Triton Compiler for Intel GPUs

Ettore Tiotto – Intel Principal Engineer

AI Software @ Intel

intel®

Notices and Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Results have been estimated or simulated.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

All product plans and roadmaps are subject to change without notice.

Statements in this document that refer to future plans or expectations are forward-looking statements. These statements are based on current expectations and involve many risks and uncertainties that could cause actual results to differ materially from those expressed or implied in such statements. For more information on the factors that could cause actual results to differ materially, see our most recent earnings release and SEC filings at www.intc.com.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.

Agenda

- Introduction to Triton
- Triton for Intel GPUs
- Optimizations
- Performance
- Summary

Introduction to Triton

What is Triton ?

- Open-source DSL for writing Deep Learning kernels by OpenAI
- Adopted by PyTorch/Inductor as a backend to generate kernels on GPUs
- Positioned by OpenAI as an embedded language to write performant **portable** DL kernels in a Pythonic way

Allows **non-experts** to write **fast** custom and **extendable** kernels.

Sources of input

- Handwriting Triton kernels with Triton operations in Python
- The output of TorchInductor from PyTorch models

User Python Script: ***add+relu***

```
import torch
def fn(a, b):
    return torch.relu(a + b)

a = torch.randn([128, 256], device="xpu")
b = torch.randn([128, 256], device="xpu")
fn_opt = torch.compile(fn, backend="inductor")
fn_opt(a, b)
```

TorchInductor →

Triton Kernel: ***add+relu***

```
import triton
import triton.language as tl
@triton.jit
def triton_(in_ptr0, in_ptr1, out_ptr0, xnumel, XBLOCK : tl.constexpr):
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + (x0), xmask)
    tmp1 = tl.load(in_ptr1 + (x0), xmask)
    tmp2 = tmp0 + tmp1
    mask = 0 > tmp2
    tmp3 = tl.where(mask, 0, tmp2)
    tl.store(out_ptr0 + (x0), tmp3, xmask)
```

Triton roadmap in PyTorch: 8/14/2025
[Meta 2H 2025 Pytorch Roadmap]

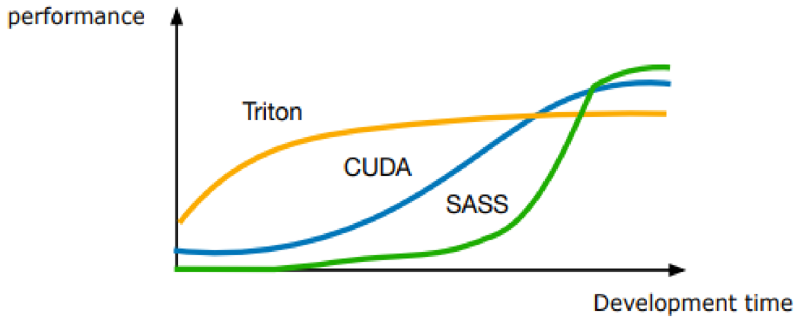
Introduction to Triton

Division of Responsibility

- CUDA: full control, at the expense of productivity (requires expert knowledge of the target GPU architecture), non portable across vendors
- Torch: easy to use, abstracts away GPU HW characteristic, at the expense of performance
- Triton: meet in the middle, abstracts complex GPU concepts, but leaves control to the user on the algorithm and tuning grid tuning

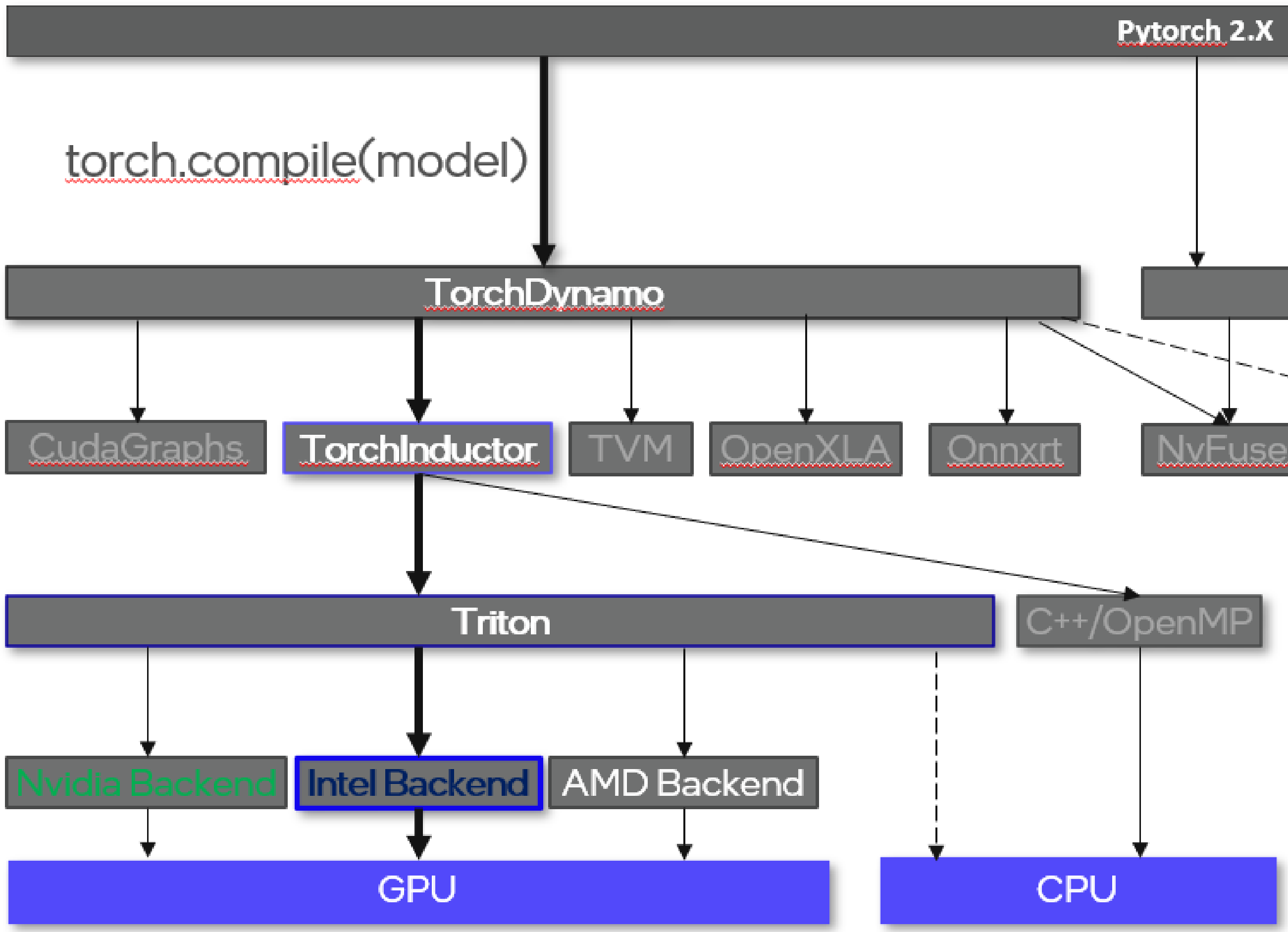
	CUDA	Triton	Torch Op
Algorithm	User	User	Compiler
Shared memory	User	Compiler	Compiler
Barriers	User	Compiler	Compiler
Distribution to blocks	User	User	Compiler
Grid size	User	User	Compiler
Distribution to Warps/threads	User	Compiler	Compiler
Tensor Core usage	User	Compiler	Compiler
Coalescing	User	Compiler	Compiler
Intermediate data layout	User	Compiler	Compiler
Workgroup size	User	User	Compiler

Time to performance



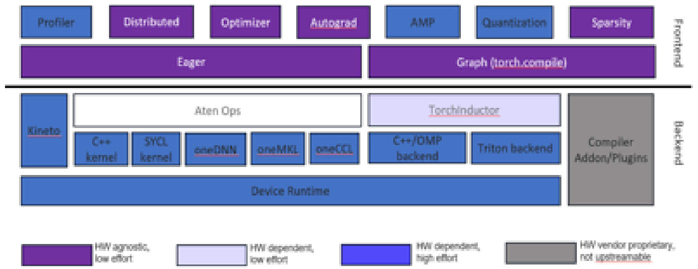
Reference: https://www.youtube.com/watch?v=AtbnRIzpwho

Triton in the PyTorch Ecosystem

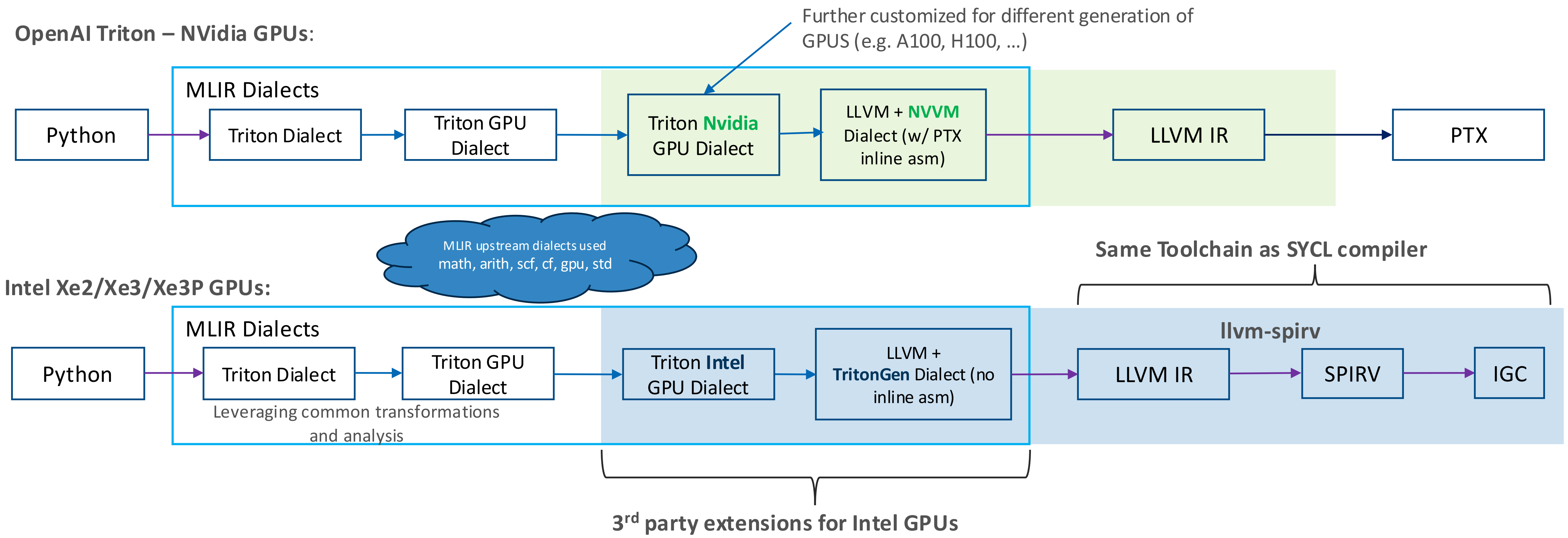


Fit into existing default PyTorch pipelines and provide a **transparent backend** for running Triton kernels on Intel GPUs (both inference and training)

SW architecture – PyTorch Core



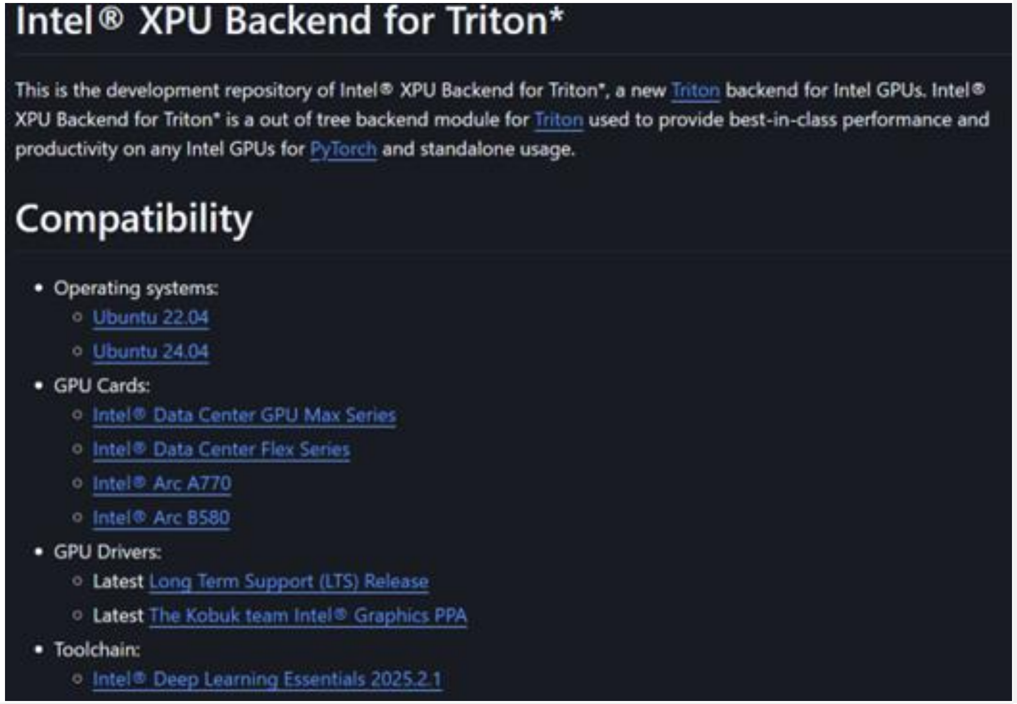
Triton for Intel GPU: compiler architecture



- Allows reuse of OpenAI Triton FrontEnd, and test infrastructure (with customizations)
- Developed an Intel BE for current generation GPUs (e.g. PVC, BMG, LNL) as a "third_party" extension, support for upcoming Xe3P architecture GPUs (e.g. Crescent Island) is underway
- Added target specific optimization pipeline to exploit key HW features of Intel Xe2/Xe3P GPUs

Triton for Intel GPUs: part of PyTorch

- Open-Source development on Github:
  - [intel/intel-xpu-backend-for-triton: OpenAI Triton backend for Intel® GPUs]
- Official Triton’s GPU backend integration in PyTorch launched last year
  - [PyTorch 2.4 support for Intel® GPU Acceleration of AI Workloads]
- Extended OpenAI Triton to add Windows support for Intel Client GPUs (not available upstream)
- Production Quality implementation:
  - > 20k test cases running including microbenchmarks
  - Tracking performance for several key benchmarks



Intel® XPU Backend for Triton*

This is the development repository of Intel® XPU Backend for Triton*, a new Triton backend for Intel GPUs. Intel® XPU Backend for Triton* is a out of tree backend module for Triton used to provide best-in-class performance and productivity on any Intel GPUs for PyTorch and standalone usage.

Compatibility

- Operating systems:
  - Ubuntu 22.04
  - Ubuntu 24.04
- GPU Cards:
  - Intel® Data Center GPU Max Series
  - Intel® Data Center Flex Series
  - Intel® Arc A770
  - Intel® Arc B580
- GPU Drivers:
  - Latest Long Term Support (LTS) Release
  - Latest The Kobuk team Intel® Graphics PPA
- Toolchain:
  - Intel® Deep Learning Essentials 2025.2.1

Triton: Intel Xe2 key Architecture Features

Xe2 vector engine

- SIMD16 native ALUs
  - Support for SIMD16 and SIMD32 ops
- Matrix Extensions (XMX)
  - Systolic Array for efficient matrix-matrix multiply
  - Requires operands to be in lay out in registers
  - HW has efficient instruction to load matrix block from memory to registers
  - HW supports prefetching operands into cache

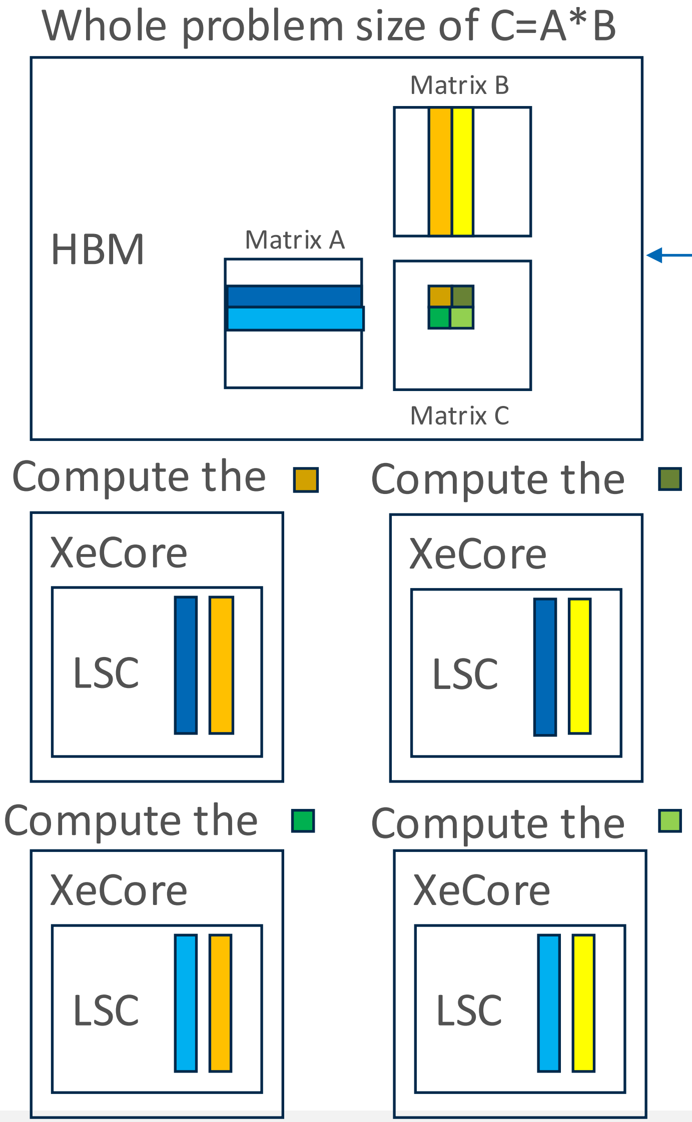
Compiler Exploitation

- Intel Triton BE builds on top of OpenAI optimization infrastructure
- Exploits XMX engine for efficient GEMM operations
- Prefetching and loading of operands blocks via HW instructions
- Significant enhancements to layout conversion removal transformations
- Software pipelining

GEMM like Kernel Tiling in Triton XPU

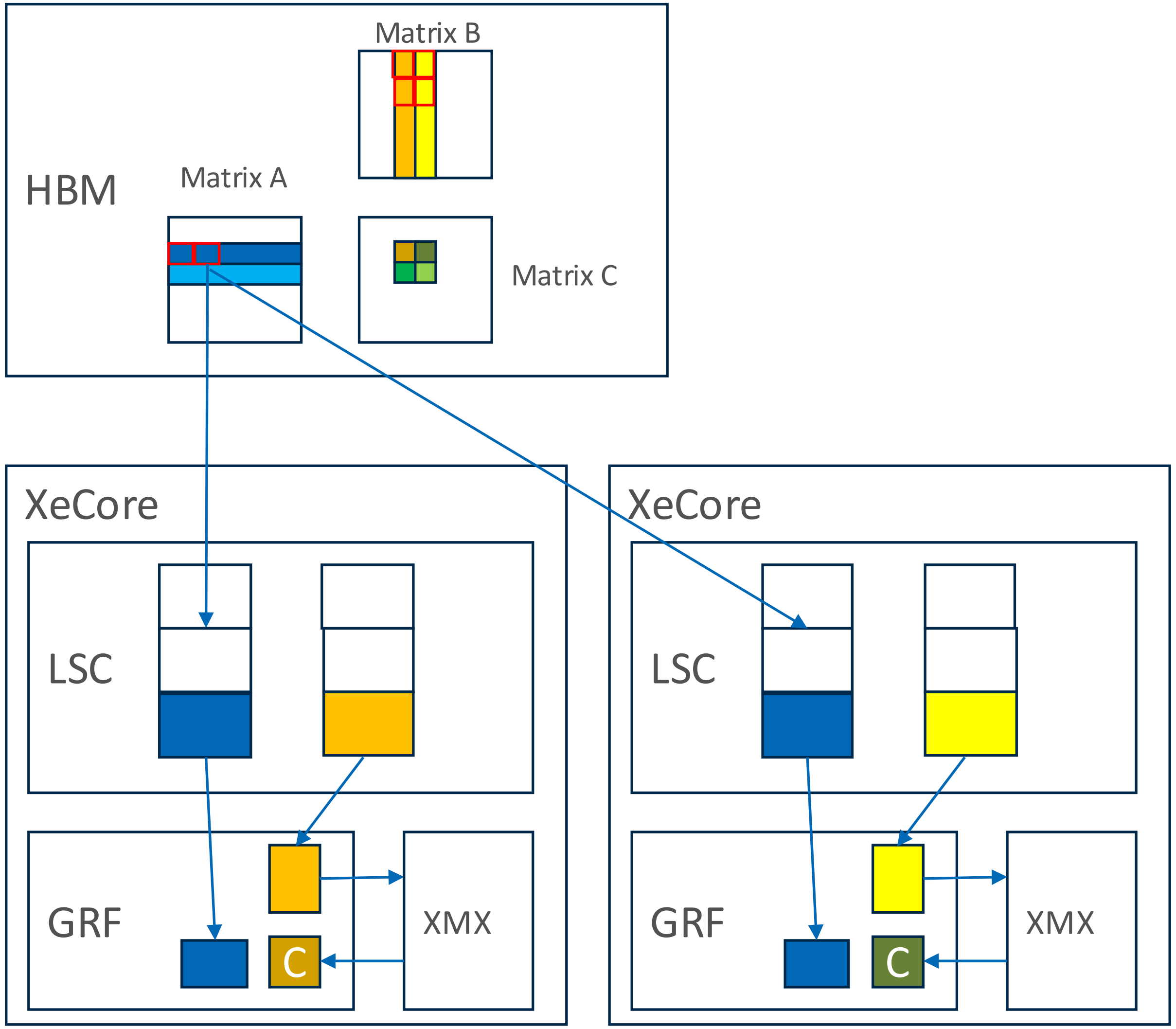
- Architectural characteristics Xe2-Xe3(p) GPUs:
  - The LSC is not shareable across XeCores. Matrices A and B must be loaded redundantly from HBM to LSC which increase the traffic in HBM-LSC bus.
  - GRFs are not shareable across physical threads in the same EU. Matrices A and B in a GEMM must be loaded redundantly from LSC to registers, therefore increasing traffic in the LSC-register bus.
  - There is no direct path (synchronous or asynchronous) to move the data from global memory to share local memory(SLM).
    Because of this limitation, SLM cannot be used effectively.
- Mitigation:
  - HW provides **2D block Input/Output primitives** to load matrices tile directly (HBM->GRFs)
  - HW provides **asycn prefetch ops** to prefetch data from global memory to LSC.

GEMM like Kernel Tiling in Triton XPU



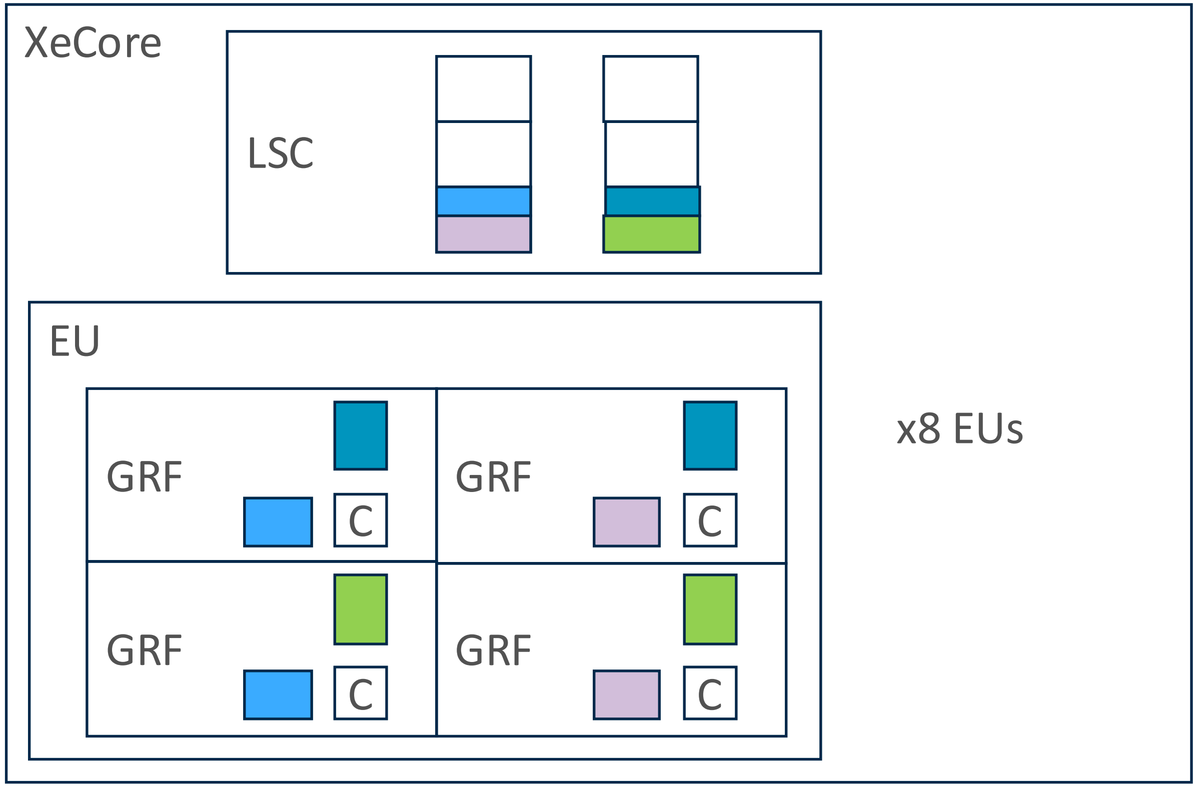
- GEMM tiling strategy:
  - Initial tiling is done algorithmically by the Triton kernel producer. The tiling pattern is illustrated on the left. Each tile in matrix C is computed by a workgroup.
    Tile computations are done in parallel.
    - There are different algorithms to improve the load-balance and cache locality, e.g., grouped tile, split-k, stream-k etc.
  - This strategy has the drawback of increasing traffic required to load matrices A and B
    - LSC is not shareable cross XeCore. A and B tiles need to be loaded from HBM to LSC redundantly, increasing traffic in HBM-LSC bus.
    - A * (Number of tiling on N) + B * (Number of tiling on M)

GEMM like Kernel Tiling in Triton XPU



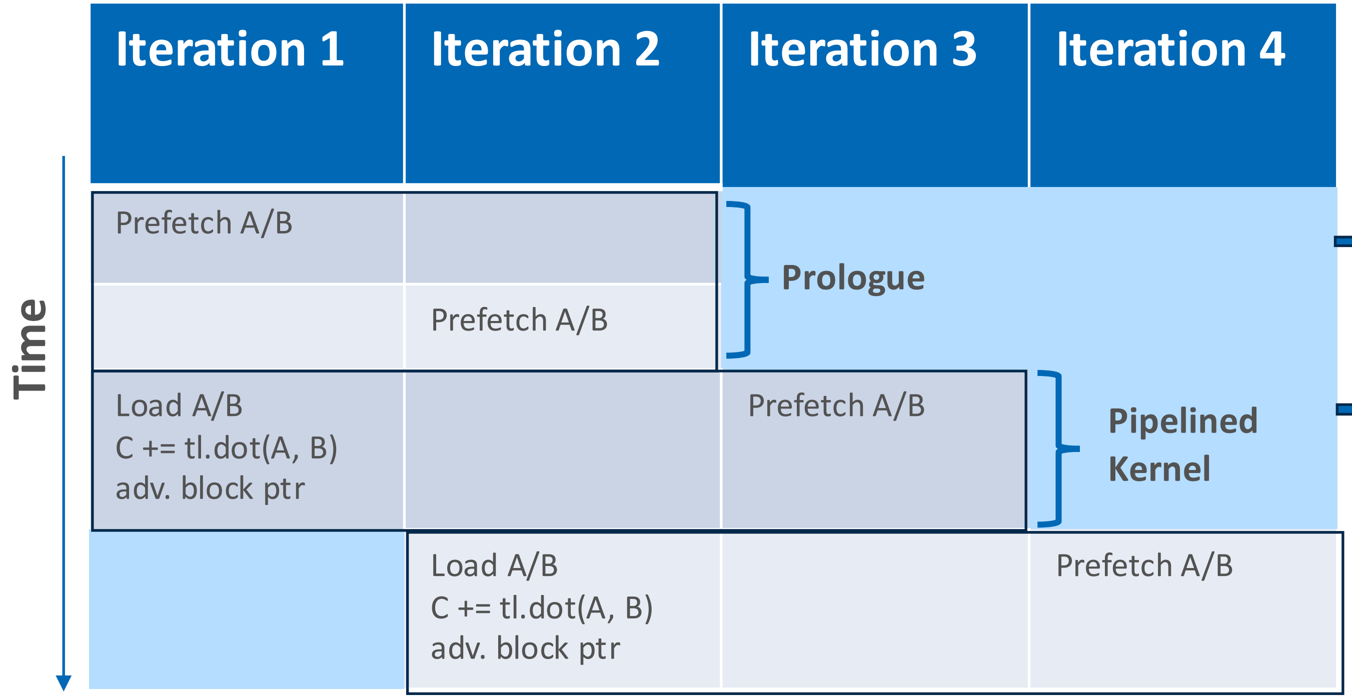
- The Triton XPU backend for Xe2-Xe3p implements loop pipelining, to overlap memory operations with XMX computation:
  - hides memory accessing latency
  - the total number of bytes that needs to be loaded is unchanged
  - As heuristic, if there are more XeCores in the chip, then more bandwidth in GLB-LSC bus is required to saturate the XMX engine as matrix A and B are duplicated in XeCores.

GEMM like Kernel Tiling in Triton XPU



- GEMM tiling is recursive:
  - The 2nd round of tiling is performed by the Triton XPU compiler: uses inner-product to tile the smaller block per work group.
  - Matrices A and B tiles are still loaded redundantly by each sub-group in EU.
  - Instruction scheduling (IGC) is critical to reduce the stall time in waiting for data from LSC.
  - The increased number of the size:
    - A * (Number of tiling on N) + B * (Number of tiling on M)

Reusing Software Pipeline Pass for Prefetching




```
%18 = tt.make_tensor_ptr %arg0, [%15, %16], [%17, %c1_i64], [%14, %c0_i32]
%22 = tt.make_tensor_ptr %arg1, [%16, %20], [%21, %c1_i64], [%c0_i32, %19]
triton_intel_gpu.prefetch %18
triton_intel_gpu.prefetch %22
%23 = tt.advance %18, [%c0_i32, %c32_i32]
%24 = tt.advance %22, [%c32_i32, %c0_i32]
triton_intel_gpu.prefetch %23
triton_intel_gpu.prefetch %24

%25:5 = scf.for %arg9 = %c0_i32 to %arg5 step %c32_i32 iter_args(%arg10 = %cst, ...)
  %29 = tt.advance %arg11, [%c0_i32, %c32_i32]
  %30 = tt.advance %arg12, [%c32_i32, %c0_i32]
  triton_intel_gpu.prefetch %29
  triton_intel_gpu.prefetch %30
  %31 = tt.load %arg13
  %32 = tt.load %arg14
  %33 = tt.dot %31, %32, %arg10
  scf.yield %33, %29, %30, %arg11, %arg12
}
```

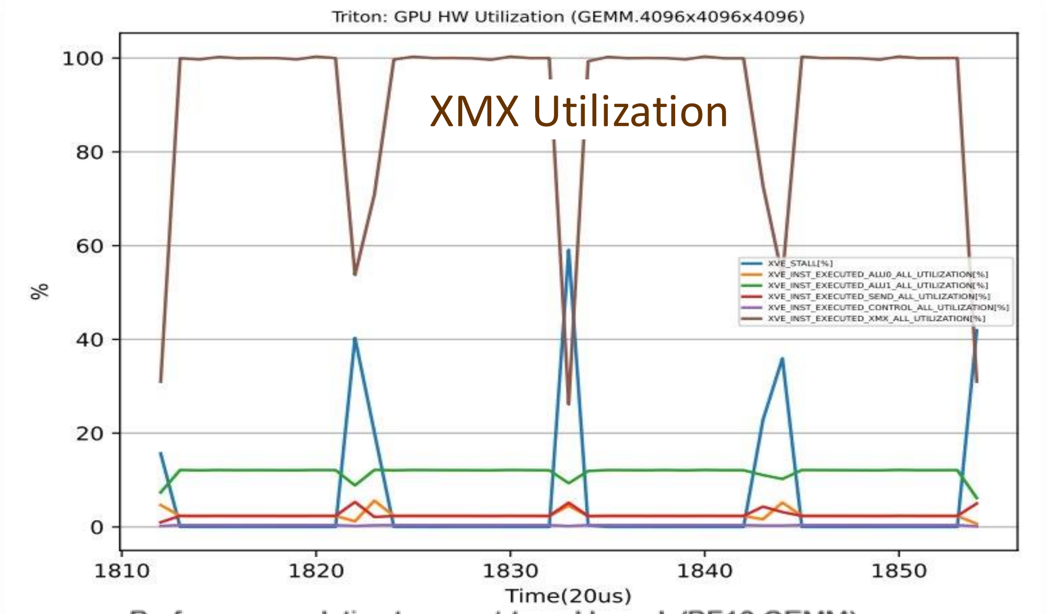
- Set up stages: prefetch, load/dot/block_ptr update
- Reuse triton passes to get prologue and pipelined kernel
- Customization: added a *prefetch* operation in the *TritonIntelGPU* dialect to prefetch to cache instead of shared memory
- Prefetch distance is tunable, trade-off between cache usage vs. latency hiding

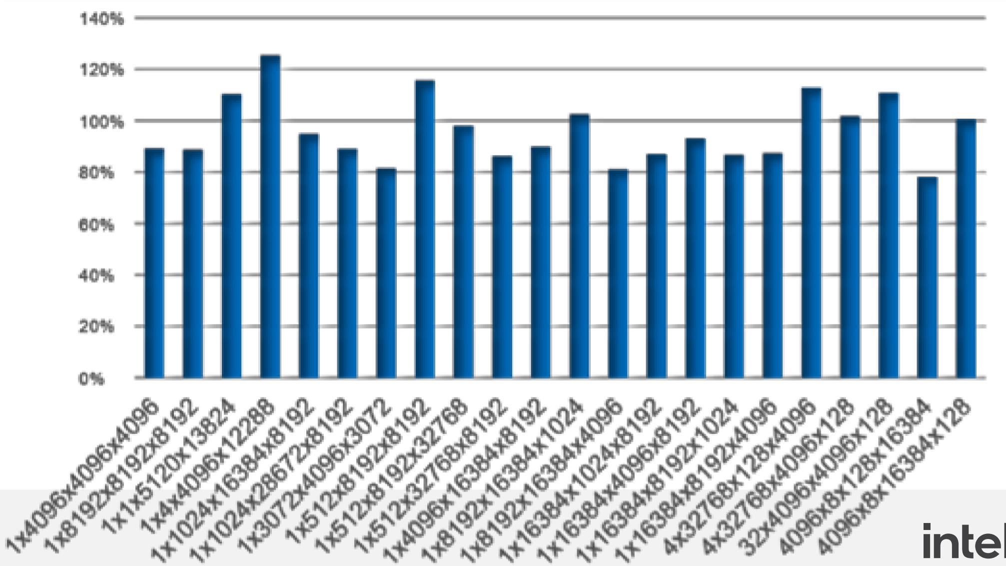
HW Mapping

- triton_intel_gpu.prefetch ➔ 2D Block Prefetches
- tt.load ➔ 2D Block Loads
- tt.dot ➔ DPAS (Dot Product Accumulate Systolic
  - uses XMX engine)

Prefetching: HW Utilization (GEMM)

- Software Pipelining pass adapted to perform prefetching (prefetching distance tunable via Triton's autotuning infrastructure)
- HW utilization measured on Intel Data Center Max GPU 1550, achieved high XMX peak (brown line), indication GEMM operands available when required
- Utilization graph approximates XMX utilization for Intel's tuned library implementation





GEMM kernel: implementations

Using default pointers (top picture)

- tensor<MxKx!tt.ptr<f16>>
- Pointers can point to non-contiguous memory
- Loads are masked

```
accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
    a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
    b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
    accumulator = tl.dot(a, b, accumulator)
    a_ptrs += BLOCK_SIZE_K * stride_ak
    b_ptrs += BLOCK_SIZE_K * stride_bk

c = accumulator.to(tl.float32)

offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
c_ptrs = c_ptr + stride_cm * \
    offs_cm[:, None] + stride_cn * offs_cn[None, :]
c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
tl.store(c_ptrs, c, mask=c_mask)
```

Using tensor descriptors (bottom picture)

- base ptr + strides + block shape
- Like a structure pointer
- Dot operation operand are loaded as 2-dim blocks
- Loads are unmasked

```
a_desc = tl.make_tensor_descriptor(base=a_ptr, shape=(M, K), strides=(stride_am, stride_ak),
                                   block_shape=(BLOCK_SIZE_M, BLOCK_SIZE_K))
b_desc = tl.make_tensor_descriptor(base=b_ptr, shape=(K, N), strides=(stride_bk, stride_bn),
                                   block_shape=(BLOCK_SIZE_K, BLOCK_SIZE_N))

accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
off_k = 0
for _ in range(0, K, BLOCK_SIZE_K):
    a = a_desc.load([pid_m * BLOCK_SIZE_M, off_k])
    b = b_desc.load([off_k, pid_n * BLOCK_SIZE_N])
    accumulator += tl.dot(a, b)
    off_k += BLOCK_SIZE_K
c = accumulator.to(tl.float32)

c_desc = tl.make_tensor_descriptor(base=c_ptr, shape=(M, N), strides=(stride_cm, stride_cn),
                                   block_shape=(BLOCK_SIZE_M, BLOCK_SIZE_N))
c_desc.store([pid_m * BLOCK_SIZE_M, pid_n * BLOCK_SIZE_N], c)
```

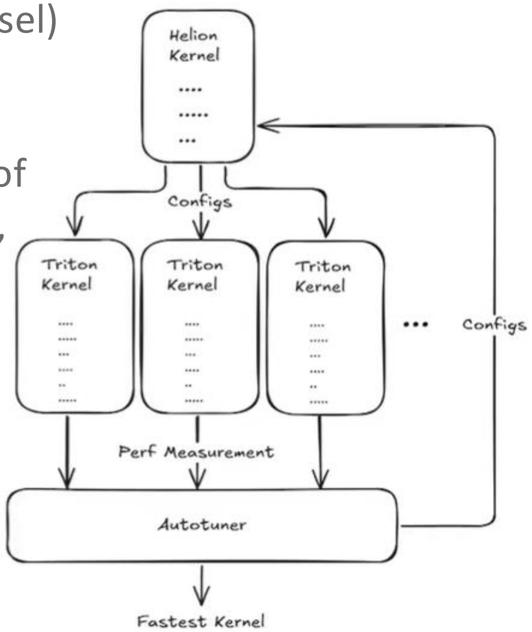
GEMM kernel: performance implications

Performance Implications

- Tensor descriptor loads can be lowered **to efficient HW instructions to load a 2-dim block of data (**Intel Xe2/Xe3/Xe3P architectures)
- Tensor of pointers:
    - compiler needs to prove ptrs point to contiguous memory → **not always possible**
    - Masked loads → loop versioning

[Triton Helium](): designed to generate optimal code for a given target GPU architecture (Jason Ansel)

- can generate kernels that use different language features
- has support for different kind of indexing (tensor_desc, pointer, block_ptr)
- searches metaparameters configurations (block sizes, num_warps, prefetch depth, etc…)




```python
accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
    a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
    b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
    accumulator = tl.dot(a, b, accumulator)
    a_ptrs += BLOCK_SIZE_K * stride_ak
    b_ptrs += BLOCK_SIZE_K * stride_bk

c = accumulator.to(tl.float32)

offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
c_ptrs = c_ptr + stride_cm * \
    offs_cm[:, None] + stride_cn * offs_cn[None, :]
c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
tl.store(c_ptrs, c, mask=c_mask)
```

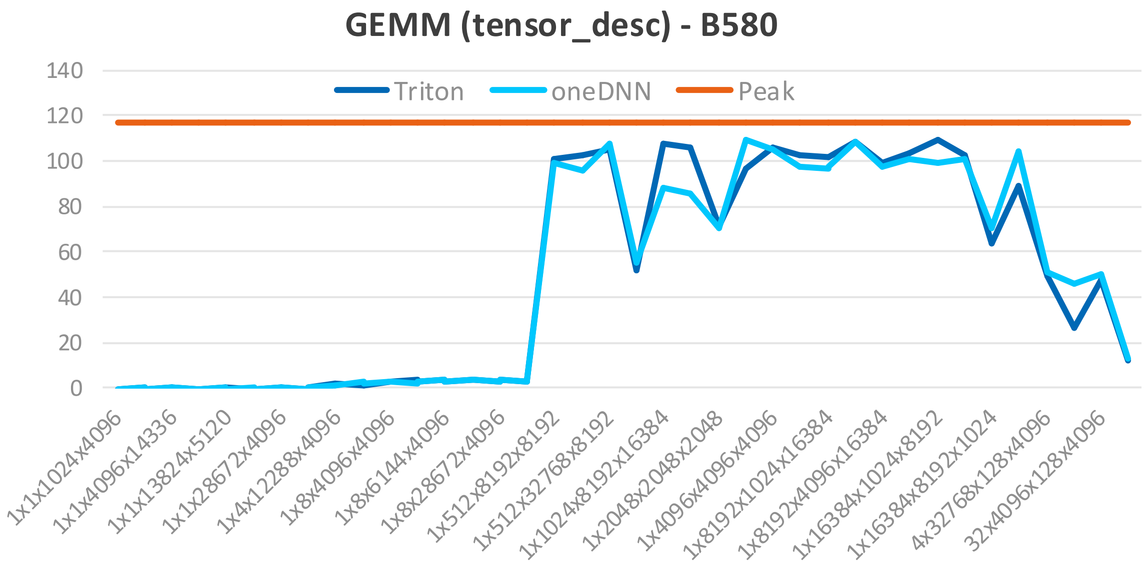
```python
a_desc = tl.make_tensor_descriptor(base=a_ptr, shape=(M, K), strides=(stride_am, stride_ak),
                                   block_shape=(BLOCK_SIZE_M, BLOCK_SIZE_K))
b_desc = tl.make_tensor_descriptor(base=b_ptr, shape=(K, N), strides=(stride_bk, stride_bn),
                                   block_shape=(BLOCK_SIZE_K, BLOCK_SIZE_N))

accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
off_k = 0
for _ in range(0, K, BLOCK_SIZE_K):
    a = a_desc.load([pid_m * BLOCK_SIZE_M, off_k])
    b = b_desc.load([off_k, pid_n * BLOCK_SIZE_N])
    accumulator += tl.dot(a, b)
    off_k += BLOCK_SIZE_K
c = accumulator.to(tl.float32)

c_desc = tl.make_tensor_descriptor(base=c_ptr, shape=(M, N), strides=(stride_cm, stride_cn),
                                   block_shape=(BLOCK_SIZE_M, BLOCK_SIZE_N))
c_desc.store([pid_m * BLOCK_SIZE_M, pid_n * BLOCK_SIZE_N], c)
```

GEMM Performance (Arc-B series GPUs)

Triton Implementation

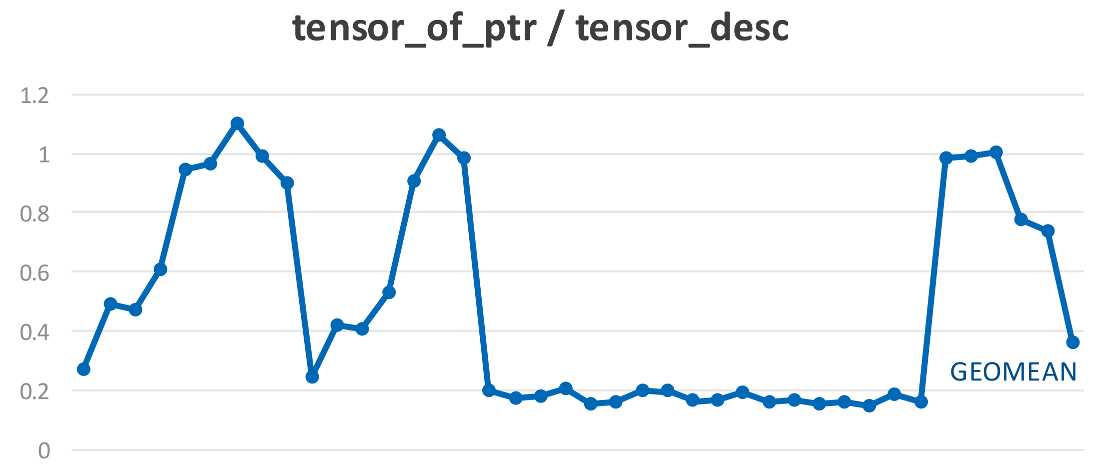
Intel implementation leverages key Xe2 HW features:

- DPAS (Dot Product Accumulate Systolic)
- 2D block reads (read blocks of data from GM to registers)
- Performance: Triton reaches 93% of peak Tflops (on B50 GPUs) for a GEMM kernel using tensor descriptors to point to GEMM's operands (e.g. tl.dot operation)



Challenges

- Intel 2D block reads generated generation require use of Triton's structured pointers (i.e. tensor descriptors)
- Tensor of pointer performance (unstructured) ~40% of implementation using tensor descriptors
- Ongoing efforts to reduce performance gap



Flex Attention performance (Arc-B Series GPUs)

Introduction

- FlexAttention is a new PyTorch API designed to provide flexibility to end users (lets users define custom variants)
- The new API supports several optimized variants of the classic FlashAttention algorithm (e.g. Causal, PagedAttention, etc…)
- FlexAttention was released as a *prototype* in PyTorch 2.5

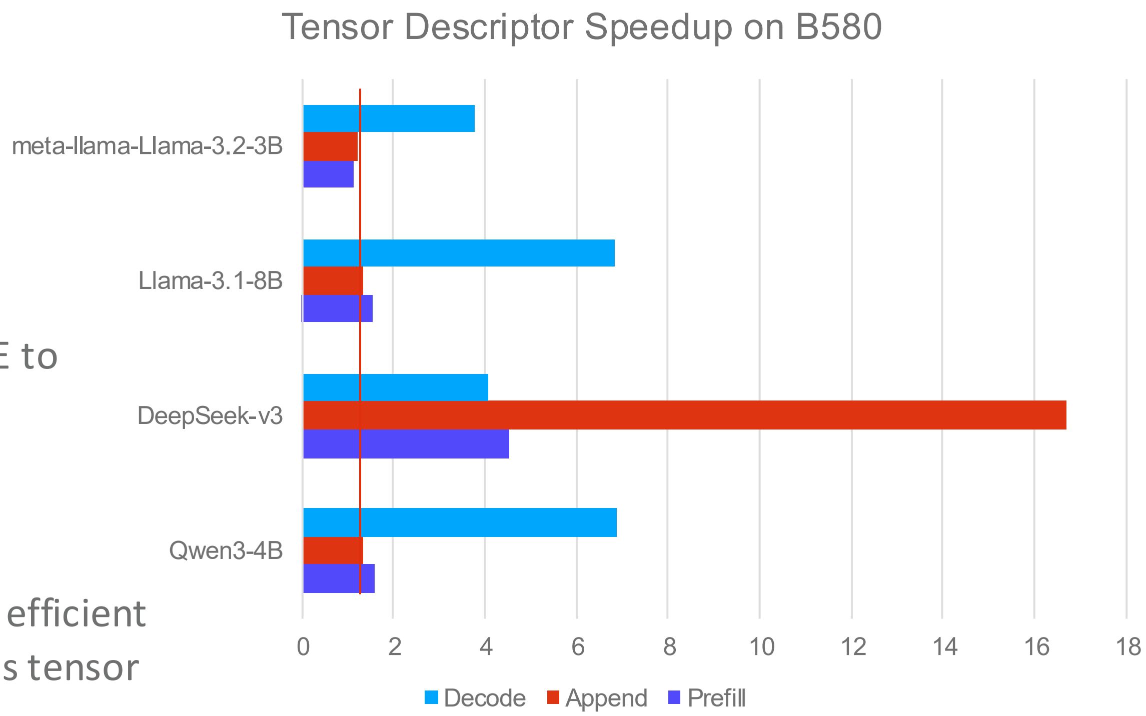
Support

- Enabled FlexAttention [support] for Intel GPUs in PyTorch 2.9
- Workload significantly stressed ability of Intel's low-level GPU BE to schedule kernel instructions without register spills.

Challenges

- FlexAttention forward pass uses tensor descriptors, allowing for efficient implementation on Intel GPUs. However, the backward pass uses tensor of pointers. Working with the PyTorch community to update the backward pass.
- Certain input shapes have suboptimal performance: additional scheduler/vectorizer improvement WIP

PyTorch FlexAttention (BATCH_SIZE=1)

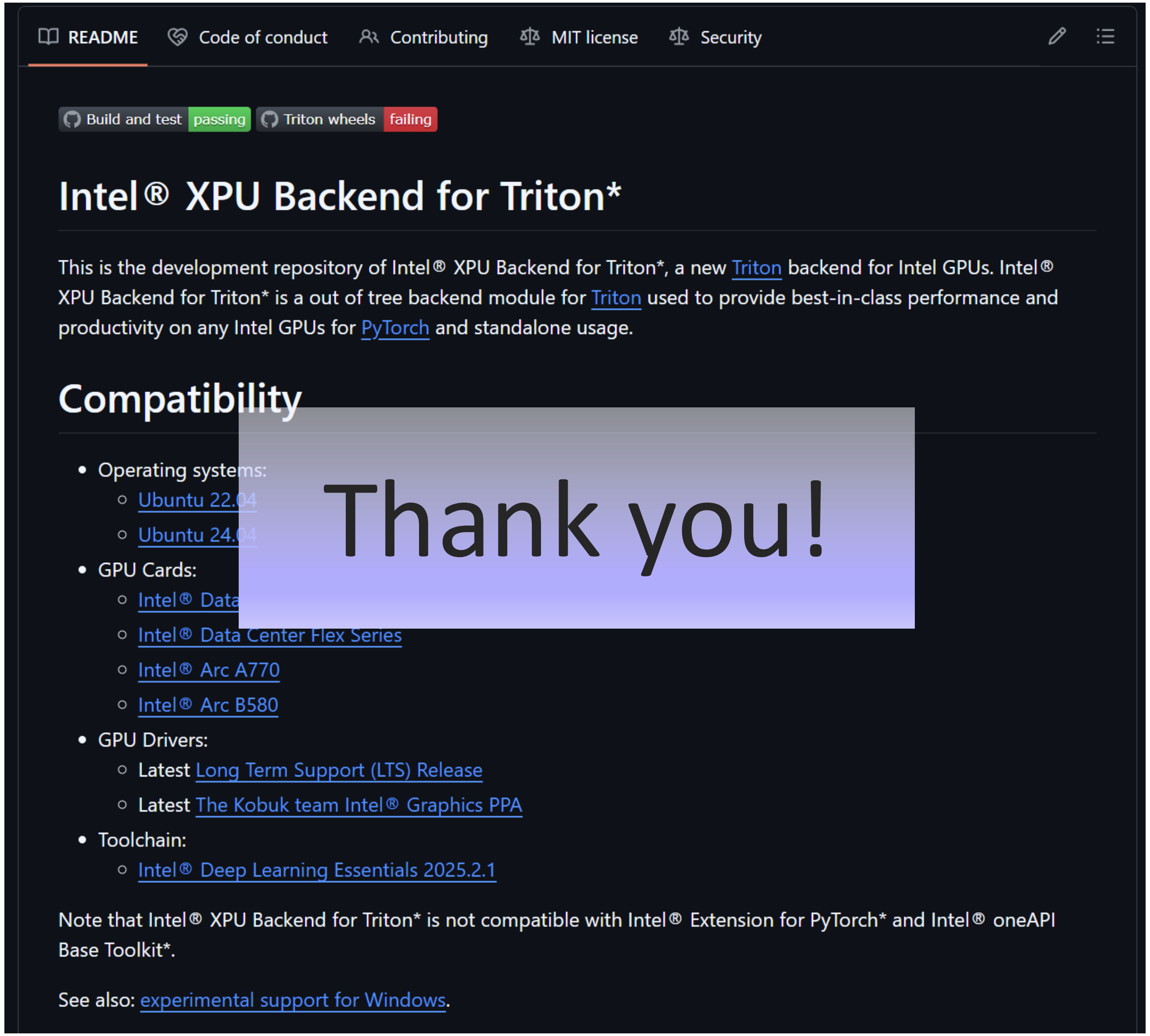


Summary

- Extended OpenAI Triton compiler with a 3rd party backend for Intel GPUs
  - ✓ Supports current generation GPUs
  - ✓ Added Windows OS support
  - ✓ Shipped as part of PyTorch official releases
  - ✓ Support for upcoming inference GPU (Cresent Island) is ongoing
- GEMM-like operations (tl.dot in Triton) have been optimized
  - ✓ Exploitation of efficient HW operation (2D block reads, async prefetch, XMX engine)
  - ✓ New Triton tensor descriptors language feature required for peak performance
  - ✓ PyTorch Helium to the rescue: designed to generate optimal kernel for each target GPU architecture
- New PyTorch Flex Attention feature is supported (PyTorch 2.9)
  - ✓ Optimized for FWD pass
  - ✓ BWD pass: require changes in PyTorch to use tensor descriptors (Meta)

Project on GitHub

[Triton backend for Intel® GPUs]



README | Code of conduct | Contributing | MIT license | Security

Build and test passing | Triton wheels failing

Intel® XPU Backend for Triton*

This is the development repository of Intel® XPU Backend for Triton*, a new Triton backend for Intel GPUs. Intel® XPU Backend for Triton* is a out of tree backend module for Triton used to provide best-in-class performance and productivity on any Intel GPUs for PyTorch and standalone usage.

Compatibility

- Operating systems:
  - Ubuntu 22.04
  - Ubuntu 24.04
- GPU Cards:
  - Intel® Data
  - Intel® Data Center Flex Series
  - Intel® Arc A770
  - Intel® Arc B580
- GPU Drivers:
  - Latest Long Term Support (LTS) Release
  - Latest The Kobuk team Intel® Graphics PPA
- Toolchain:
  - Intel® Deep Learning Essentials 2025.2.1

Note that Intel® XPU Backend for Triton* is not compatible with Intel® Extension for PyTorch* and Intel® oneAPI Base Toolkit*.

See also: experimental support for Windows.

Thank you!

intel®