oneDNN ukernel API

Mourad Gouicem
06/06/2024

oneDNN API abstractions

Abstraction level	oneDNN API
Graph	oneDNN graph API
Complex / simple operators	oneDNN graph API oneDNN primitive API
Custom operators building block	?

Need for custom implementation stems from:

- Easy prototyping with new numerical workflows (e.g. mxfp4, weights dequant, …)
- Easy prototyping with emerging patterns (e.g. sdpa)

Developers are already doing it through:

- Dedicated libraries (e.g. FBGEMM, gemmlowp)
- Dedicated internal abstractions (e.g. indirect GEMM in XNNPack, GEMM in ATen/Eigen/MLAS).

oneDNN already has some internal abstraction for basic building blocks on CPU that could be leveraged for efficient custom implementations

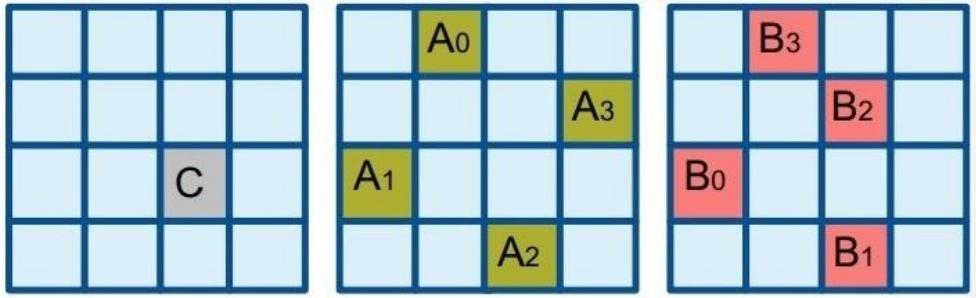
ukernel API design philosophy (CPU)

- Single threaded API, but thread-safe:
  - API is independent of threading runtime
  - can be safely used in parallel region
- No extra abstractions for memory objects or threading runtime
- Memory to memory operations:
  - simpler programming model,.
  - higher customization: user can add custom pre/post-processing, within parallel section
- Single configurable object for each ukernel: simpler API, reduces overheads of object creation
  - Configure computation with setters and finalize
  - Query properties (e.g. scratchpad size, packing type, …)
  - JIT generate executable

References:

- RFC: oneDNN proposal for ukernel APIs
- Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning & HPC Workloads

BRGEMM ukernel



$$C = \beta * C + \alpha \sum_{i=0}^{N-1} A_i * B_i$$

- Computes batch-reduced GEMM operations: Flexible abstraction applicable to Matmul, Convolution, RNNs, …
- Configurable datatypes: only scales supported, custom compensation possible through binary post-op
- Post-ops available for eltwise/binary ops fusions and conversions.

```
struct brgemm {
    // Vanilla version of brgemm with no post-op or destination conversion.
    brgemm(dim_t batch, dim_t M, dim_t N, dim_t K,
            data_type dtA, dim_t ldA,
            data_type dtB, dim_t ldB,
            data_type dtC, dim_t ldC);

    // If true (default), computes C = sum_i A_i * B_i
    // If false, computes C = C + sum_i A_i * B_i
    status_t set_add_C(bool add_C);

    // adds post-operation, and conversion to final destination D.
    status_t set_postops(post_ops &po, data_type dtD, dim_t ldD);

    // scales can be applied before (upconversion) or after the GEMM operation
    status_t set_scales(int a_scale_mask, int b_scale_mask, int d_scale_mask);

    status finalize();
    // separate kernel generation to allow query without jit overhead
    status generate();

    // Queries for expected layouts and temporary memory
    pack_type get_A_pack_type() const; // Not really needed, just for consistency
    pack_type get_B_pack_type() const;
    size_t get_scratchpad_size() const;

    // HW context handling.
    void set_hw_context() const;
    void reset_hw_context() const;
    static void release_hw_context() const;

    // Execution function for the vanilla brgemm variant.
    void execute(const void *A, const void *B,
            const std::vector<std::pair<size_t, size_t>> &A_B_offsets,
            void *C, void *scratch = nullptr);

    // Execution function for the advanced brgemm<> variant
    void execute(const void *A, const void *B,
            const std::vector<std::pair<size_t, size_t>> &A_B_offsets,
            const void *C, void *D, void *scratch = nullptr,
            const attr_params &attr_args = attr_args());
}
```

Reference: [High-Performance Deep Learning via a Single Building Block]

BRGEMM ukernel

- Object configuration need to be finalized before query
- pack_type available to match HW acceleration requirements on data layout
- Option to use scratchpad for ukernel

```
struct brgemm {
    // Vanilla version of brgemm with no post-op or destination conversion.
    brgemm(dim_t batch, dim_t M, dim_t N, dim_t K,
            data_type dtA, dim_t ldA,
            data_type dtB, dim_t ldB,
            data_type dtC, dim_t ldC);

    // If true (default), computes C = sum_i A_i * B_i
    // If false, computes C = C + sum_i A_i * B_i
    status_t set_add_C(bool add_C);

    // adds post-operation, and conversion to final destination D.
    status_t set_postops(post_ops &po, data_type dtD, dim_t ldD);

    // scales can be applied before (upconversion) or after the GEMM operation
    status_t set_scales(int a_scale_mask, int b_scale_mask, int d_scale_mask);

    status finalize();
    // separate kernel generation to allow query without jit overhead
    status generate();

    // Queries for expected layouts and temporary memory
    pack_type get_A_pack_type() const; // Not really needed, just for consistency
    pack_type get_B_pack_type() const;
    size_t get_scratchpad_size() const;

    // HW context handling.
    void set_hw_context() const;
    void reset_hw_context() const;
    static void release_hw_context() const;

    // Execution function for the vanilla brgemm variant.
    void execute(const void *A, const void *B,
            const std::vector<std::pair<size_t, size_t>> &A_B_offsets,
            void *C, void *scratch = nullptr);

    // Execution function for the advanced brgemm<> variant
    void execute(const void *A, const void *B,
            const std::vector<std::pair<size_t, size_t>> &A_B_offsets,
            const void *C, void *D, void *scratch = nullptr,
            const attr_params &attr_args = attr_args());
}
```

BRGEMM ukernel

- Support for HW accelerator state possible (e.g. Intel AMX or ARM SME).
- 2 execution functions to support K-partitioning:
  - One for partial accumulators
  - One for final accumulation with post-op and conversion
- All blocks are passed as pointers:
  - C passed as single pointer to a block
  - A_i, B_i are passed as base_ptr + array of offsets. Allows offset precomputation.

```
struct brgemm {
    // Vanilla version of brgemm with no post-op or destination conversion.
    brgemm(dim_t batch, dim_t M, dim_t N, dim_t K,
            data_type dtA, dim_t ldA,
            data_type dtB, dim_t ldB,
            data_type dtC, dim_t ldC);

    // If true (default), computes C = sum_i A_i * B_i
    // If false, computes C = C + sum_i A_i * B_i
    status_t set_add_C(bool add_C);

    // adds post-operation, and conversion to final destination D.
    status_t set_postops(post_ops &po, data_type dtD, dim_t ldD);

    // scales can be applied before (upconversion) or after the GEMM operation
    status_t set_scales(int a_scale_mask, int b_scale_mask, int d_scale_mask);

    status finalize();
    // separate kernel generation to allow query without jit overhead
    status generate();

    // Queries for expected layouts and temporary memory
    pack_type get_A_pack_type() const; // Not really needed, just for consistency
    pack_type get_B_pack_type() const;
    size_t get_scratchpad_size() const;

    // HW context handling.
    void set_hw_context() const;
    void reset_hw_context() const;
    static void release_hw_context() const;

    // Execution function for the vanilla brgemm variant.
    void execute(const void *A, const void *B,
            const std::vector<std::pair<size_t, size_t>> &A_B_offsets,
            void *C, void *scratch = nullptr);

    // Execution function for the advanced brgemm<> variant
    void execute(const void *A, const void *B,
            const std::vector<std::pair<size_t, size_t>> &A_B_offsets,
            const void *C, void *D, void *scratch = nullptr,
            const attr_params &attr_args = attr_args());
}
```

Transform ukernel

- Support for HW accelerator specific packing requirements
- All execution arguments passed as pointers
- Can be extended to other packing types (e.g. transposition)

```
enum pack_type {
    pack_32;
    pack_64;
    no_pack;
}

struct transform {
    transform(dim_t M, dim_t N,
            data_type dt_src, pack_type tag_src, dim_t ld_src,
            data_type dt_dst, pack_type tag_dst, dim_t ld_dst);
    void finalize();
    void generate();
    void execute(const void *src, void *dst);
}
```

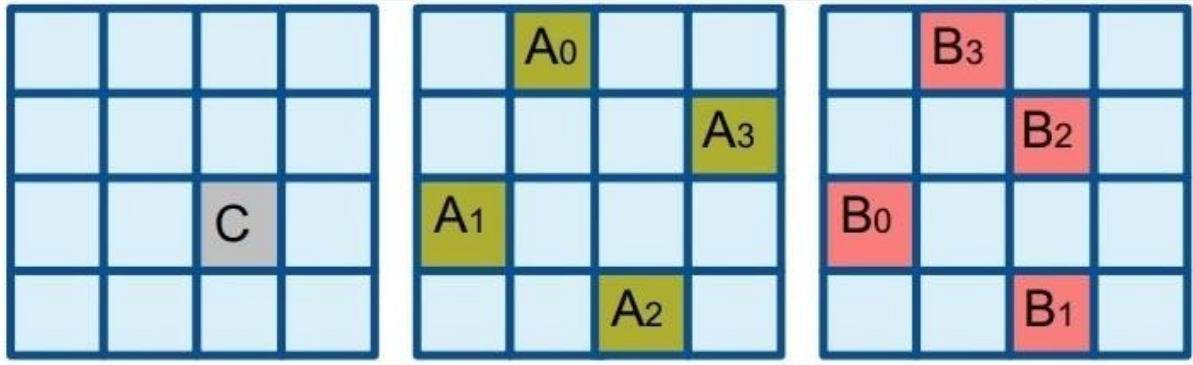
Call for feedback

We need your feedback, please review the RFC!

- Potential user? Please check the proposal matches your target usages
- Potential contributor? Please check the proposal is compatible with your target hardware
- Feel free to comment on the RFC with any question/comment/proposal

Thank you!

BRGEMM ukernel



$$C = \beta * C + \alpha \sum_{i=0}^{N-1} A_i * B_i$$

- Computes batch-reduce GEMM operation: Flexible abstraction applicable to Matmul, Convolution, RNNs…
- Low precision support available through scales, custom compensation through binary post-op
- HW acceleration possible though
  - pack_type management
  - CPU state management APIs.
- All blocks are passed as pointers:
  - C passed as single pointer to a block
  - A_i are passed as base_ptr + array of offsets. Allows precomputed offsets.
  - B_i are passed as base_ptr + array of offsets. Allows precomputed offsets.

Reference: [High-Performance Deep Learning via a Single Building Block]

oneDNN primitive API concepts

Key abstractions

Primitive

- primitive descriptor (lightweight implementation aware descriptor)
- primitive (jitted code)

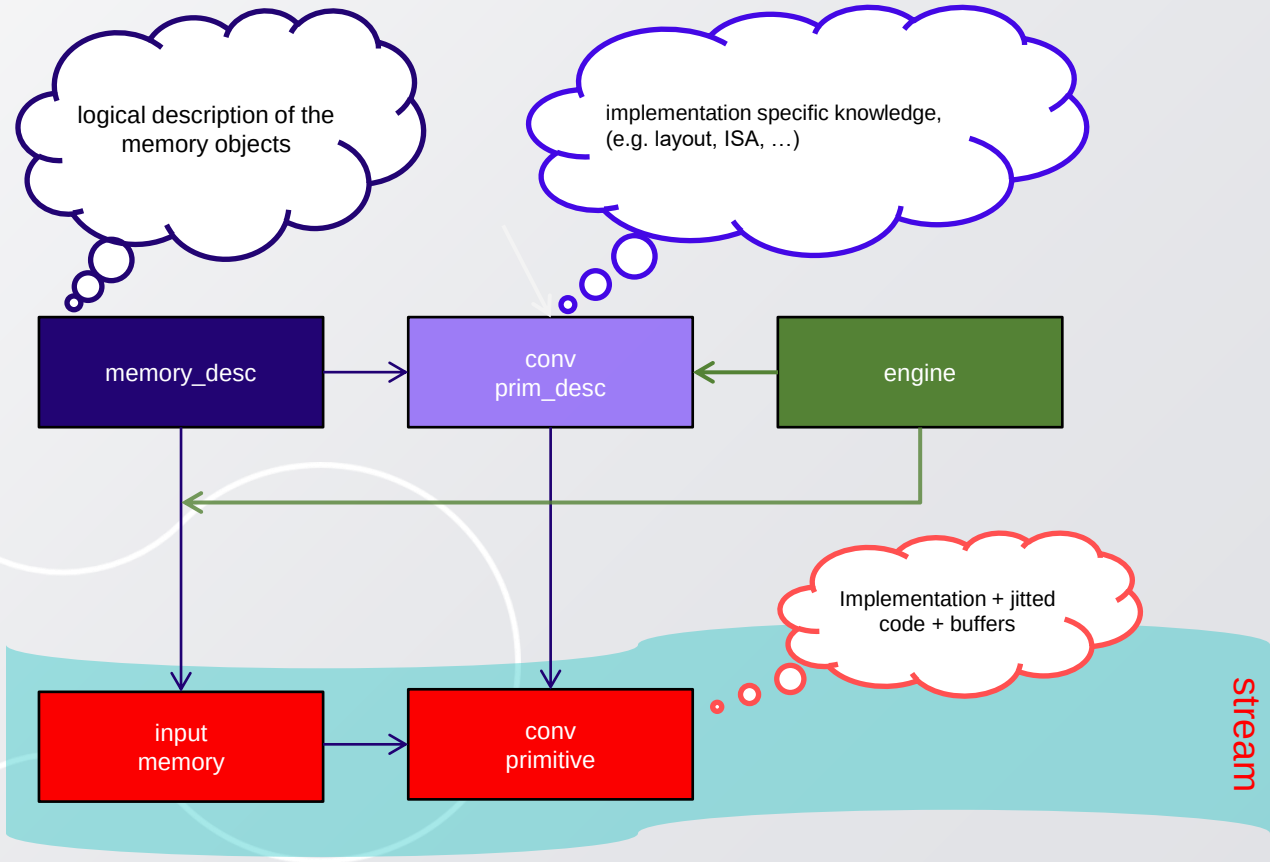
Memory

- memory desc (dims, data type, layout)
- memory (desc + engine, pointer to data)

Engine – execution device

Stream – execution context

Simplified programming model



oneDNN graph API concepts

Key abstractions

Graph

- Ops compose the graph and describe operations
- logical_tensor describe dependencies between ops

Partition

- Represents subgraph that can be run in a single compiled unit
- Must be queried from a graph

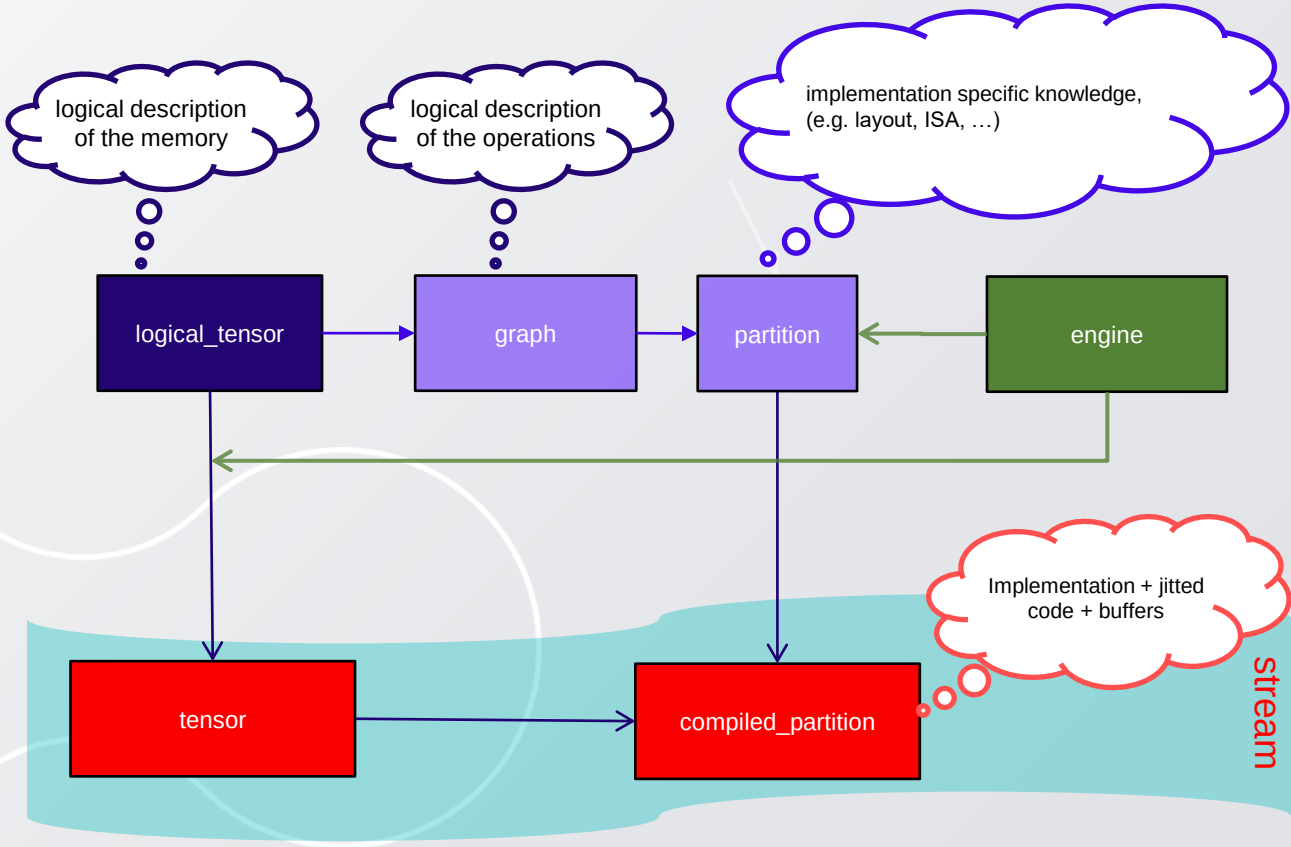
Tensor

- logical_tensor (id, dims, datatype, layout)
- tensor (logical_tensor + engine + pointer to data)

Engine – execution device

Stream – execution context

Simplified programming model

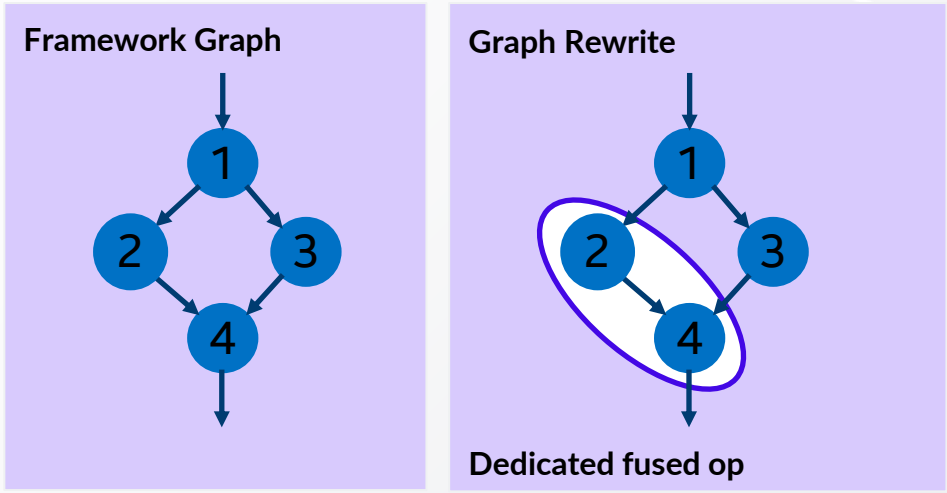


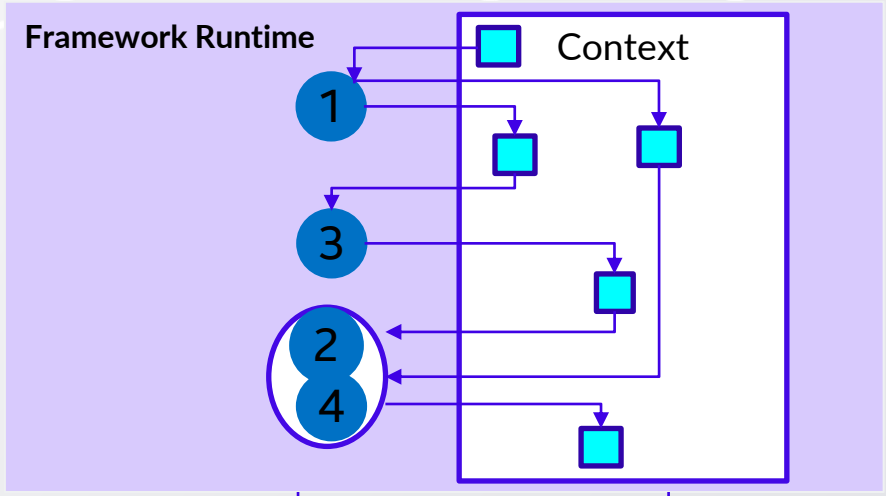
Integration path: fused-op level

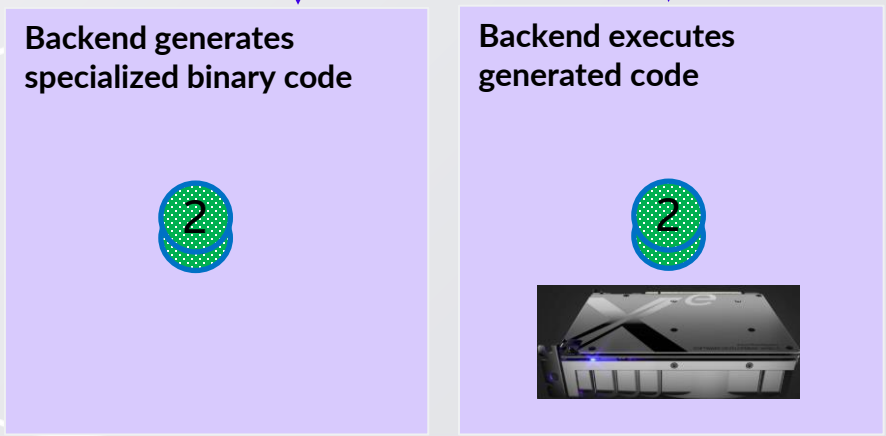
DL Framework

oneDNN API

oneDNN implementation







Integration path: graph level