Appendix

A Generated Wrapper and Kernel Pairs

Here we present a few interesting kernel-wrapper pairs for different operators.

A.1 nn.functional.binary_cross_entropy

```
@triton.jit
def kernel(
    input_ptr,
    target_ptr,
    weight_ptr,
    output_ptr,
    n_elements,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    input = tl.load(input_ptr + offsets, mask=mask, other=0.0)
    target = tl.load(target_ptr + offsets, mask=mask, other=0.0)
    weight = tl.load(weight_ptr + offsets, mask=mask, other=1.0)
    eps = 1e-8
    loss = -target * tl.log(input + eps) - (1 - target) * tl.log(1 - input + eps)
    loss = loss * weight
    tl.store(output_ptr + offsets, loss, mask=mask)

def wrapper(input, target, weight=None, reduction='mean'):
    if input.shape != target.shape:
        raise RuntimeError("input and target must have the same shape")
    if weight is not None:
        if weight.shape != input.shape:
            weight = weight.broadcast_to(input.shape)
        weight = weight.contiguous()
    else:
        weight = torch.ones_like(input, dtype=input.dtype, device=input.device)
    if reduction not in ['none', 'mean', 'sum']:
        raise ValueError("reduction must be 'none', 'mean', or 'sum'")
    output = torch.empty_like(input)
    input = input.contiguous()
    target = target.contiguous()
    n_elements = input.numel()
    if n_elements == 0:
        if reduction == 'none':
            return output
        else:
            return torch.tensor(0.0, dtype=input.dtype, device=input.device)
    BLOCK_SIZE = 1024
    grid = (triton.cdiv(n_elements, BLOCK_SIZE),)
    kernel[grid](
        input,
        target,
        weight,
        output,
        n_elements,
        BLOCK_SIZE=BLOCK_SIZE,
    )
    if reduction == 'none':
        return output
    elif reduction == 'sum':
        return output.sum()
    elif reduction == 'mean':
        return output.mean()
```

A.2 outer

```
@triton.jit
def kernel(
    input_ptr,
    vec2_ptr,
    output_ptr,
    n,
    m,
    input_stride,
    vec2_stride,
    output_stride0,
    output_stride1,
):
    pid = tl.program_id(0)
    if pid >= n:
        return
    input_val = tl.load(input_ptr + pid * input_stride)
    for j in range(m):
        vec2_val = tl.load(vec2_ptr + j * vec2_stride)
        product = input_val * vec2_val
        output_offset = pid * output_stride0 + j * output_stride1
        tl.store(output_ptr + output_offset, product)

def wrapper(input, vec2, *, out=None):
    if input.dim() != 1 or vec2.dim() != 1:
        raise ValueError("Both input and vec2 must be 1D tensors")
    n = input.size(0)
    m = vec2.size(0)
    output_shape = (n, m)
    dtype = input.dtype
    if input.dtype != vec2.dtype:
        pass
    device = input.device
    if out is None:
        output = torch.empty(output_shape, dtype=dtype, device=device)
    else:
        if out.shape != output_shape:
            raise RuntimeError(f"Expected out tensor to have shape {output_shape}, but got {out.shape}")
        if out.dtype != dtype:
            raise RuntimeError(f"Expected out tensor to have dtype {dtype}, but got {out.dtype}")
        if out.device != device:
            raise RuntimeError(f"Expected out tensor to be on device {device}, but got {out.device}")
        output = out
    input_contig = input.contiguous()
    vec2_contig = vec2.contiguous()
    output_contig = output.contiguous()
    input_stride = input_contig.stride(0)
    vec2_stride = vec2_contig.stride(0)
    output_stride0 = output_contig.stride(0)
    output_stride1 = output_contig.stride(1)
    grid = (n,)
    kernel[grid](
        input_contig,
        vec2_contig,
        output_contig,
        n,
        m,
        input_stride,
        vec2_stride,
        output_stride0,
        output_stride1,
    )
    return output
```

A.3 nn.functional.layer_norm

```
@triton.jit
def kernel_mean_var(
   input_ptr,
   mean_ptr,
   var_ptr,
   M,
   N,
   epsilon,
   BLOCK_SIZE: tl.constexpr,
):
   pid = tl.program_id(0)
   if pid >= N:
      return

   sum = 0.0
   sum_sq = 0.0
   for i in range(M):
      x = tl.load(input_ptr + pid * M + i)
      sum += x
      sum_sq += x * x

   mean = sum / M
   var = sum_sq / M - mean * mean + epsilon

   tl.store(mean_ptr + pid, mean)
   tl.store(var_ptr + pid, var)
@triton.jit
def kernel_normalize(
   input_ptr,
   output_ptr,
   mean_ptr,
   var_ptr,
   weight_ptr,
   bias_ptr,
   M,
   N,
   epsilon,
   elementwise_affine,
   BLOCK_SIZE: tl.constexpr,
):
   pid = tl.program_id(0)
   if pid >= N:
      return

   mean = tl.load(mean_ptr + pid)
   var = tl.load(var_ptr + pid)

   for i in range(M):
      x = tl.load(input_ptr + pid * M + i)
      x_float = x.to(tl.float32)
      x_float = x_float - mean
      x_float = x_float / tl.sqrt(var) # Corrected line: removed epsilon here

      if elementwise_affine != 0:
         weight = tl.load(weight_ptr + i).to(tl.float32)
         bias = tl.load(bias_ptr + i).to(tl.float32)
         x_float = x_float * weight + bias

      x = x_float.to(x.dtype)
      tl.store(output_ptr + pid * M + i, x)

def wrapper(input, normalized_shape, weight=None, bias=None, eps=1e-5, elementwise_affine=True):
   if not isinstance(input, torch.Tensor):
      raise TypeError("Input must be a torch.Tensor")
   if not input.is_contiguous():
      input = input.contiguous()

   if isinstance(normalized_shape, int):
      normalized_shape = (normalized_shape,)
   else:
      normalized_shape = tuple(normalized_shape)
   D = len(normalized_shape)
   input_shape = input.shape
   if D > len(input_shape):
      raise ValueError("normalized_shape cannot be larger than input shape")

   M = 1
   for dim in normalized_shape:
      M *= dim
```

```
    N = 1
    for dim in input_shape[:-D]:
        N *= dim

    if N == 0 or M == 0:
        return torch.empty_like(input)

    input_float = input.to(torch.float32)
    input_float = input_float.view(N, M)
    mean = torch.empty(N, dtype=torch.float32, device=input.device)
    var = torch.empty(N, dtype=torch.float32, device=input.device)

    BLOCK_SIZE = 1
    grid = (N,)
    kernel_mean_var[grid](
        input_float,
        mean,
        var,
        M,
        N,
        eps,
        BLOCK_SIZE=BLOCK_SIZE,
    )

    output = torch.empty_like(input)
    output = output.view(N, M)

    if elementwise_affine:
        if weight is None:
            weight = torch.ones(normalized_shape, dtype=input.dtype, device=input.device)
        else:
            weight = weight.contiguous().view(-1)
        if bias is None:
            bias = torch.zeros(normalized_shape, dtype=input.dtype, device=input.device)
        else:
            bias = bias.contiguous().view(-1)
    else:
        weight = torch.ones(M, dtype=input.dtype, device=input.device)
        bias = torch.zeros(M, dtype=input.dtype, device=input.device)

    input_flattened = input.view(N, M)
    kernel_normalize[grid](
        input_flattened,
        output,
        mean,
        var,
        weight,
        bias,
        M,
        N,
        eps,
        1 if elementwise_affine else 0,
        BLOCK_SIZE=BLOCK_SIZE,
    )

    output = output.view(input_shape)
    return output
```

B State Transition Prompts

As TritorX executes its feedback loop, we have different prompts for different scenarios. The initial prompt provides rules and background information, most critically PyTorch docstring(s) explaining both the operator in question and other referenced PyTorch docs or operators, alongside three fully working MTIA Triton reference kernels for `exp`, `argmax`, and `diag`:

```
init_prompt = f"""
Write me a Triton implementation of the {op_name} ATen operator from PyTorch using MTIA's version of Triton. This
     implementation should support the following input dtypes: {str(dtypes)}.

To make it easy to use for me please make sure to call your jitted kernel "kernel" and the wrapper code that
     translates the original signature of {op_name} to the kernel's signature "wrapper".

Multiple kernels are fine if needed, but try to generalize as much as possible and limit the number of kernels.
     Thoroughly analyze the problem and decide if one kernel is enough or if multiple are needed. If you get
     stuck trying to put all logic in one kernel, feel free to create multiple. Note that if you have multiple
     kernels, their names MUST all start with "kernel".

Do NOT use or fallback to the base PyTorch or Triton implementations of these kernels - this is considered
     cheating and is NOT allowed.

Do NOT include any comments or import statements (the required ones will be added while running the kernel - do
     not worry about import errors).

Additionally, for MTIA, we add 32-bit padding to input tensors. Take this into account while you are iterating
     and debugging your kernel.

Think carefully about a correct and efficient implementation and finally output your code in a Python codeblock.
     Please only output a single module's code containing the kernel and wrapper functions.

I'll paste the docstr of ATen's {op_name} for reference, which defines the spec:
{docstring}
{supplemental_docstrings}

\n\n\nFor your reference, I am including a few different types of fully working MTIA Triton implementations of
     ATen operators.

These will be listed in triplets, where each example will have "Operator", "Kernel(s)", and "Wrapper". Analyze
     these to better understand MTIA Triton:\n

<reference kernels>

Please think carefully and output a full implementation of {op_name} in MTIA Triton now!"""
```

If we are starting a new LLM session but have a partially working kernel already, we instead prompt it to *debug* an existing kernel, and provide the current implementation:

```
init_prompt_with_existing_kernel = f"""
Debug a Triton implementation of the {op_name} ATen operator from PyTorch, written in MTIA's version of Triton.
     This implementation should support the following input dtypes: {str(dtypes)}.
...
...
Lastly, here is the work-in-progress implementation of this operator. It has a few issues that I need your help
     debugging, so be sure to analyze it thoroughly:
<current partial implementation>
"""
```

We describe the details of linting more in Appendix D, but the prompt itself is fairly straightforward, simply showing the LLM the lint errors.

Our feedback prompts for compilation errors and correctness errors are more nuanced, and consist of three separate prompts. To begin with, part of our approach to optimize context length includes using an LLM-based summarizer for long compilation logs that often contain the same error message multiple times:

```
summarization_prompt = """
...
...
To recap, return the following information ONLY:
1. The EXACT error message
2. The EXACT code snippet that caused the error, both the exact line and the lines before it
3. The EXACT traceback of the error, if present - do NOT include duplicates
"""
```

This summary is then passed into a simple prompt telling the LLM to evaluate the compilation error and fix the kernel.

When the generated kernel fails a test case due to a correctness error, we provide quite a bit of feedback to the LLM based on the test case to use as context while debugging. We've found that this does not increase the risk of overfitting to a specific test case whilst increasing the success rate:

```
correctness_feedback_prompt = """
...
...
**Summary of the CPU output tensor for the input**:
{str(latest_result.cpu_tensor_results)}

**Summary of the MTIA output tensor for the input**:
{str(latest_result.mtia_tensor_results)}

Additionally, here is the input and output data for the first failing test case. Do NOT overfit your kernel to
     this - remember you need to build a generalized kernel implementation. This is here solely to help you debug.

**INPUT SIGNATURE**:
{str(input_sig)}

**OUTPUT SIGNATURE**:
{str(output_sig)}

**INPUT SHAPE**:
{str(latest_result.shape)}

**INPUT TENSOR**:
{str(latest_result.input_tensor)}

**INPUT ARGS**:
{str(latest_result.input_args)}

**INPUT KWARGS**:
{str(latest_result.input_kwargs)}

Generate the corrected MTIA kernel now:
"""
```

C REASONING TRAJECTORIES

```
You are an expert in generating Triton MTIA (Meta Training and Inference Accelerator) kernels. Write me a Triton implementation of the nn.functional.logsigmoid
ATen operator from PyTorch using MTIA's version of Triton. This implementation should support the following input dtypes: ['bfloat16', 'float16', 'float32'].
...
I'll paste the docstr of ATen's nn.functional.logsigmoid for reference, which defines the spec:
logsigmoid(input) -> Tensor
Applies element-wise :math:`\text{LogSigmoid}(x_i) = \log \left(\frac{1}{1 + \exp(-x_i)}\right)`
See :class:`~torch.nn.LogSigmoid` for more details.
For your reference, I am including a few different types of fully working MTIA Triton implementations of ATen operators.
...
```

```
Okay, I need to implement the logsigmoid function using MTIA Triton. Let's start by understanding the problem. The logsigmoid function applies the element-wise
operation LogSigmoid(x_i) = log(1 / (1 + exp(-x_i))). This is equivalent to -softplus(-x_i), where softplus is the softplus activation function.

First, I should consider the input dtypes: bfloat16, float16, and float32. MTIA Triton might handle these differently, but I'll assume that the kernel can be
written in a way that works for all three types, possibly using type specialization or generic operations.
...
```

```
@triton.jit
def kernel(
   input_ptr,
   output_ptr,
   n_elements,
   BLOCK_SIZE: tl.constexpr,
):
   pid = tl.program_id(0)
   block_start = pid * BLOCK_SIZE
   offsets = block_start + tl.arange(0, BLOCK_SIZE)
   mask = offsets < n_elements

   x = tl.load(input_ptr + offsets, mask=mask)
   exp_neg_x = tl.exp(-x)
   logsigmoid = -tl.loglp(exp_neg_x)
   tl.store(output_ptr + offsets, logsigmoid, mask=mask)

def wrapper(input):
   output = torch.empty_like(input, device=input.device)
   if input.numel() == 0:
      return output
   input_contig = input.contiguous().view(-1)
   output_contig = torch.empty_like(input_contig, device=input.device)
   n_elements = input.numel()
   BLOCK_SIZE = 128
   grid = (triton.cdiv(n_elements, BLOCK_SIZE),)
   kernel[grid](
      input_contig,
      output_contig,
      n_elements,
      BLOCK_SIZE=BLOCK_SIZE,
   )
   return output_contig.view(input.shape)
```

```
Your previous MTIA kernel implementation failed to pass the linter. Please analyze the lint error and provide a corrected version. Remember to adhere to the
original output guidelines.
**Lint Error**:
Found 1 linting violation(s):
[module_restrictions] Forbidden tl module usage: tl.loglp (line 15)
Details: Allowed tl functions: tl.abs, tl.add, tl.advance, ...
...
Generate the corrected MTIA kernel now
```

```
Okay, the linter is complaining about the use of tl.loglp in the kernel. Let me check the allowed tl functions. The error message lists the allowed functions,
and tl.loglp is not among them.
...
```

```
...
```

```
Your MTIA kernel implementation failed to compile. Please analyze the compilation error and provide a corrected version. Remember to adhere to the original
output guidlines.
**Compilation Error**:
...
ValueError: Expected dtype ['fp32', 'fp64'] but got fp16
...
Generate the corrected MTIA kernel now:
```

```
...
```

```
def kernel(
   input_ptr,
   output_ptr,
   n_elements,
   BLOCK_SIZE: tl.constexpr,
):
   pid = tl.program_id(0)
   block_start = pid * BLOCK_SIZE
   offsets = block_start + tl.arange(0, BLOCK_SIZE)
   mask = offsets < n_elements

   x = tl.load(input_ptr + offsets, mask=mask)
   x_float32 = tl.cast(x, tl.float32)
   exp_neg_x = tl.exp(-x_float32)
   logsigmoid_float32 = -tl.log(1 + exp_neg_x)
   logsigmoid = tl.cast(logsigmoid_float32, x.dtype)
   tl.store(output_ptr + offsets, logsigmoid, mask=mask)

def wrapper(input):
   ...
```

Figure C.4. Example of TritorX writing a kernel-wrapper pair for the `nn.functional.logsigmoid` operator in 3 LLM calls after receiving the linter and compiler feedback. The reasoning steps are shortened and some details are omitted due to space constraints. Initial prompt in green, LLM reasoning in blue, LLM kernel-wrapper generation in purple, and TritorX feedback in orange.

```
You are an expert in generating Triton MTIA (Meta Training and Inference Accelerator) kernels. Write me a Triton implementation of the
nn.functional.channel_shuffle ATen operator from PyTorch using MTIA's version of Triton. This implementation should support the following input dtypes:
['bfloat16', 'float16', 'float32', 'int32', 'int64'].
...
I'll paste the docstr of ATen's nn.functional.channel_shuffle for reference, which defines the spec:
channel_shuffle(input, groups) -> Tensor
Divide the channels in a tensor of shape :math:`(*, C , H, W)`
into g groups and rearrange them as :math:`(*, C \frac g, g, H, W)`,
while keeping the original tensor shape.
...
```

```
Okay, so I need to implement the nn.functional.channel_shuffle operator using MTIA Triton. Let me start by understanding what this operator does.
...
```

```
@triton.jit
def kernel(
   input_ptr,
   output_ptr,
   ...
):
   ...
      values = tl.load(input_ptr + input_offsets, mask=mask)
      tl.store(output_ptr + output_offsets, values, mask=mask)

def wrapper(input, groups):
   ...
```

```
Your MTIA kernel implementation failed to compile. Please analyze the compilation error and provide a corrected version. Remember to adhere to the original
output guidlines.
**Compilation Error**:
Compilation log with error:
error: Scatter stores are disabled by default. Please set the "enable_scatter_stores" flag or revisit the algorithm to avoid this unsafe pattern.
error: failed to legalize operation 'tts.scatter' that was explicitly marked illegal
...
Generate the corrected MTIA kernel now
```

```
Okay, the compilation error is due to scatter stores being disabled by default in Triton. The error message suggests that the kernel is using a scatter store
operation, which is not allowed unless explicitly enabled.
Looking at the kernel code, the issue arises from the line ... This is a scatter operation because each thread is writing to potentially different,
non-contiguous memory locations based on ...
To fix this, we need to avoid scatter stores. One way to do this is to change the approach so that each thread writes to a contiguous block of memory.
...
```

```
...
```

```
While running tests, your MTIA kernel implementation failed to produce the correct output (wrt a CPU implementation, which is our source of truth).
Note that it compiled successfully. It just failed to produce the correct output for a test case.
** Please analyze the differences between the CPU output and your MTIA kernel's output and provide a corrected version. **
**Summary of the CPU output tensor for the input**:
...
**Summary of the MTIA output tensor for the input**:
...
Generate the corrected MTIA kernel now
```

```
Let me analyze the differences between the expected CPU output and the MTIA kernel's output to find the issue.
First, I'll compare the input and output tensors. Looking at the MTIA output, there are zero values in some positions where the CPU output has non-zero values.
...
```

```
...
```

```
The provided MTIA kernel implementation compiled but had a PE crash on MTIA hardware.
This is often caused by memory access errors. Please analyze the coredump and provide a corrected version.
...
```

```
Okay, so the MTIA kernel implementation is crashing due to a machine external interrupt, likely caused by a memory access error. The root cause is probably an
out-of-bounds access or incorrect memory alignment. Let's think about possible causes.
...
```

```
...
```

```
Your MTIA kernel implementation failed to compile. Please analyze the compilation error and provide a corrected version. Remember to adhere to the original
output guidlines.
**Compilation Error**:
...
ValueError: arange's arguments must be of type tl.constexpr
...
```

```
...
```

```
@triton.jit
def kernel(
   input_ptr,
   output_ptr,
   ...
   BLOCK_SIZE: tl.constexpr,
):
   ...

   # Process each position and group to avoid scatter stores
   for pos in range(0, k):
      for group in range(0, g):
         c_orig = group * k + pos
         c_new = pos * g + group

         input_offset = input_base + c_orig * stride_c
         output_offset = output_base + c_new * stride_c

         value = tl.load(input_ptr + input_offset)
         tl.store(output_ptr + output_offset, value)

def wrapper(input, groups):
...
```

Figure C.5. Example of TritorX writing a kernel-wrapper pair for the `nn.functional.channel-shuffle` operator in 14 LLM calls. The LLM has to incorporate the feedback from the `scatter-store` compilation error (hardware restriction) and the discrepancy between the MTIA test output compared to the CPU implementation, as well as address other compilation errors and hardware crashes.

D TRITON MTIA LINTER

The linter is a rule-based, static analysis tool that combines Python's AST parsing with regex matching per rule. Several of these rules are formatting restrictions to make the output compatible with our execution system, such as disallowing import statements and enforcing a specific output format.

Other rules are more concerned with what modules are being used and where. We have allowlists on several key modules, such as `tl` (`triton.language`), and `torch`. Anything from one of these modules not found in such an allowlist is considered a linter violation:

```yaml
module_restrictions:
  modules:
    - module_name: "tl"
      allowed_functions:
        - "tl.load"
        - "tl.store"
        - "tl.arange"
        # ... 200+ allowed Triton MTIA operations
    - module_name: "torch"
      allowed_functions:
        - "torch.empty"
        - "torch.zeros"
        # ... tensor allocation/reshaping only
```

This highlights exactly which aspects of upstream Triton are compatible with MTIA.

Additionally, we have *scope restrictions* on modules, such as allowing `tl` only in the kernel, not in the wrapper:

```yaml
module_scope_restrictions:
  restrictions:
    - module: "tl"
      allowed_scope_patterns: ["^kernel.*"] # tl.* only inside kernel functions
```

As part of our anti-cheating efforts, we enforce several rules. To prevent moving tensors back to CPU or attempting to move tensors to CUDA:

```yaml
forbidden_tensor_methods:
  enabled: true
  description: "Prohibit tensor methods that move data between devices (CPU/CUDA
      transfers)"
  forbidden_methods:
    - "cpu" # tensor.cpu() - moves tensor to CPU
    - "cuda" # tensor.cuda() - moves tensor to CUDA

forbidden_function_arguments:
  enabled: true
  description: "Prohibit specific argument values in function calls"
  restrictions:
    # Forbid explicit CPU/CUDA device specifications in torch.device()
    - function: "torch.device"
      forbidden_string_args:
        - "cpu"
        - "cuda"
```

And to prevent workarounds such as `eval` or `exec`:

```yaml
# Rule to ban dangerous built-in functions that enable dynamic code execution
forbidden_functions:
  enabled: true
```

```
  description: "Prohibit built-in functions that enable dynamic code execution"
  forbidden_functions:
    - "eval" # Evaluates Python expressions from strings
    - "exec" # Executes Python code from strings
    - "compile" # Compiles Python code from strings (used with exec)
```

E MTIA ARCHITECTURE DIAGRAM

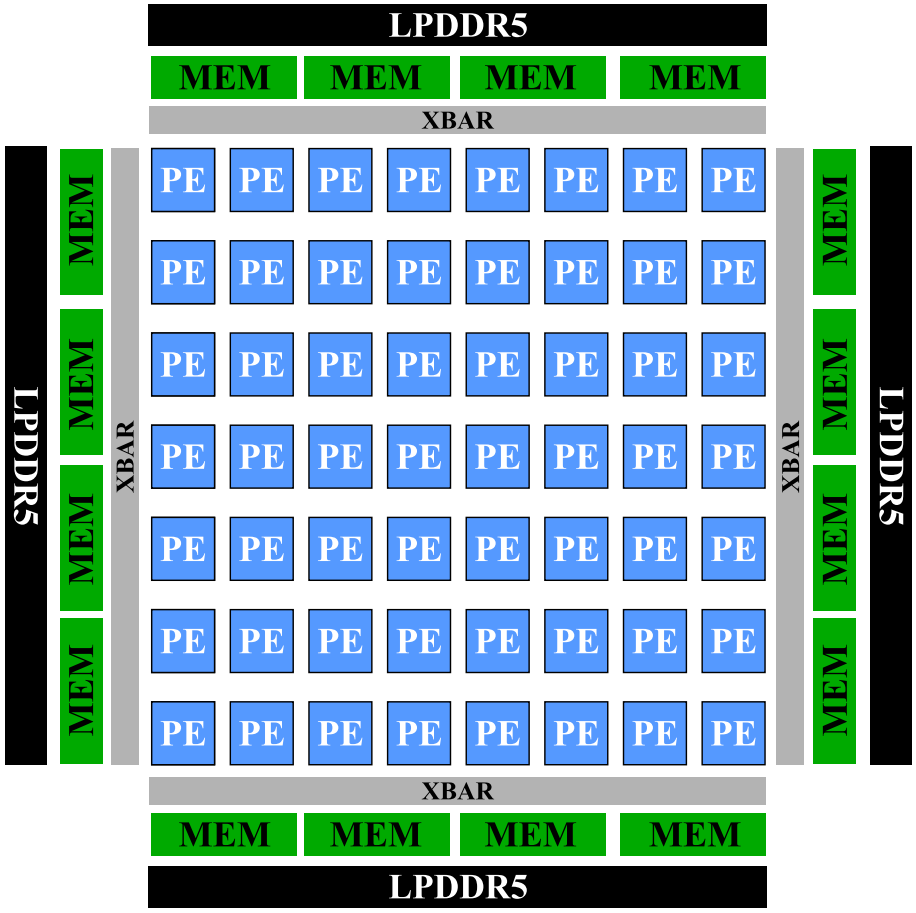

Figure E.6. MTIA's architecture overview. The core kernel computation is performed by a grid of process elements (PEs) which consist of a scalar core, a vector core, and special function units.