Productive Performance Portability: Building in Rust with PETSc and libCEED

Jeremy L Thompson

University of Colorado Boulder

jeremy@jeremylt.org

23 Feburary 2022

Funding

The authors acknowledge support by the Department of Energy, National Nuclear Security Administration, Predictive Science Academic Alliance Program (PSAAP) under Award Number DE-NA0003962.

Overview

Rust for HPC

- Rust provides high performance with modern tooling and ergonomic language features
- We look at the libCEED and PETSc Rust wrappers
- The libCEED wrapper provides performance portable matrix-free finite element operators
- The PETSc wrapper is a more complex ongoing challenge

Rust is ready for HPC - but it requires rethinking your design!

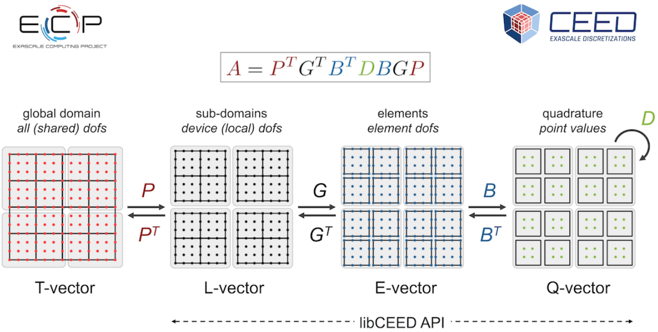
Rust - Language features

- Strongly typed language with a focus on performance and safety
- Compile-time memory management via lifetimes
- Borrow checker protects data, tracking ownership and access
- Many C run-time bugs become Rust compile-time errors

Rust - Ergonomics

- Installing - Cargo + crates.io manages dependencies vs Make
- Abstractions - zero cost abstractions for high-level language features
- Documentation - Automatic documentation with doctests using Cargo + docs.rs
- Unit Tests - Doctests and unit tests integrated with Cargo

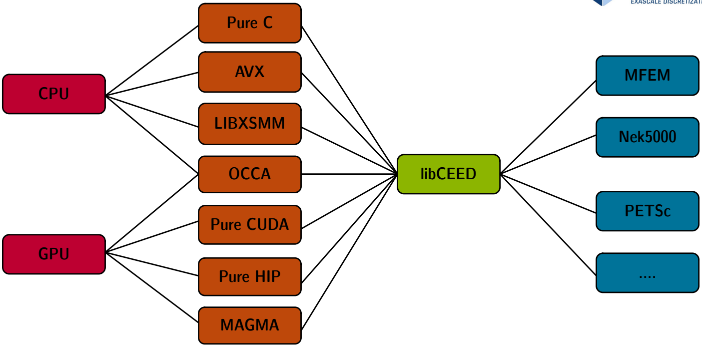
libCEED Representation



[2]

- **P** - parallel element assembly operator
- **G** - local element assembly operator
- **B** - basis action operator
- **D** - weak form and geometry at quadrature points

libCEED Backends



[2]

Multiple backends provide performance portability at runtime

Rust wrapper preserves this capabilty (mostly)!

libceed-rs

https://lib.rs/crates/libceed
https://github.com/ceed/libceed

- Faithful translation of C API into Rust
- Additional objects to help manage borrowed vector access
- Currently no native Rust GPU QFunction support

```rust
extern crate libceed;
fn main() -> libceed::Result<()> {
    let ceed = libceed::Ceed::init("/cpu/self/ref");
    let u = ceed.vector_from_slice(&[0.0, 0.5, 1.0])?;
    let u_view = u.view()?;
    assert_eq!(u_view[..], [0.0, 0.5, 1.0]);
    Ok(())
}
```

Building

Building dependencies reliabily across platforms can be... painful

```
...

ifneq ($(wildcard $(XSMM_DIR)/lib/libxsmm.*),)
  PKG_LIBS += -L$(abspath $(XSMM_DIR))/lib -lxsmm -ldl
  MKL ?=
  ifeq (,$(MKL)$(MKLROOT))
    BLAS_LIB = -lblas
  else
    ifneq ($(MKLROOT),)
      # Some installs put everything inside subdirectory
      MKL_LIBDIR = $(dir $(firstword $(wildcard $(MKLROOT)/lib/intel64/libmkl_sequentia
      MKL_LINK = -L$(MKL_LIBDIR)
      PKG_LIB_DIRS += $(MKL_LIBDIR)
    endif
    BLAS_LIB = $(MKL_LINK) -Wl,--push-state,--no-as-needed -lmkl_intel_lp64 -lm
  endif
  PKG_LIBS += $(BLAS_LIB)
endif
...
```

Building

Far simpler dependency specification for Rust!

```toml
[package]
name = "ex1-volume"
version = "0.1.0"
authors = [
    "Jeremy L Thompson <thompson.jeremy.luke@gmail.com>",
]
edition = "2018"

[dependencies]
libceed = "0.9.0"
structopt = { version = "0.3", default-features = false }
mesh = { path = "../mesh" }
```

Easier building

```
$ cargo build
```

High Level Abstractions - User QFunctions

User source for physics at quadrature points requires some additional boilerplate in C/CUDA single source

```
CEED_QFUNCTION(MassApply)(void *ctx, const CeedInt Q,
                          const CeedScalar *const *in,
                          CeedScalar *const *out) {
  // Unpack input and output arrays
  const CeedScalar *u = in[0], *rho = in[1];
  CeedScalar *v = out[0];

  // Loop over all quadrature points
  CeedPragmaSIMD
  for (CeedInt i = 0; i < Q; i++) {
    v[i] = u[i] * rho[i];
  }

  return 0;
}
```

High Level Abstractions - User QFunctions

Rust inteface presents more clarity, less opportunity for error

```
// QFunction from user closure
let apply_mass = |
    [u, qdata, ..]: QFunctionInputs,
    [v, ..]: QFunctionOutputs,
| {
    // Protected array access!
    v.iter_mut()
        .zip(u.iter().zip(qdata.iter()))
        .for_each(|(v, (u, rho))| *v = u * rho);

    0
};
```

Documentation

Documentation and testing separate in C
Often multiple processing steps to generate documentation

```c
/**
  @brief Create a CeedVector of the specified length
             (does not allocate memory)

  @param       ceed   Ceed context
  @param       length Length of vector
  @param[out]  vec    Address where the newly created
                        CeedVector will be stored

  @return An error code: 0 - success, otherwise - failure

  @ref User
**/
int CeedVectorCreate(Ceed ceed, CeedInt length,
                     CeedVector *vec) { /* */ };
```

Documentation

Integrated documentation and unit tests in Rust

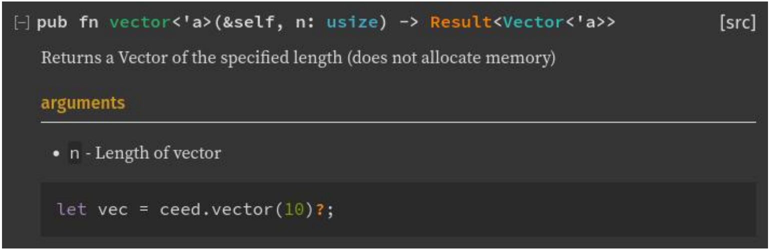
```
/// Returns a Vector of the specified length
///   (does not allocate memory)
///
/// # arguments
///
/// * ‘n‘ - Length of vector
///
/// ‘‘‘
/// # use libceed::prelude::*;
/// # fn main() -> libceed::Result<()> {
/// # let ceed = libceed::Ceed::default_init();
/// let vec = ceed.vector(10)?;
/// # Ok(())
/// # }
/// ‘‘‘
pub fn vector<’a>(&self, n: usize) -> Result<Vector<’a>> { /* */ }
```

Documentation

Easier testing and documentation

```
$ cargo test
$ cargo fmt
$ cargo doc
```

Automatically tested code snippits in documentation examples



```
[-] pub fn vector<'a>(&self, n: usize) -> Result<Vector<'a>>        [src]
```

Returns a Vector of the specified length (does not allocate memory)

arguments

- `n` - Length of vector

```
let vec = ceed.vector(10)?;
```

Lost in Translation

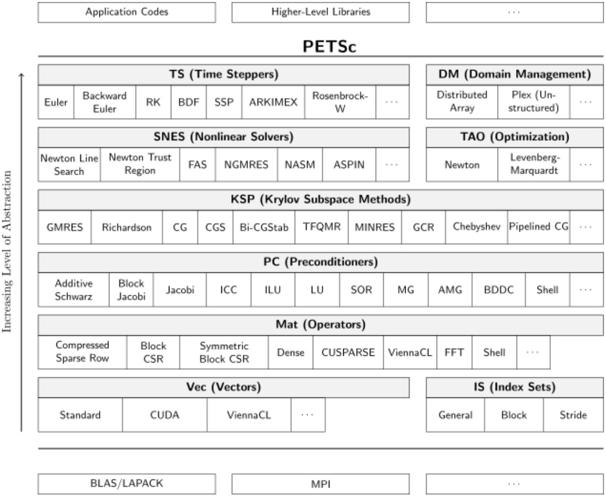
Passive vectors can be mutated by operators

```
// Mass operator
let qf_mass = ceed.q_function_interior_by_name("MassApply")?;
let op_mass = ceed
      .operator(&qf_mass, QFunctionOpt::None, QFunctionOpt::None)?
      .field("u", &ru, &bu, VectorOpt::Active)?
      .field("qdata", &rq, BasisOpt::Collocated, &qdata)?
      .field("v", &ru, &bu, VectorOpt::Active)?
      .check()?;

v.set_value(0.0)?;
op_mass.apply(&u, &mut v)?;
```

If `"qdata"` was an output, the operator could mutate the underlying data without the Rust compiler knowing

PETSc - the Portable, Extensible Toolkit for Scientific Computation



[1]

petsc-rs

https://lib.rs/crates/petsc
https://gitlab.com/petsc/petsc-rs

- Attempt to maintain flexibility of C API
- Wider range of additional objects to manage borrowed access
- Currently incomplete shell object support

High Level Abstractions - Matrix assembly

```rust
let n = 5;
let mut mat = petsc.mat_create()?;
mat.set_sizes(None, None, n as usize, n as usize)?;
mat.set_from_options()?;
mat.set_up()?;

// Stencil (-1, 2, -1)
let v = [Scalar::from(-1.0), Scalar::from(2.0), Scalar::from(-1.0)];
mat.assemble_with_batched(
    (0..n)
        .map(|i| {
            if i == 0 { ([i], [-1, i, i + 1], &v) }
            else if i == n - 1 { ([i], [i - 1, i, -1], &v) }
            else { ([i], [i - 1, i, i + 1], &v) }
        }),
    InsertMode::INSERT_VALUES,
    MatAssemblyType::MAT_FINAL_ASSEMBLY,
)?;
```

Lost in Translation - Part II

- PETSc objects have bi-directional data ownership
- Compiler can't reliably reason about complex lifetimes
- The libCEED wrapper provides performance portable matrix-free finite element operators
- The PETSc wrapper is a more complex ongoing challenge

Rust for HPC

- Rust provides high performance with modern tooling and ergonomic language features
- We looked at the libCEED and PETSc Rust wrappers
- The libCEED wrapper provides performance portable matrix-free finite element operators
- The PETSc wrapper is a more complex ongoing challenge

Rust is ready for HPC - but it requires rethinking your design!

Future Work

- Expand PETSc Rust wappers
- Native Rust libCEED QFunctions for GPU
- PETSc + libCEED Rust examples
- Clarify inner mutability vs interface level mutability across FFI barrier
- Disentangling object lifetimes

Productive Performance Portability: Building in Rust with PETSc and libCEED

Jeremy L Thompson

University of Colorado Boulder

jeremy@jeremylt.org

23 Feburary 2022

Satish Balay, Shrirang Abhyankar, Mark F Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G Knepley, Dave A May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F Smith, Stefano Zampini, Hong Zhang, and Hong Zhang.
PETSc users manual.
Technical Report ANL-95/11 - Revision 3.15, Argonne National Laboratory, 2021.

Jed Brown, Ahmad Abdelfattah, Valeria Barra, Natalie Beams, Jean Sylvain Camier, Veselin Dobrev, Yohann Dudouit, Leila Ghaffari, Tzanio Kolev, David Medina, Will Pazner, Thilina Ratnayaka, Jeremy L Thompson, and Stanimire Tomov.
libCEED: Fast algebra for high-order element-based discretizations.
Journal of Open Source Software, 6:1945, 2021.

Productive Performance Portability: Building in Rust with PETSc and libCEED

Jeremy L Thompson

University of Colorado Boulder

jeremy@jeremylt.org

23 Feburary 2022