oneDNN

Mourad Gouicem, Intel, oneDNN Lead Architect

Unified Acceleration Foundation

Libraries? What for?

Developer relies on commonly useful feature

- Dev knows about library implementing it

Developer relies on novel/custom feature

- Typically not part of a library
- Dev just implements the feature themselves
- Library eventually adds the feature, but dev does not necessarily migrate to library once the feature is available there.
  Better perf usually justifies the migration.

Challenge for AI libraries, as rate of new feature introduction is high

What are those new features

Fused patterns

- Composition of smaller known operations (e.g. SDPA)
- Fusion is mostly required for performance reasons (fewer passes over memory)

Numerical recipes

- Sometimes require new datatypes (e.g. int4, mxfp4, nf4, …)
- Relies on higher precision implementation (through upconversion) or HW acceleration (systolic)
- Non-fused implementation defeat the purpose (reduce memory capacity/BW requirements)

Novel operations not expressed as composition of existing ones

- New activation function that are not composition of existing ones
- New normalization algorithm
- Less frequent than the two above

UXL Unified Acceleration Foundation

An API for each usage

	Graph API	Primitive API	ukernel API
Fused patterns support	+ No API extension needed - Medium TTM for oneDNN impl	- No API extension if post-op - API extension if not post-op - Medium TTM for oneDNN impl	+ No API extension needed + User can fuse custom operation - User needs to optimize sharding/parallelization for fused implementation
Emerging numerical recipes	- API extension needed - Medium TTM for oneDNN impl	- API extension needed - Medium TTM for oneDNN impl	+ No API extension for up-conversion logic - API extension for register-level fusion
Custom operation	- API extension needed	- API extension needed	- API extension needed
	Maps composite operators Flexible fusion	Maps framework operators	Maps to block level abstractions (Eigen, Aten) Composable with custom operations Allows efficient experimentation

Responsibility transfer

Composability vs optimization

		Graph API	Primitive API	ukernel API
Optimization responsibility	oneDNN	Hides ISA complexities Hides sharding Hides parallelization	Hides ISA complexities Hides sharding Hides parallelization	Hides ISA complexities
	User	Creates oneDNN objects	Creates oneDNN objects	Creates oneDNN objects Controls parallelization Controls sharding Manages HW state (AMX/SME)
Composability		Main memory between partitions In-cache/in-register inside partition	Main memory between primitives In-cache/in-register between op/post-op	In-cache between ukernels/custom kernels if proper sharding
		Fully optimized, but higher TTM for new features/pattern		More flexible and efficient composability for experimentation. Important optimization work on user side.

Ukernel for experimentation. Continued graph/primitive extensions as features get traction.

UXL Unified Acceleration Foundation

Example: matmul primitive

```
// Create runtime abstractions
engine eng(engine_kind, 0);
stream strm(eng);

// Create memory descriptors.
memory::desc src_md(src_dims, dt::bf16, tag::ab);
memory::desc weights_md(weights_dims, dt::bf16,
tag::ab);
memory::desc dst_md(dst_dims, dt::f32, tag::ab);

// Create primitive descriptor.
matmul::primitive_desc matmul_pd(eng,
    src_md, weights_md, dst_md);

// Create the primitive (jitting).
matmul matmul_prim(matmul_pd);
```

```
// Create memory abstractions
memory src_mem(src_md, eng);
memory weights_mem(weights_md, eng);
memory dst_mem(dst_md, eng);

// Primitive arguments.
std::unordered_map<int, memory> matmul_args =
    {{DNNL_ARG_SRC, src_mem},
     {DNNL_ARG_WEIGHTS, weights_mem},
     {DNNL_ARG_DST, dst_mem}};

// Primitive execution
matmul_prim.execute(strm, matmul_args);

// Wait for the computation to finalize.
strm.wait();
```

Historical API: maps to framework operations

Example: matmul graph

```
// Create runtime abstractions
engine eng(engine_kind, 0);
stream strm(eng);

//Create matmul arguments (graph edges)
logical_tensor src_lt(0, data_type::bf16,
    src_dims, layout_type::strided);
logical_tensor weights_lt (1, data_type::bf16,
    weights_dims, layout_type::strided);
logical_tensor dst_lt (2, data_type::f32,
    dst_dims, layout_type::strided);

// Create ops (graph nodes)
op matmul(0, op::kind::MatMul,
    {src_lt, weights_lt}, {dst_lt}, "matmul");
matmul.set_attr<bool>(op::attr::transpose_a, false);
matmul.set_attr<bool>(op::attr::transpose_b, false);
```

```
// Create graph and compile the resulting partition
graph g(engine::kind::gpu);
g.add_op(matmul);
g.finalize();
compiled_partition cp = g.get_partition()[0].compile(
    {src_lt, weights_lt}, {dst_lt}, eng);

// Create memory abstractions
tensor src(src_lt, eng);
tensor weights(weights_lt, eng);
tensor dst(dst_lt, eng);

// Primitive execution
cp.execute(strm, {src, weights}, {dst});

// Wait for the computation to finalize.
strm.wait();
```

Graph abstraction allows efficient handling of complex patterns

Example: matmul ukernel

```
// We create the brgemm ukernel object
brgemm brg(K / BK, BM, BN, BK,
  data_type::bf16, K,
  data_type::bf16, BN,
  data_type::f32, BN);
brg.finalize();
brg.generate();

// We create the transform ukernel object
transform tB(BK, BN,
  data_type::bf16, pack_type::no_trans, N,
  data_type::bf16, brg.get_B_pack_type(), BN);
tB.finalize();
tB.generate();

// We pack B matrix ahead of time
// Each thread gets a C block
parallel_for(range(N/BN, K/BK), [&](range r){
    auto wei_off = (r[0] * N + r[1] * BK) * bf16_sz;
    auto pwei_off = (r[0] * K + r[1] * BK) * BN *
bf16_sz;
    tB.execute(weights + wei_off, packedB + pwei_off);
});
```

```
// Each thread gets one C block
parallel_for(range(M/BM, N/BN), [&](range r){
  brg.set_hw_context();
  std::vector<std::pair> A_B;
  A_B.reserve(K/BK);

  // Allocate temp buffers, with thread_local pools
  void *scratch = myalloc(brg.get_scratchpad_size());

  // we set up the arguments for the execution
  for (int k = 0; k < K; k += BK)
    A_B.emplace_back(
      src + (r[0] * BM * K + k) * bf16_sz,
      packedB + (r[1] * BN * K + k * BN) * bf16_sz);

  // we run brgemm ukernel
  auto dst_off = (r[0] * BM * N + r[1] * BN) * bf16_sz;
  brg.execute(A_B, dst + dst_off, scratch);

  brg.release_hw_context();
});
```


Efficiently composable with custom constructs

Thanks

Unified Acceleration Foundation