# Design Document: A Python `shmem4py` Frontend Targeting an MLIR OpenSHMEM Dialect

Michael Beebe

## 1 Overview

This document outlines the design of a compiler frontend that translates existing `shmem4py` programs into an MLIR-based OpenSHMEM dialect. The goal is to enable whole-program OpenSHMEM optimizations such as message aggregation, asynchronous communication conversion, collective fusion, and future GPU/NVSHMEM support.

The major components of the system are:

- A C++/ODS OpenSHMEM dialect (already implemented).
- Python bindings for that dialect, enabling construction of IR from Python.
- A Python frontend that parses or traces `shmem4py` code and produces structured MLIR containing `scf`, `memref`, `arith`, and the OpenSHMEM dialect.
- Optimization and lowering pipelines implemented in MLIR.
- Backends targeting C/OpenSHMEM or LLVM (and later NVSHMEM).

## 2 Goals and Non-Goals

### 2.1 Goals

- Allow the user to write *unmodified* `shmem4py` code.
- Emit MLIR with structured control flow, explicit memory semantics, and explicit communication operations.
- Preserve enough semantic information to safely perform communication optimizations.
- Support both offline compilation and optional JIT-style usage.
- Build a foundation for GPU/NVSHMEM extensions.

### 2.2 Non-Goals

- Supporting full dynamic Python semantics (reflection, generators, meta-classes).
- Supporting advanced NumPy features (broadcasting, fancy indexing, views).
- Compiling arbitrary Python programs.
- Supporting all of `shmem4py` initially.

## 3 High-Level Architecture

The architecture consists of the following layers:

1. **OpenSHMEM Dialect (C++/ODS):** Defines operations, types, attributes, and passes.
2. **Python Bindings:** The dialect must be exposed to MLIR's Python bindings so Python code can construct IR.
3. `shmem4py` **Frontend:** A Python library that extracts IR from Python functions via AST or tracing.
4. **Driver Infrastructure:** CLI tool and optional decorators for JIT.
5. **Backend:** C/OpenSHMEM stub lowering or LLVM lowering; future NVSHMEM lowering.

## 4 IR Requirements

Optimizations such as message aggregation and async conversion require IR expressing:

- Explicit loops (`scf.for`).
- Explicit memory operations (`memref.load/store`, `memref.subview`).
- Arithmetic and index manipulation (`arith`).
- OpenSHMEM communication (`openshmem.put`, `openshmem.get`, `openshmem.barrier_all`, etc.).

Below is an illustrative example of the kind of MLIR the frontend should produce. This IR corresponds to a nearest-neighbor halo exchange written in `shmem4py`.

```
func.func @halo_step(%u: memref<?xf64, #openshmem.sym>,
                     %niters: index) {
  %me   = openshmem.my_pe  : index
  %npes = openshmem.n_pes  : index
  %n    = memref.dim %u, %c0 : memref<?xf64, #openshmem.sym>

  scf.for %it = %c0 to %niters step %c1 {
```

```
    scf.for %i = %c1 to %n_minus_1 step %c1 {
      ... interior computation ...
    }

    openshmem.barrier_all

    %left_src = memref.subview %u[...] : memref<1xf64, #openshmem.sym>
    openshmem.put %left_src, %left, %dest_off_left

    %right_src = memref.subview %u[...] : memref<1xf64, #openshmem.sym>
    openshmem.put %right_src, %right, %c0

    openshmem.barrier_all
  }

  return
}
```

This form exposes PE relationships, ranges, and synchronization structure.

# 5 Python Bindings for the Dialect

## 5.1 Requirements

- Dialect ops must be defined in ODS so bindings can be auto-generated.
- The dialect must register itself with MLIR's Python extension module.
- The build must enable `MLIR_ENABLE_BINDINGS_PYTHON=ON`.

## 5.2 Result

Once bindings are generated, Python code can construct OpenSHMEM ops:

```
from mlir.ir import *
from mlir.dialects import openshmem, scf, arith, memref

with Context() as ctx:
    module = Module.create()
    with InsertionPoint(module.body):
        pe = openshmem.MyPeOp()
```

# 6 Frontend Design

## 6.1 Package Layout

- `python/shmem4py_mlir/frontend.py` — AST visitor / tracer.

- `python/shmem4py_mlir/builder.py` — IR construction helpers.
- `python/shmem4py_mlir/passes.py` — convenience wrappers for your optimization pipeline.
- `python/shmem4py_mlir/jit.py` — optional decorator-based JIT usage.
- `python/shmem4py_mlir/cli.py` — command-line interface.

### 6.2 Supported Subset of `shmem4py`

- **Control Flow:** `for i in range(...)`, `if / elif / else`.
- **Data:** 1D or 2D NumPy arrays mapped to `memref`.
- **OpenSHMEM Ops:**
  - `shmem.my_pe()`, `shmem.n_pes()`
  - `shmem.barrier_all()`
  - `shmem.put(buf, pe, idx)`
  - `shmem.get(buf, pe, idx)`

### 6.3 AST to MLIR Mapping

Using Python's `inspect` and `ast` modules:

- `ast.FunctionDef` → `func.func`.
- `ast.For` with `range()` → `scf.for`.
- `ast.If` → `scf.if`.
- `ast.Assign`, `ast.AugAssign` → `memref.load/store` + `arith`.
- `ast.Subscript` (e.g., `u[i]`, `u[a:b]`) → index calculations and/or `memref.subview`.
- `ast.Call` into `shmem` module → OpenSHMEM dialect ops.

## 7 Integration with `shmem4py`

### 7.1 Offline Compilation

CLI usage:

```
shmem4py-mlir myprog.py --fn halo_step --emit-mlir=ir.mlir --emit-exe=a.out
```

Pipeline:

1. Import module and locate the function.

2. AST → MLIR.
3. Run OpenSHMEM optimization passes.
4. Lower to LLVM or C/OpenSHMEM.
5. Produce executable.

### 7.2 Optional JIT Mode

```
from shmem4py_mlir import shmem_jit

@shmem_jit
def halo_step(u, niters):
    ...
```

The decorator:

1. Extracts and lowers the function to MLIR.
2. Runs pass pipeline.
3. JIT-compiles via LLVM backend.

## 8 Backend Strategy

### 8.1 CPU OpenSHMEM

Two lowering strategies are possible:

- **C/OpenSHMEM Stub Emission:** Lower to C code calling `shmem_*` routines.
- **LLVM Lowering:** Implement LLVM dialect conversions so `openshmem.*` ops lower to function calls into the OpenSHMEM C API.

### 8.2 Future NVSHMEM/GPU Support

Future phases will:

- Extend the dialect with GPU/NVSHMEM types, attributes, and ops.
- Recognize NVSHMEM4Py imports.
- Lower device-side regions to MLIR's GPU dialect or Triton.
- Lower communication ops to NVSHMEM device and host APIs.

# 9 Roadmap

## 9.1 Phase 0: Preparatory Work

- Ensure dialect is cleanly ODS-based.
- Add analysis attributes/types as needed.

## 9.2 Phase 1: Python Bindings

- Add Python bindings generation to build.
- Verify `import mlir.dialects.openshmem` works.

## 9.3 Phase 2: Minimal Frontend

- Implement AST visitor for loops, arithmetic, subscript, and communication.
- Support a minimal `shmem4py` subset.
- CLI that emits MLIR for simple examples.

## 9.4 Phase 3: Optimization + Backend

- Run OpenSHMEM optimization passes.
- Lower to C/OpenSHMEM or LLVM + runtime calls.
- Validate on microbenchmarks (halo, 2D stencil, ping-pong, etc.).

## 9.5 Phase 4: Extensions and NVSHMEM

- Add more `shmem4py` operations.
- Add JIT mode.
- Begin GPU-side NVSHMEM dialect and lowering.

# 10 Conclusion

This design leverages your existing C++ MLIR dialect, introduces Python bindings, and builds a practical frontend for `shmem4py` programs. It provides a clean path toward structured IR suitable for communication optimizations while offering a user-friendly Python interface and long-term extensibility to NVSHMEM.