ReProspect

A framework for reproducible prospecting of CUDA applications

Romin Tomasetti [1] Maarten Arnst [1]

[1]University of Liège

HPSF Community Summit
TU Braunschweig, Germany
February 25th, 2026

HPSF
HIGH PERFORMANCE SOFTWARE FOUNDATION

LIÈGE université

fnrs
FREEDOM TO RESEARCH

What does your code *actually* do?

You wrote it. You compiled it. You ran it.
But did it do what you *think* it did?

Which API calls are happening under the hood?

```
using view_t = Kokkos::View<char*, Kokkos::CudaSpace>;
view_t data(Kokkos::view_alloc(exec, "data"), 42);
```

⇒ API tracing

How will this impact performance?

```
template <class RealType>
class
#ifdef KOKKOS_ENABLE_COMPLEX_ALIGN
    alignas(2 * sizeof(RealType))
#endif
complex { RealType re_, im_; };
```

⇒ Kernel profiling

Is it wired to the right half precision intrinsics?

```
Kokkos::half_t a = ..., b = ...;
c = Kokkos::fmax(a, b);
```

⇒ Binary analysis

API tracing

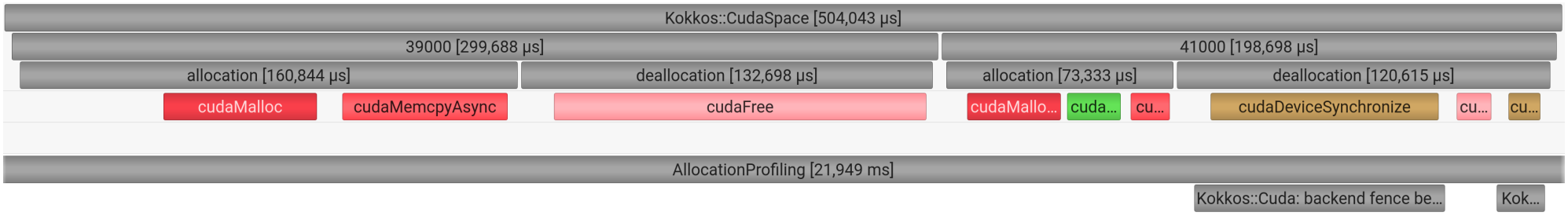
You need performance portability and turn to Kokkos.

```
using view_t = Kokkos::View<char*, Kokkos::CudaSpace>;
view_t data(Kokkos::view_alloc(Kokkos::WithoutInitializing, exec, "data"), 42);
```

Questions:

- What CUDA API calls are made for a given set of arguments?
- Is there any synchronization happening?

```
nsys profile --cuda-memory-usage=true --capture-range=nvtx \
    --trace=nvtx,cuda ... a.out --kokkos-tools-libs=...
```



- 🤔 As a user, you want to understand what happens.
- 🛠️ As an implementer, you want to optimize/test the behavior **beyond output correctness**.

Kernel profiling

You're doing electromagnetic simulations in the frequency domain and need `Kokkos::complex<double>`.

```
template <class RealType>
class
#ifdef KOKKOS_ENABLE_COMPLEX_ALIGN
    alignas(2 * sizeof(RealType))
#endif
complex { RealType re_, im_; };
```

🤔 As a user, you'd like to understand the implications in terms of:

1. memory traffic (L1/TEX and L2 cache misses/hits, global memory)
2. executed instructions

```
ncu --metrics=... --nvtx ... a.out --kokkos-tools-libs=...
```

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req
Local Load	0	0	0	0	0	0
Global Load	192	192			3.072	16

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req
Local Load	0	0	0	0	0	0
Global Load	96	96			1.536	16

Binary analysis - SASS

Kokkos provides half types. Support varies accross architectures/vendors, leading to a macro-and-templates implementation 💣

```
Kokkos::half_t a = ..., b = ...;
c = Kokkos::fmax(a, b);
```

- 🛠️ As an implementer, you need to ensure that the **correct code path is taken**. You'd craft sensitive runtime tests. You'll need access to devices 💸

What if you could test the generated SASS code instead?

```
cuobjdump --dump-sass -arch=sm_120 a.out
```

```
__hmax(const __half, const __half)
```

```
LDG.E.U16 R2, desc[UR6][R2.64]
LDG.E.U16 R5, desc[UR6][R4.64]
...
HMNMX2 R5, R2.H0_H0, R5.H0_H0, !PT
...



STG.E.U16 desc[UR6][R6.64], R5
```

```
fmax(const float, const float)
```

```
LDG.E.U16 R2, desc[UR6][R2.64]
LDG.E.U16 R4, desc[UR6][R4.64]
...
HADD2.F32 R6, -RZ, R2.H0_H0
HADD2.F32 R7, -RZ, R4.H0_H0
FMNMX R6, R6, R7, !PT
F2FP.F16.F32.PACK_AB R3, RZ, R6
...
STG.E.U16 desc[UR6][R6.64], R3
```

Binary analysis - ELF

You're implementing a new feature in `Kokkos` that launches a parallel region on device with a single thread. You tie it to `Kokkos::LaunchBounds<1>`.

```
Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::LaunchBounds<1>>(...), ...);
```

How do you validate that it maps to:

```
__global__ __launch_bounds__(1, ...) void kernel() {...}
```

Such kernel attributes are encoded in the ELF! 

```
cuobjdump --dump-elf -arch=sm_120 a.out

<0x1>
Attribute: EIATTR_CUDA_API_VERSION
Format:    EIFMT_SVAL
Value:     0x82
...
<0x9>
Attribute: EIATTR_MAX_THREADS
Format:    EIFMT_SVAL
Value:     0x1 0x1 0x1
```

Towards fully programmatic analysis

All these tools are great but:

 require a human (GUI),

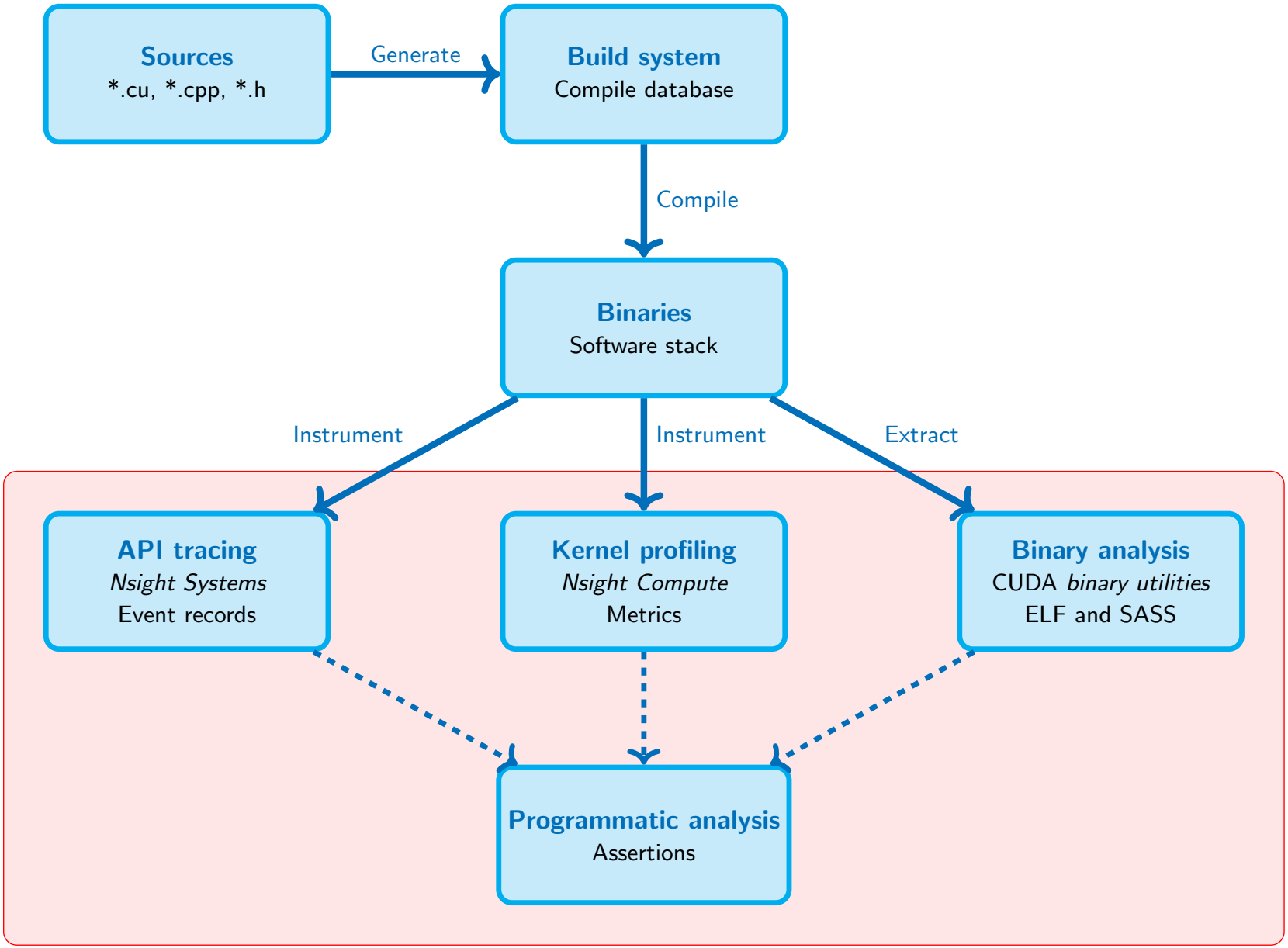
 significant boilerplate code to be written

 and is veeery procedural.

And then you start wondering about:

- Sharing analyses
- Reproducibility
- Integration in your CI pipeline
- Bundle nice research artifacts

reprospect


Sources
*.cu, *.cpp, *.h
Generate
Build system
Compile database
Compile
Binaries
Software stack
Instrument
Instrument
Extract
API tracing
Nsight Systems
Event records
Kernel profiling
Nsight Compute
Metrics
Binary analysis
CUDA binary utilities
ELF and SASS
Programmatic analysis
Assertions

Typical workflow

Streamline data collection and extraction from NVIDIA tools.
Complement them with new functionalities for a fully programmatic analysis.

Typical structure:

- An executable you want to test.
- A concise Python script that drives the analysis.

test_fmax.cpp

```cpp
template <Method method, typename ViewType>
struct FunctorMax {
    ViewType dst;
    typename ViewType::const_type src_a, src_b;

    template <std::integral T> requires (method == Method::CUDA_HMAX)
    KOKKOS_FUNCTION void operator()(const T idx) const {
        dst(idx) = __hmax(src_a(idx).operator __half(), src_b(idx).operator __half());
    }

    template <std::integral T> requires (method == Method::FMAX)
    KOKKOS_FUNCTION void operator()(const T idx) const {
        dst(idx) = fmax(src_a(idx).operator float(), src_b(idx).operator float());
    }

    template <std::integral T> requires (method == Method::KOKKOS_FMAX)
    KOKKOS_FUNCTION void operator()(const T idx) const {
        dst(idx) = Kokkos::fmax(src_a(idx), src_b(idx));
    }
};
```

test_fmax.py

```python
def test_cuda_hmax(self, decoder: dict[Method, Decoder]) -> None:
    """
    Check SASS code for :py:attr:`Method.CUDA_HMAX`.

    .. note::

        Before compute capability 8.0, the intrinsic :code:`__hmax` generates FP32 ins
    """
    if self.arch.compute_capability >= 80:
        self.match_fp16(instructions=decoder[Method.CUDA_HMAX].instructions)
    else:
        self.match_fp32(instructions=decoder[Method.CUDA_HMAX].instructions)

def test_kokkos_fmax(self, decoder: dict[Method, Decoder]) -> None:
    """
    Check SASS code for :py:attr:`Method.KOKKOS_FMAX`.

    .. note::

        It always leads to the exact same SASS code as :py:attr:`Method.CUDA_HMAX`, th
        it is implemented correctly.
    """
    assert decoder[Method.KOKKOS_FMAX].instructions == decoder[Method.CUDA_HMAX].instr

def test_fmax(self, decoder: dict[Method, Decoder]) -> None:
    """
    Check SASS code for :py:attr:`Method.FMAX`.
    """
    self.match_fp32(instructions=decoder[Method.FMAX].instructions)
```

API tracing and kernel profiling

Encapsulate the entire `nsys`/`ncu` analysis in a concise `Python` script as:

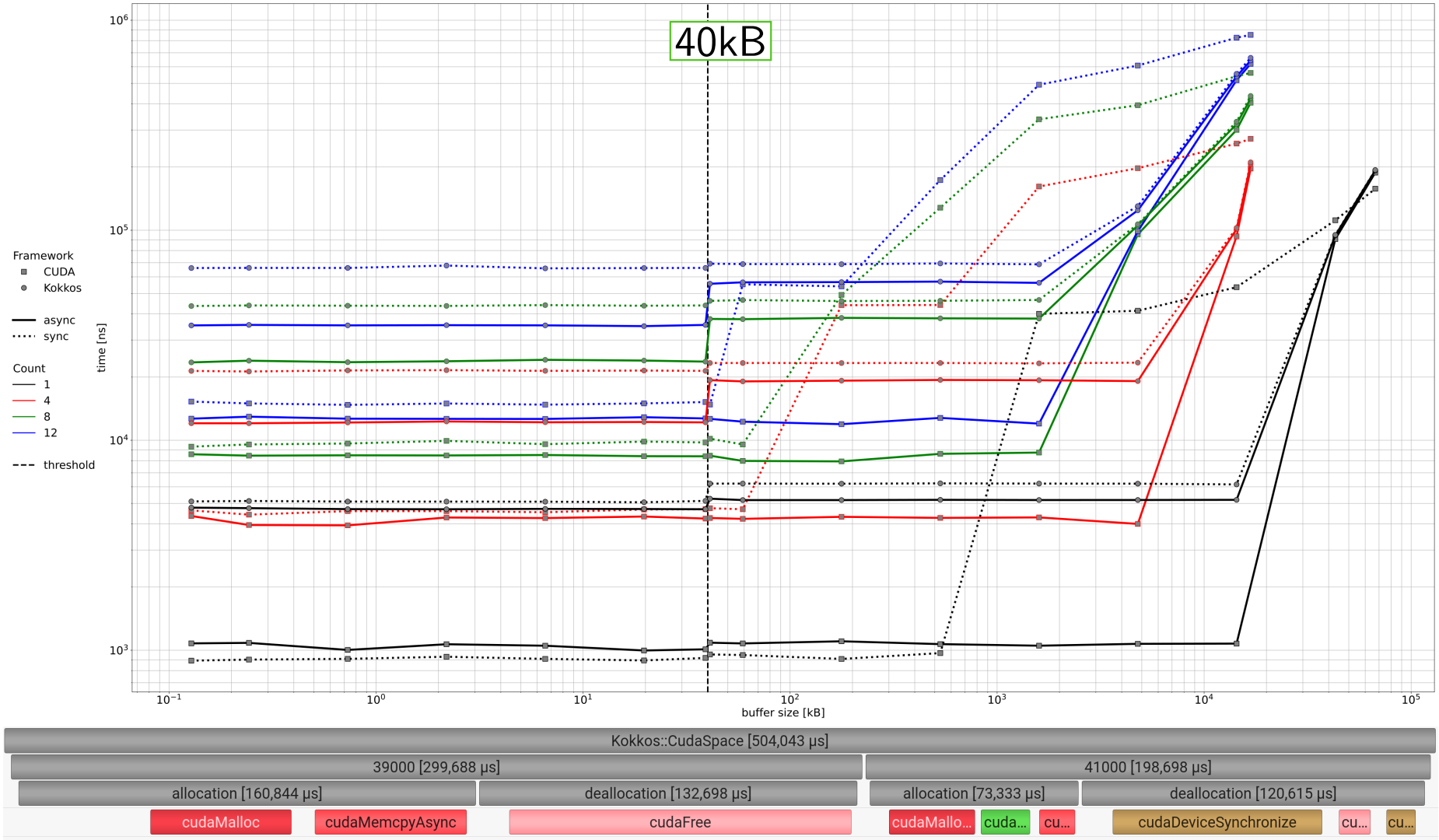
- `cacher = Cacher(directory=workdir)` (optional)
- `command = Command(executable='a.out', ...)`
- `session = Session(command=...); session.run(...)` or `entry = cacher.run(command, ...)`
- `report = Report(...) # from session or entry`
- `assert my_metric == 42`
- `...`

The `report` structures data by NVTX range annotations.

```
Profiling results
└─ my_nvtx_push_region
   └─ my_kernel
      ├─ smsp__inst_executed.sum: 100.0
      └─ L1/TEX cache global load sectors.sum: 0.0
```

API tracing to elucidate benchmarking results

```
using view_t = Kokkos::View<char*, Kokkos::CudaSpace>;
view_t data(Kokkos::view_alloc(Kokkos::WithoutInitializing, exec, "data"), 42);
```



core(allocation): avoid copying shared allocation header on device if not needed

Binary analysis - Instruction matching

CUDA does not document SASS.
ReProspect ships with an extensible SASS knowledge base.

SASS instructions depend on the architecture:

```
LDG.E.128.CONSTANT.SYS R8 [R2]              /* up to Turing 75 */
LDG.E.128.CONSTANT     R8 [R2.64]           /* up to Ada 89    */
LDG.E.128.CONSTANT     R8 desc[UR4][R2.64]
    LoadGlobalMatcher(arch=..., size=128, readonly=True)
```

... but also on bits extension, memory location, and so on:

```
LDG.E.U16.CONSTANT R3, [R2.64]
    LoadMatcher(arch=cc86, size=16, readonly=True, extend='U', memory='G')
LDG.E.S16.CONSTANT R3, [R2.64]
    LoadGlobalMatcher(arch=cc86, size=16, readonly=True, extend='S')
LD.E.64 R2, desc[UR10][R4.64]
    LoadMatcher(arch=cc100, size=64, memory=MemorySpace.GENERIC)
LDG.E.64.CONSTANT R22, desc[UR10][R48.64+0x180]
    LoadMatcher(arch=cc90, size=64, readonly=True, memory='G')
LDG.E.ENL2.256.CONSTANT R12, R8, desc[UR4][R2.64]
    LoadMatcher(arch=cc120, size=256, readonly=True, memory='G')
```

Binary analysis - Instruction sequence pattern matching

```
LDG.E.U16 R2, desc[UR6][R2.64]
LDG.E.U16 R5, desc[UR6][R4.64]
...
HMNMX2 R5, R2.H0_H0, R5.H0_H0, !PT
...



STG.E.U16 desc[UR6][R6.64], R5
```

```
LDG.E.U16 R2, desc[UR6][R2.64]
LDG.E.U16 R4, desc[UR6][R4.64]
...
HADD2.F32 R6, -RZ, R2.H0_H0
HADD2.F32 R7, -RZ, R4.H0_H0
FMNMX R6, R6, R7, !PT
F2FP.F16.F32.PACK_AB R3, RZ, R6
...
STG.E.U16 desc[UR6][R6.64], R3
```

```
cfg = ControlFlow.analyze(instructions)

matcher_ldg = instructions_contain(instructions_are(
    LoadGlobalMatcher(arch, size=16, extend='U', readonly=False),
    LoadGlobalMatcher(arch, size=16, extend='U', readonly=False)))
blk, matched_ldg = BasicBlockMatcher(matcher_ldg).match(cfg)

matcher_hmnmx2 = instructions_contain(instruction_is(
    Fp16MinMaxMatcher(pmax=True))
    .with_operand(index=1, operand=f'{matched_ldg[0].operands[0]}.H0_H0')
    .with_operand(index=2, operand=f'{matched_ldg[1].operands[0]}.H0_H0')
    .with_operand(index=0, operand=RegisterMatcher(special=False)))
matched_hmnmx2 = matcher_hmnmx2.match(blk.instructions[matcher_ldg.next_index:])

matcher_stg = instructions_contain(instruction_is(
    StoreGlobalMatcher(arch, size=16, extend='U')
    .with_operand(index=1, operand=matched_ldg[0].operands[0])
matched_stg = matcher_stg.match(blk.instructions[matched_hmnmx2.next_index:])
```

SASS matching to validate code path

Kokkos provides extended atomic operations support through desul:

- native atomic instruction
- CAS loop
- sharded lock table (slowest)

Testing which method is used with a micro-benchmarking approach is feasible, but requires the physical device and suffers from **runtime variability**.

```
Kokkos::atomic_add(&data(index), value); // __int128_t
```

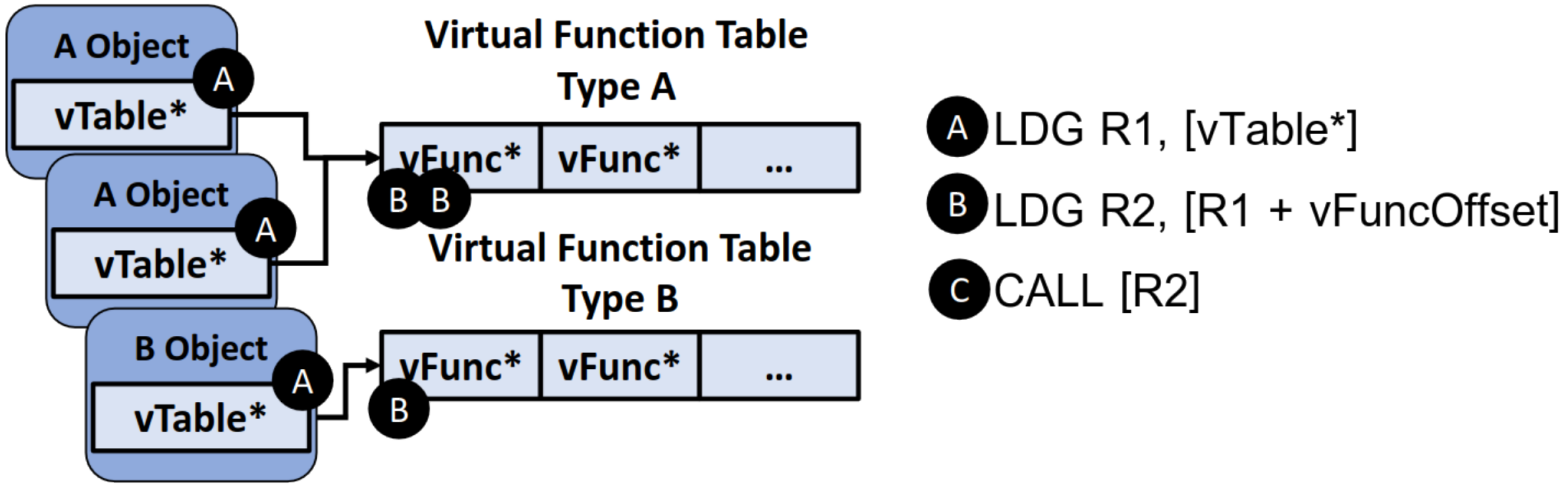
Validate Kokkos dispatch logic with custom SASS pattern matchers:

```
if arch.compute_capability.as_int < 90
    LockBasedAtomicMatcher(arch,
                           operation=AddInt128(),
                           compiler_id=toolchains['CUDA']['compiler']['id'])
        .assert_matches(decoder.instructions)
else:
    AtomicCASMatcher(arch,
                     operation=AddInt128(),
                     size=128)
        .assert_matches(cfg=ControlFlow.analyze(decoder.instructions))
```

Reproducing research artifacts

- Analyse memory accesses and branch instructions generated by `ptxas` for virtual function dynamic dispatch.
- Use `ReProspect` to reproduce their observations in a programmatic way.
- Leverage SASS matching and ELF metadata (resource usage, constant memory bank correlation, and so on).

Judging a Type by Its Pointer: Optimizing GPU Virtual Functions



Zhang et al., `https://dl.acm.org/doi/10.1145/3445814.3446734`

Origins, Roadmap & Outlook

Successful use cases

- Support code design decisions
- Benchmark interpretation
- Regression tests

Directions

- HIP support in progress
- `rust` for faster parsing
- PTX analysis?

Community adoption

- Contribute to `Kokkos` PRs
- Extensible framework

Horizon

- Agentic/human optimization
- `CuTile` primitives

Find out more!

Feedback and contributions welcome!

- 🐛 Open issues or submit PRs for bug fixes
- ✨ Propose or implement new features
- 🤝 Collaborate with us on new use cases

uliegecsm/reprospect

documentation

JOSS Under Review