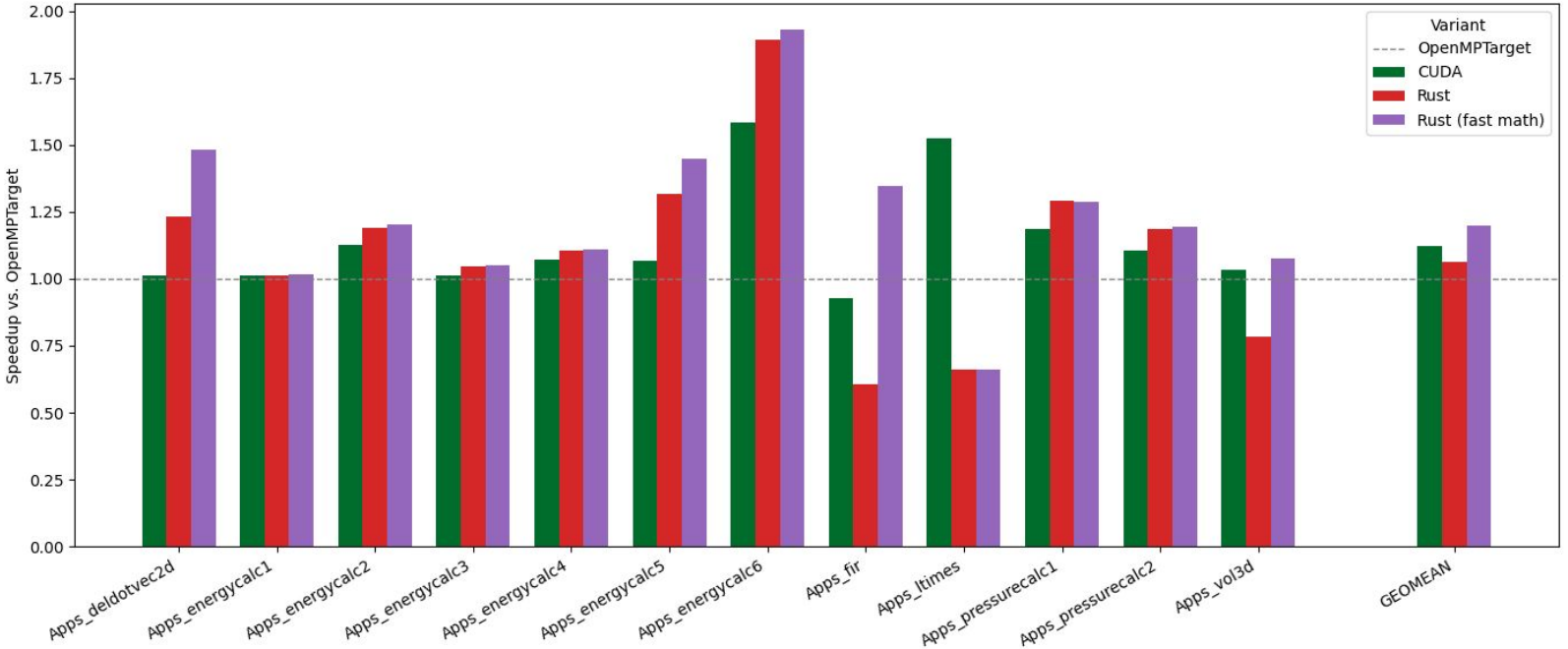
The RAJA Performance Suite provides C++ loop-based kernels from real scientific computing applications.

Implemented kernels in Rust, mimicking the original C++ logic for one-to-one comparison.

Hardware: 16-core, 384 GB RAM, RTX 2070 GPU

Toolchain: `clang` and `rustc` based on LLVM 21.1.18

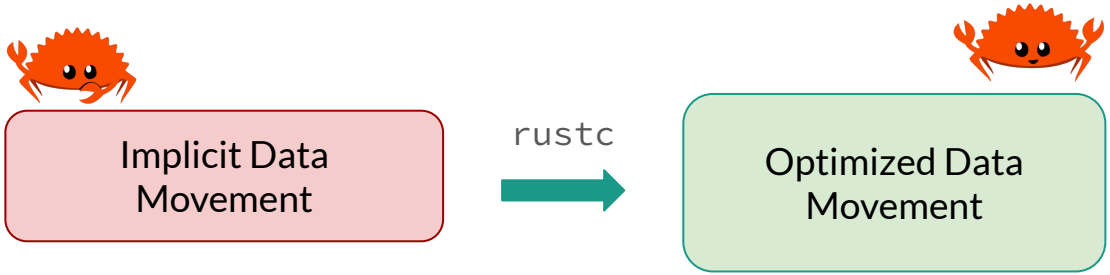
RAJAPerf Benchmarks: raw kernel times



Zero-Cost abstractions for safe Rust

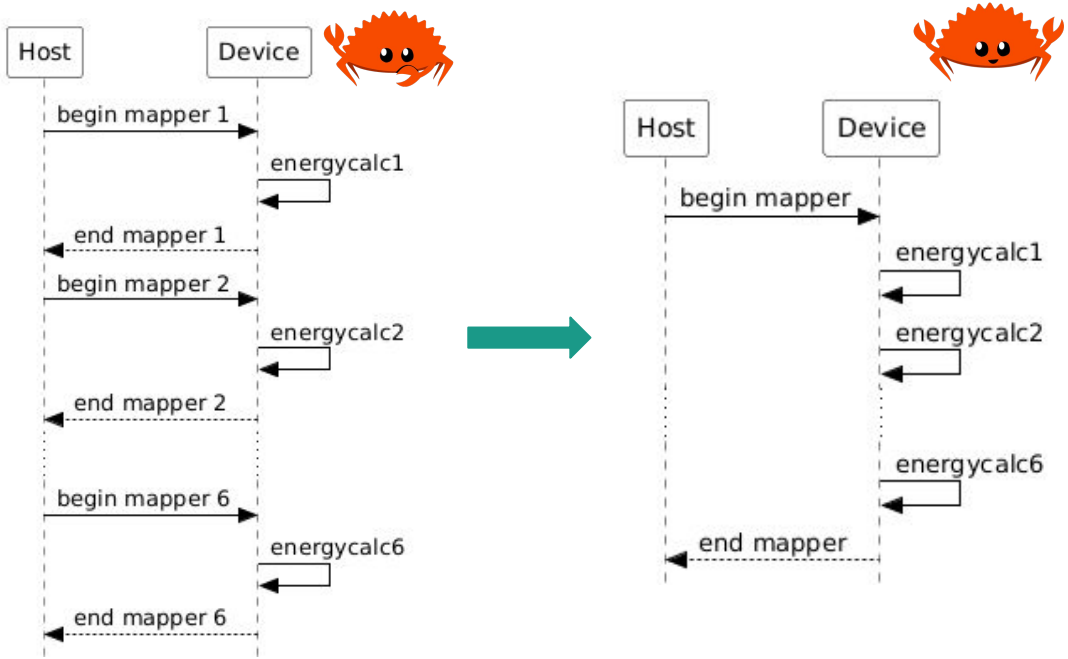
Automatic data movement won't be as "smart" as manual optimization.

1. Add manual data movement
2. Compiler-level optimizations to cover most common cases.



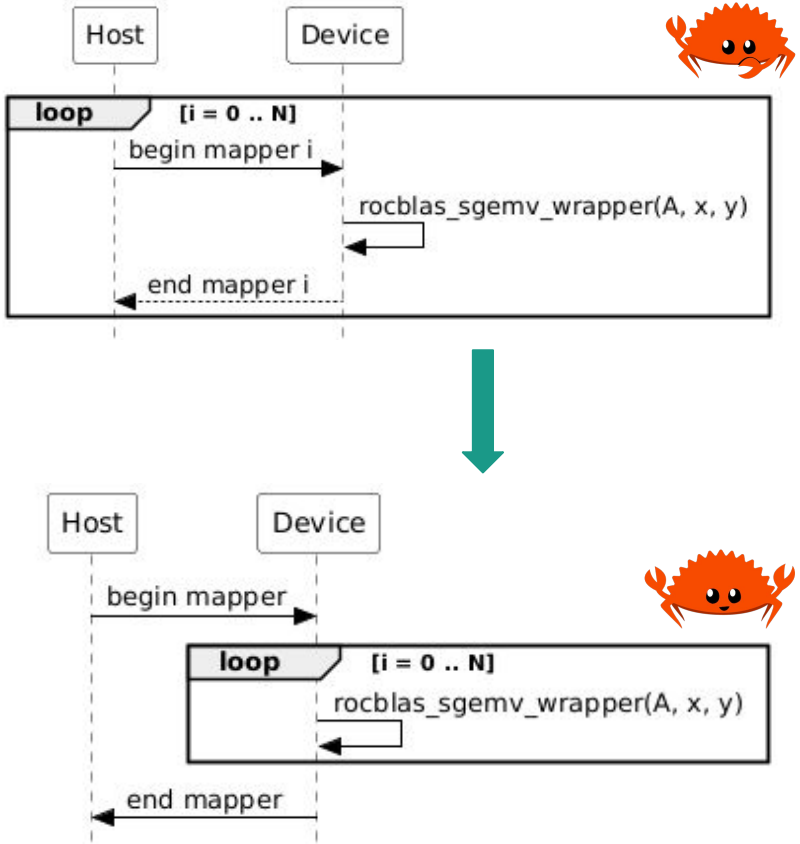
Fuse memtransfers

```
// RAJAPerf Apps_ENERGY
fn main() {
  // begin mapper 1
  offload!(energycalc1(...))
  // end mapper 1
  // begin mapper 2
  offload!(energycalc2(...))
  // end mapper 2
  ...
  // begin mapper 6
  offload!(energycalc6(...))
  // end mapper 6
}
```



Hoist memtransfers

```
fn main() {
  for i in 0..N {
    // begin mapper i
    offload_args(
        rocblas_sgemv_wrapper,
        (&A, &x, &mut y)
    );
    // end mapper i
  }
}
```



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).

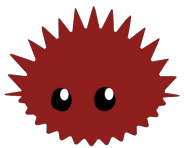
But now in sound Rust code

```
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
}
```



Optimizing `unsafe` code

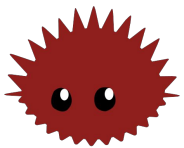
Memtransfers require consistency, reordering is only safe with `noalias`.

Cannot optimize in presence of raw pointers and `UnsafeCell`
`only with LLVM info`

Rust's aliasing models: define rules for how references interact and overlap in memory.

Use high-level info to guarantee `noalias` safely.

Stacked Borrows[1] & Tree Borrows[2]



(1) Jung et al. Stacked Borrows: An Aliasing Model for Rust. POPL 2019.
(2) Villani et al. Tree Borrows. PLDI 2025.

Data Movement Orchestration via TB Analysis

```rust
unsafe fn fuse(ptr: *mut i32) {
  gpu_foo(ptr);
  let val = *ptr;
  gpu_foo(ptr);

  println!("{*ptr}");
}

unsafe fn gpu_foo(p: *mut i32) {
  *p = 42;
}
```

We can leave `ptr` on the device between kernel calls

```rust
unsafe fn fuse_ub(ptr: *mut i32) {
  gpu_foo(ptr);
  let val = *ptr;
  gpu_foo(ptr);

  println!("{val}");    // ! UB detected
}

unsafe fn gpu_foo(p: *mut i32) {
  *p = 42;
}
```

We must move `ptr` back to host after the first kernel call

Summary

Upsteam Rust: Simple & Portable Offloading

Use Case A: Write GPU code in pure Rust

Built on top of LLVM Offload, which works on several different architectures such as NVPTX, AMDGPU.

This makes Rust Offload portable!

```rust
pub unsafe extern "gpu-kernel" fn vector_add(A: &[f32], B: &[f32], C: &mut [f32]) {
  let i = unsafe { thread_idx_x() as usize };

  if let i < C.len() {
    C[i] = A[i] + B[i];
  }
}

fn main() {
  ...
  core::intrinsics::offload(
    vector_add,      // kernel
    [256, 1, 1],     // grid dims
    [32, 1, 1],      // workgroup dims
    (&mut arg,),     // kernel args
  );
}
```

Available in upstream Rust!

Upsteam Rust: Simple & Portable Offloading

Use Case A: Write GPU code in pure Rust

Built on top of LLVM Offload, which works on several different architectures such as NVPTX, AMDGPU.

This makes Rust Offload portable!

```
pub unsafe extern "gpu-kernel" fn vector_add(A: &[f32], B: &[f32], C: &mut [f32]) {
  let i = unsafe { thread_idx_x() as usize };

  if let i < C.len() {
    C[i] = A[i] + B[i];
  }
}

fn main() {
  ...
  core::intrinsics::offload(
    vector_add,      // kernel
    [256, 1, 1],     // grid dims
    [32, 1, 1],      // workgroup dims
    (&mut arg,),     // kernel args
  );
}
```

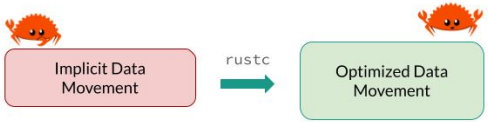
Available in upstream Rust!

Optimizing Data Transfer: Manual & Auto

Zero-Cost abstractions for safe Rust

Automatic data movement won't be as "smart" as manual optimization.

1. Add manual data movement
2. Compiler-level optimizations to cover most common cases.



Upsteam Rust: Simple & Portable Offloading

Use Case A: Write GPU code in pure Rust

Built on top of LLVM Offload, which works on several different architectures such as NVPTX, AMDGPU.

This makes Rust Offload portable!

```
pub unsafe extern "gpu-kernel" fn vector_add(A: &[f32], B: &[f32], C: &mut [f32]) {
  let i = unsafe { thread_idx_x() as usize };

  if let i < C.len() {
    C[i] = A[i] + B[i];
  }
}

fn main() {
  ...
  core::intrinsics::offload(
    vector_add,      // kernel
    [256, 1, 1],     // grid dims
    [32, 1, 1],      // workgroup dims
    (&mut arg,),     // kernel args
  );
}
```

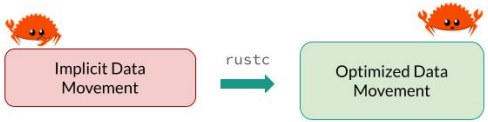
Available in upstream Rust!

Optimizing Data Transfer: Manual & Auto

Zero-Cost abstractions for safe Rust

Automatic data movement won't be as "smart" as manual optimization.

1. Add manual data movement
2. Compiler-level optimizations to cover most common cases.



Aliasing Models: where Rust knows more than LLVM

Optimizing `unsafe` code

Memtransfers require consistency, reordering is only safe with `noalias`.

Cannot optimize in presence of raw pointers and `UnsafeCell`
only with LLVM info

Rust's aliasing models: define rules for how references interact and overlap in memory.

Use high-level info to guarantee `noalias` safely.

Stacked Borrows[1] & Tree Borrows[2]

(1) Jung et al. Stacked Borrows: An Aliasing Model for Rust. POPL 2019.
(2) Villani et al. Tree Borrows. PLDI 2025.

Upsteam Rust: Simple & Portable Offloading

Use Case A: Write GPU code in pure Rust

Built on top of LLVM Offload, which works on several different architectures such as NVPTX, AMDGPU.

This makes Rust Offload portable!

```
pub unsafe extern "gpu-kernel" fn vector_add(A: &[f32], B: &[f32], C: &mut [f32]) {
  let i = unsafe { thread_idx_x() as usize };

  if let i < C.len() {
    C[i] = A[i] + B[i];
  }
}

fn main() {
  ...
  core::intrinsics::offload(
    vector_add,      // kernel
    [256, 1, 1],     // grid dims
    [32, 1, 1],      // workgroup dims
    (&mut arg,),     // kernel args
  );
}
```

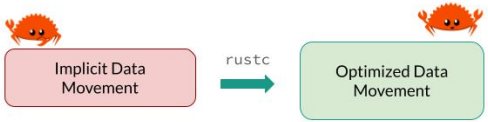
Available in upstream Rust!

Optimizing Data Transfer: Manual & Auto

Zero-Cost abstractions for safe Rust

Automatic data movement won't be as "smart" as manual optimization.

1. Add manual data movement
2. Compiler-level optimizations to cover most common cases.



Aliasing Models: where Rust knows more than LLVM

Optimizing `unsafe` code

Memtransfers require consistency, reordering is only safe with `noalias`.

Cannot optimize in presence of raw pointers and `UnsafeCell` only with LLVM info

Rust's aliasing models: define rules for how references interact and overlap in memory.

Use high-level info to guarantee `noalias` safely.

Stacked Borrows[1] & Tree Borrows[2]

(1) Jung et al. Stacked Borrows: An Aliasing Model for Rust. POPL 2019.
(2) Villani et al. Tree Borrows. PLDI 2025.

Performance: Competitive Kernel Times

RAJAPerf Benchmarks: raw kernel times