Enable Triton on Intel® GPUs

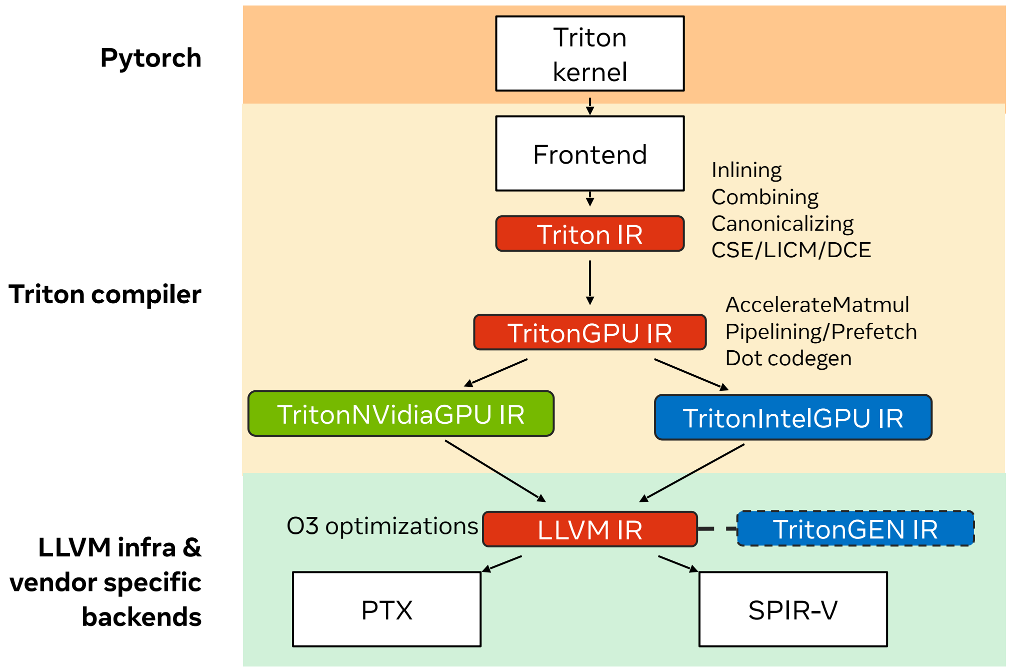
Ettore Tiotto Whitney Tsang Jianhui Li Intel Software

Notices and Disclaimers

Agenda

- Intel Backend Architecture
- Functional Results & Integration with PyTorch
- HW Feature Exploitation
- Performance Results
- Challenges

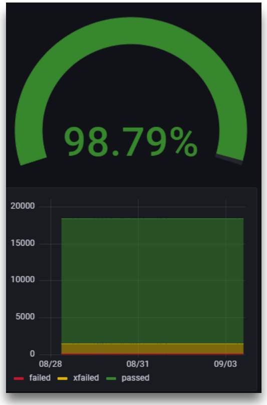
Intel Backend Architecture



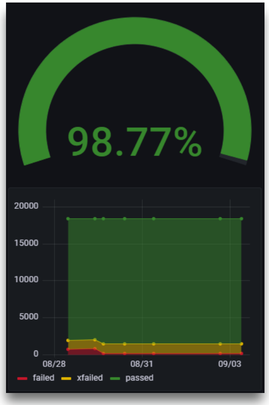
- Intel Triton backend redesigned (switched to lowering to the MLIR's LLVM dialect -> LLVMIR -> SPIRV)
- Reusing Triton IR passes unchanged, customized TritonGPU IR passes to support Intel's specific "Dot" operation layout (DPAS Layout)
- Intel GPUs HW exploitation abstracted by the TritonGEN dialect (e.g. DPAS, 2D block reads/writes, etc…), preempting the need to generate 'inline asm'

Functional Results & Integration with PyTorch

Intel® Data Center GPU Max

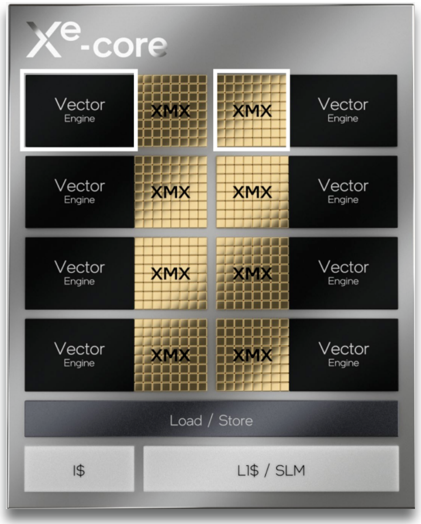


Intel® Lunar Lake GPU



- Adapted OpenAI's regression tests, approaching 99% success rate on GPU Max (aka Ponte Vecchio) and Lunar Lake (using upstream PyTorch)
- PyTorch 2.4 added support for Intel® GPU Acceleration of AI Workloads - Intel's Triton GPU backend integrated in Inductor (torch.compile mode)
- Regularly run PyTorch benchmark suites (Torchbench, Huggingface, TIMM models). On par with upstream PyTorch.

HW Feature Exploitation

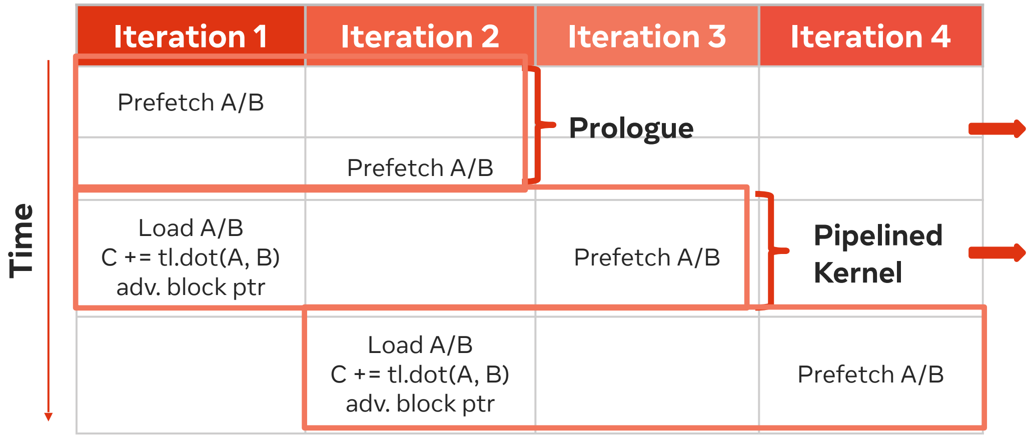


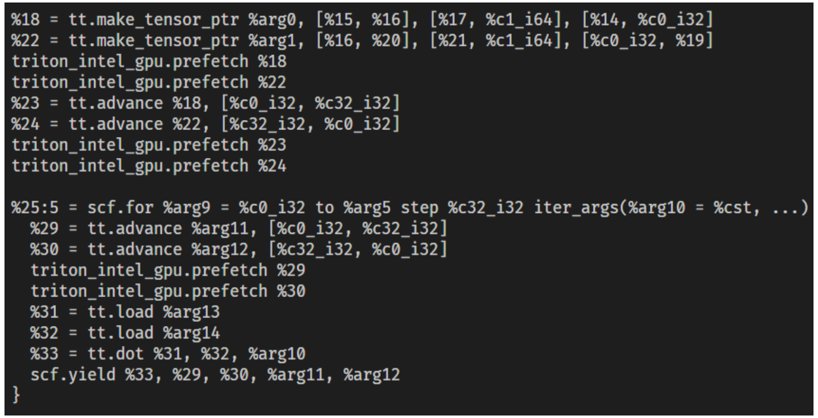
XMX Support

Data Type	Ops/Clock
TF32	2048
FP16 / BF16	4096
INT8	8192

- Intel 2nd generation Xe-core contains 8 XMX Engines supporting high performance matrix multiplications
- XMX Engine programmable via the DPAS (Dot Product Accumulate Systolic) HW instruction.
- Xe-core offers efficient HW instructions to load (and prefetch) 2D blocks from memory.

Reusing Software Pipeline Pass for Prefetch





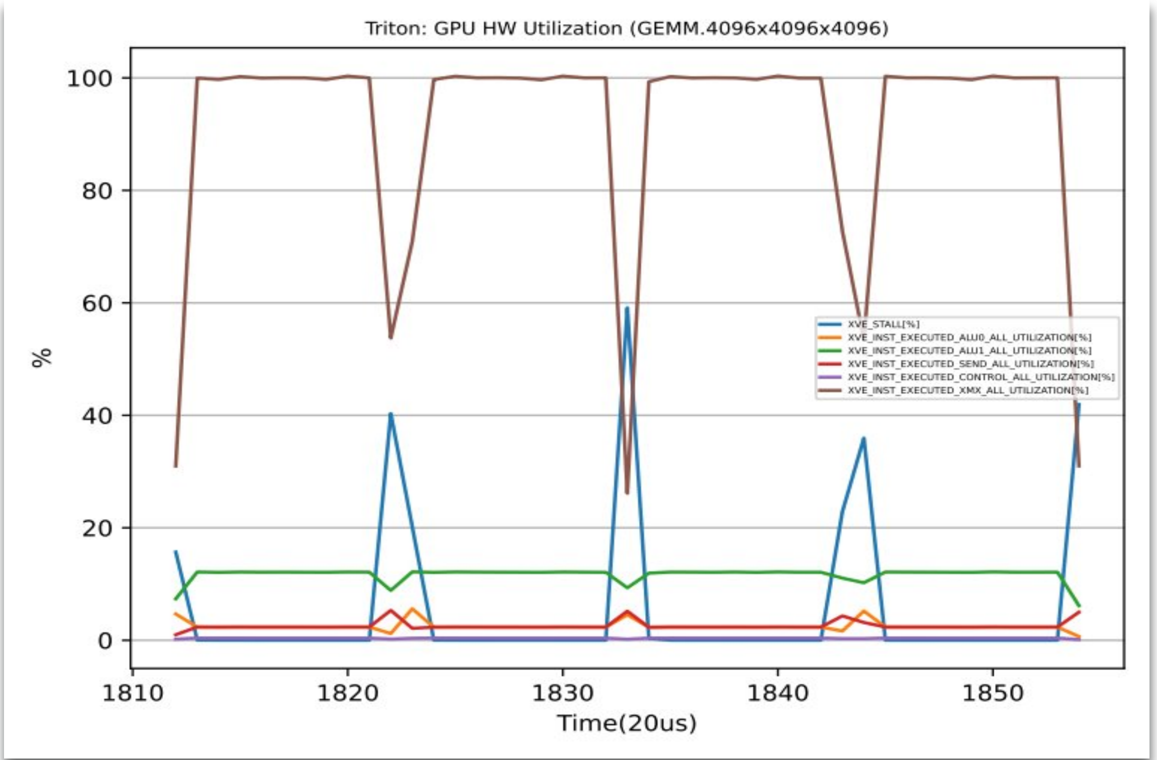
```
%18 = tt.make_tensor_ptr %arg0, [%15, %16], [%17, %c1_i64], [%14, %c0_i32]
%22 = tt.make_tensor_ptr %arg1, [%16, %20], [%21, %c1_i64], [%c0_i32, %19]
triton_intel_gpu.prefetch %18
triton_intel_gpu.prefetch %22
%23 = tt.advance %18, [%c0_i32, %c32_i32]
%24 = tt.advance %22, [%c32_i32, %c0_i32]
triton_intel_gpu.prefetch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

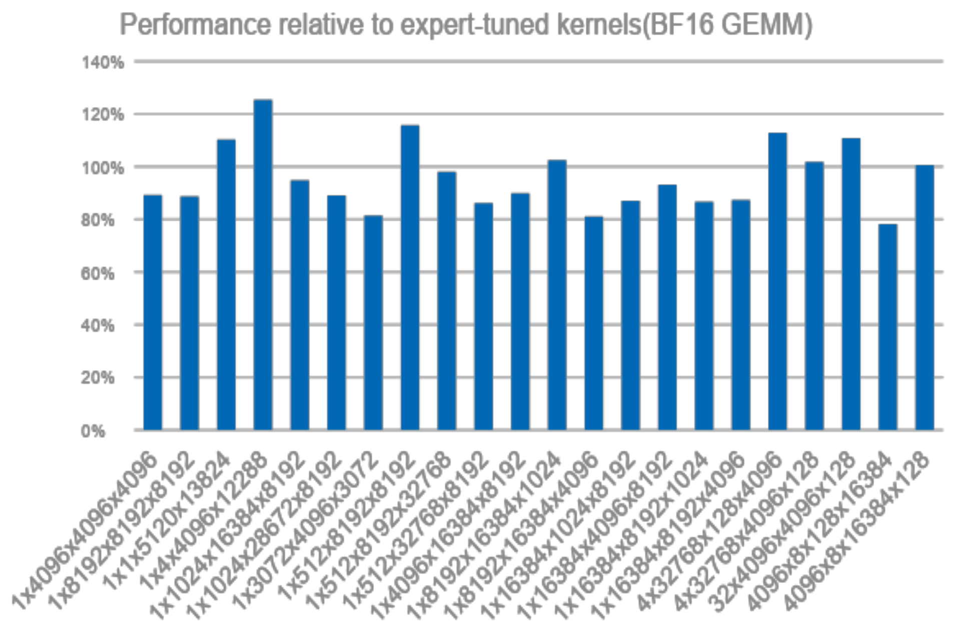
- triton_intel_gpu.prefetch ⍰2D Block Prefetches
- tt.load ⍰ 2D Block Loads
- tt.dot ⍰ DPAS (Dot Product Accumulate Systolic)

HW Utilization (GEMM)



- HW Utilization on Intel Data Center Max GPU 1550
- Achieved high XMX HW peak utilization (brown line)
- Similar utilization graph as tuned library implementation

Performance Results



- Tested on Intel Data Center Max GPU 1550
- Performance on a GEMM kernel is 90+% relative to library implementation

(*) Tested by Intel as of Aug 5, 2024. Result may vary.

Performance Results

- Intel Triton Backend uses the semantic information associated with block pointers to exploit the 2D block reads/prefetch HW instructions
- Block pointers are a Triton experimental feature and may be deprecated upstream
- Can TMA descriptors be used in place of block pointers (to allow codegen of 2D block operations)?

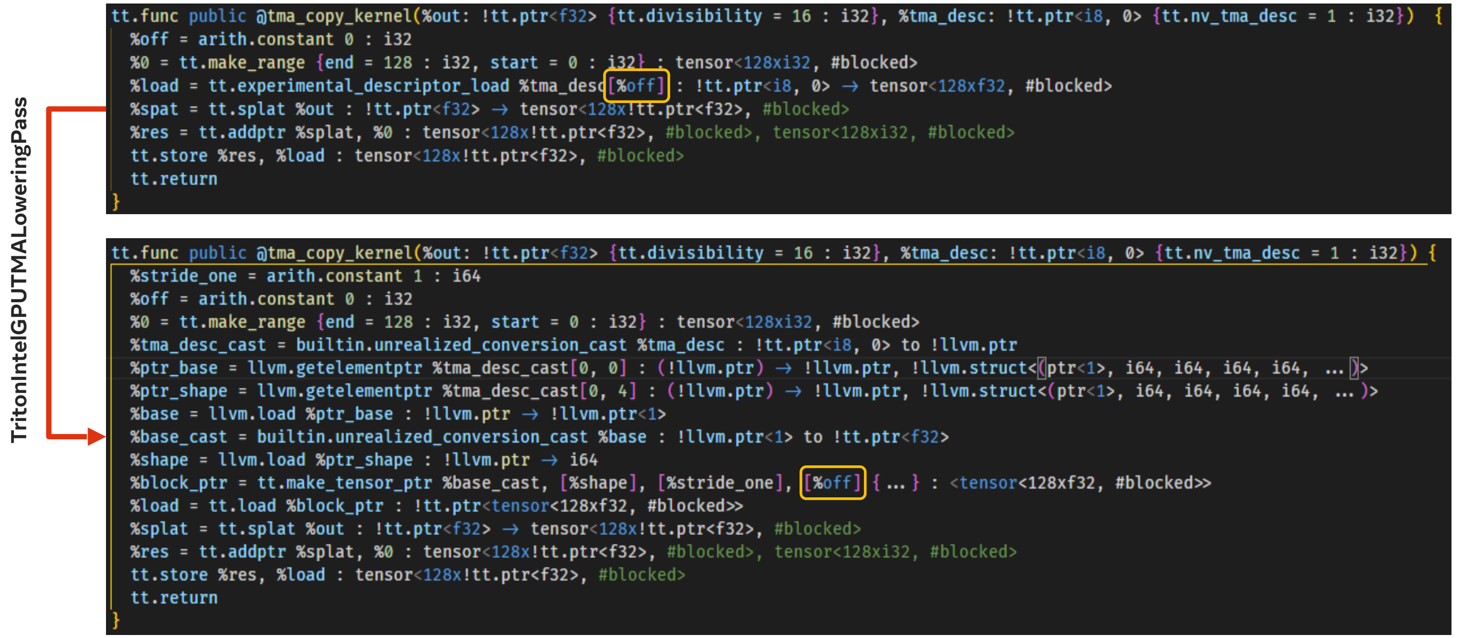
TMA Descriptors vs Block pointers

- Similar layout, but TMA desc. doesn't have "block offsets"
- TMA descriptor created on the host and passed to the kernel
- Usage comparison:

TMA Descriptor	**Block Pointer**
Tensor Base Tensor Shapes Tensor Strides Block Shapes	Tensor Base Tensor Shapes Tensor Strides Block Shapes **Block Offsets**

TMA Desc.	**Logical Operation**	**Block pointer**
create_2d_tma_descriptor (on host)	Create memory descriptor	tt.make_block_pointer (in kernel)
Offsets computed using arithmetic operations	Advance offsets	tt.advance
experimental_descriptor_load	Access memory	tt.load

TMA Desc. vs Block pointer



```
tt.func public @tma_copy_kernel(%out: !tt.ptr<f32> {tt.divisibility = 16 : i32}, %tma_desc: !tt.ptr<i8, 0> {tt.nv_tma_desc = 1 : i32})  {
  %off = arith.constant 0 : i32
  %0 = tt.make_range {end = 128 : i32, start = 0 : i32} : tensor<128xi32, #blocked>
  %load = tt.experimental_descriptor_load %tma_desc[%off] : !tt.ptr<i8, 0> → tensor<128xf32, #blocked>
  %spat = tt.splat %out : !tt.ptr<f32> → tensor<128x!tt.ptr<f32>, #blocked>
  %res = tt.addptr %splat, %0 : tensor<128x!tt.ptr<f32>, #blocked>, tensor<128xi32, #blocked>
  tt.store %res, %load : tensor<128x!tt.ptr<f32>, #blocked>
  tt.return
}
```

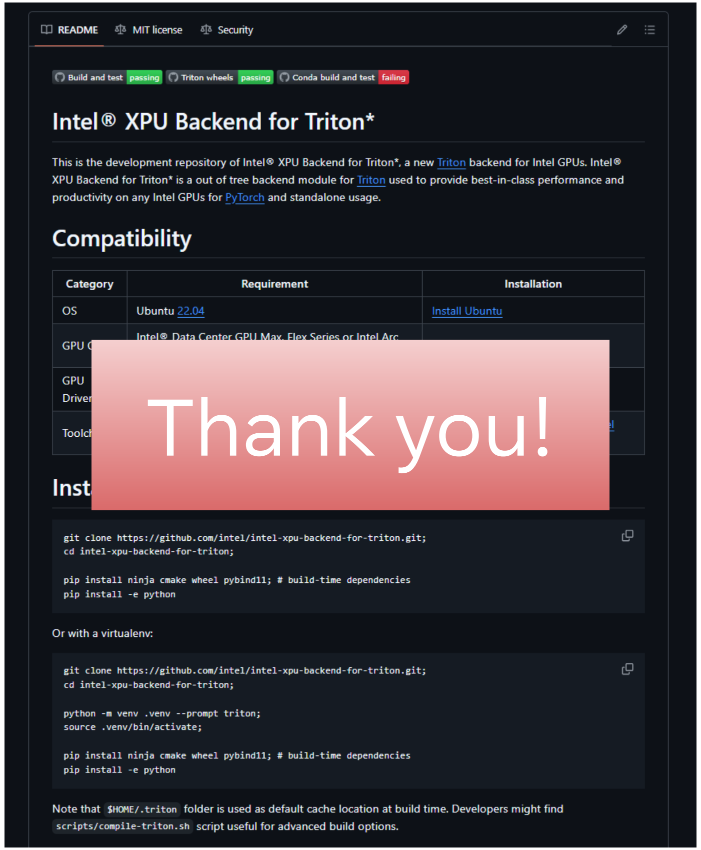
```
tt.func public @tma_copy_kernel(%out: !tt.ptr<f32> {tt.divisibility = 16 : i32}, %tma_desc: !tt.ptr<i8, 0> {tt.nv_tma_desc = 1 : i32}) {
  %stride_one = arith.constant 1 : i64
  %off = arith.constant 0 : i32
  %0 = tt.make_range {end = 128 : i32, start = 0 : i32} : tensor<128xi32, #blocked>
  %tma_desc_cast = builtin.unrealized_conversion_cast %tma_desc : !tt.ptr<i8, 0> to !llvm.ptr
  %ptr_base = llvm.getelementptr %tma_desc_cast[0, 0] : (!llvm.ptr) → !llvm.ptr, !llvm.struct<(ptr<1>, i64, i64, i64, i64, ... )>
  %ptr_shape = llvm.getelementptr %tma_desc_cast[0, 4] : (!llvm.ptr) → !llvm.ptr, !llvm.struct<(ptr<1>, i64, i64, i64, i64, ... )>
  %base = llvm.load %ptr_base : !llvm.ptr → !llvm.ptr<1>
  %base_cast = builtin.unrealized_conversion_cast %base : !llvm.ptr<1> to !tt.ptr<f32>
  %shape = llvm.load %ptr_shape : !llvm.ptr → i64
  %block_ptr = tt.make_tensor_ptr %base_cast, [%shape], [%stride_one], [%off] {...} : <tensor<128xf32, #blocked>>
  %load = tt.load %block_ptr : !tt.ptr<tensor<128xf32, #blocked>>
  %splat = tt.splat %out : !tt.ptr<f32> → tensor<128x!tt.ptr<f32>, #blocked>
  %res = tt.addptr %splat, %0 : tensor<128x!tt.ptr<f32>, #blocked>, tensor<128xi32, #blocked>
  tt.store %res, %load : tensor<128x!tt.ptr<f32>, #blocked>
  tt.return
}
```

WIP

- experimental_descriptor_load “converted” to make_tensor_ptr + tt.load
- base and shape arguments obtained from the TMA descriptor
- block offset obtained from the experimental_descriptor_load operation

Project on GitHub

Triton backend for Intel® GPUs



README MIT license Security

Build and test passing | Triton wheels passing | Conda build and test 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

Category	Requirement	Installation
OS	Ubuntu 22.04	Install Ubuntu
GPU C	Intel® Data Center GPU Max, Flex Series or Intel Arc	
GPU Driver		
Toolch		el

Thank you!

Inst

```
git clone https://github.com/intel/intel-xpu-backend-for-triton.git;
cd intel-xpu-backend-for-triton;

pip install ninja cmake wheel pybind11; # build-time dependencies
pip install -e python
```

Or with a virtualenv:

```
git clone https://github.com/intel/intel-xpu-backend-for-triton.git;
cd intel-xpu-backend-for-triton;

python -m venv .venv --prompt triton;
source .venv/bin/activate;

pip install ninja cmake wheel pybind11; # build-time dependencies
pip install -e python
```

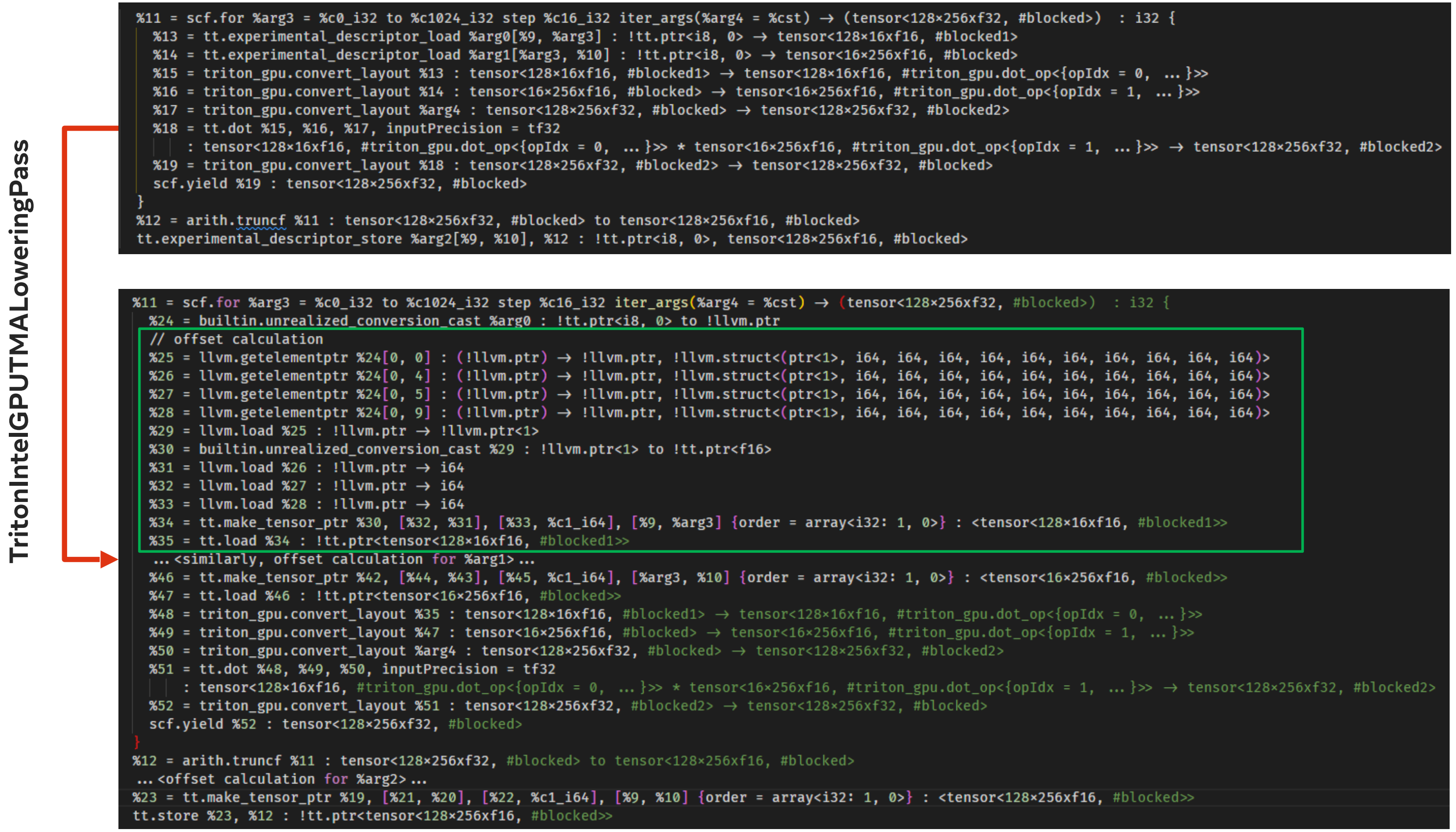
Note that `$HOME/.triton` folder is used as default cache location at build time. Developers might find `scripts/compile-triton.sh` script useful for advanced build options.

Configuration details

- GPU: Intel Data Center Max GPU 1550
- OS: Ubuntu 22.04
- Intel Extension for PyTorch: intel/intel-extension-for-pytorch at dev/triton-test-3.0
- Intel XPU Backend for Triton: intel/intel-xpu-backend-for-triton at a71e7be
- Toolchain: PyTorch Development Bundle v0.5.2
- Graph driver: Rolling Stable Version 914.32
- Command line

```
TRITON_INTEL_ENABLE_FAST_PREFETCH=1 TRITON_INTEL_ENABLE_ADDRESS_PAYLOAD_OPT=1 IGC_VISAOptions=" -TotalGRFNum
256 -enableBCR -nolocalra -printregusage -DPASTokenReduction -enableHalfLSC -abiver 2" IGC_DisableLoopUnroll=1
SYCL_PROGRAM_COMPILE_OPTIONS=" -vc-codegen -vc-disable-indvars-opt -doubleGRF -Xfinalizer ' -printregusage
-enableBCR -DPASTokenReduction ‘ “
python benchmarks/triton_kernels_benchmark/gemm_benchmark.py
```

Creating block pointer from TMA desc.



```
%11 = scf.for %arg3 = %c0_i32 to %c1024_i32 step %c16_i32 iter_args(%arg4 = %cst) → (tensor<128×256xf32, #blocked>)  : i32 {
  %13 = tt.experimental_descriptor_load %arg0[%9, %arg3] : !tt.ptr<i8, 0> → tensor<128×16xf16, #blocked1>
  %14 = tt.experimental_descriptor_load %arg1[%arg3, %10] : !tt.ptr<i8, 0> → tensor<16×256xf16, #blocked>
  %15 = triton_gpu.convert_layout %13 : tensor<128×16xf16, #blocked1> → tensor<128×16xf16, #triton_gpu.dot_op<{opIdx = 0,  …}>>
  %16 = triton_gpu.convert_layout %14 : tensor<16×256xf16, #blocked> → tensor<16×256xf16, #triton_gpu.dot_op<{opIdx = 1,  …}>>
  %17 = triton_gpu.convert_layout %arg4 : tensor<128×256xf32, #blocked> → tensor<128×256xf32, #blocked2>
  %18 = tt.dot %15, %16, %17, inputPrecision = tf32
      : tensor<128×16xf16, #triton_gpu.dot_op<{opIdx = 0,  …}>> * tensor<16×256xf16, #triton_gpu.dot_op<{opIdx = 1,  …}>> → tensor<128×256xf32, #blocked2>
  %19 = triton_gpu.convert_layout %18 : tensor<128×256xf32, #blocked2> → tensor<128×256xf32, #blocked>
  scf.yield %19 : tensor<128×256xf32, #blocked>
}
%12 = arith.truncf %11 : tensor<128×256xf32, #blocked> to tensor<128×256xf16, #blocked>
tt.experimental_descriptor_store %arg2[%9, %10], %12 : !tt.ptr<i8, 0>, tensor<128×256xf16, #blocked>
```

```
%11 = scf.for %arg3 = %c0_i32 to %c1024_i32 step %c16_i32 iter_args(%arg4 = %cst) → (tensor<128×256xf32, #blocked>)  : i32 {
  %24 = builtin.unrealized_conversion_cast %arg0 : !tt.ptr<i8, 0> to !llvm.ptr
  // offset calculation
  %25 = llvm.getelementptr %24[0, 0] : (!llvm.ptr) → !llvm.ptr, !llvm.struct<(ptr<1>, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64)>
  %26 = llvm.getelementptr %24[0, 4] : (!llvm.ptr) → !llvm.ptr, !llvm.struct<(ptr<1>, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64)>
  %27 = llvm.getelementptr %24[0, 5] : (!llvm.ptr) → !llvm.ptr, !llvm.struct<(ptr<1>, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64)>
  %28 = llvm.getelementptr %24[0, 9] : (!llvm.ptr) → !llvm.ptr, !llvm.struct<(ptr<1>, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64)>
  %29 = llvm.load %25 : !llvm.ptr → !llvm.ptr<1>
  %30 = builtin.unrealized_conversion_cast %29 : !llvm.ptr<1> to !tt.ptr<f16>
  %31 = llvm.load %26 : !llvm.ptr → i64
  %32 = llvm.load %27 : !llvm.ptr → i64
  %33 = llvm.load %28 : !llvm.ptr → i64
  %34 = tt.make_tensor_ptr %30, [%32, %31], [%33, %c1_i64], [%9, %arg3] {order = array<i32: 1, 0>} : <tensor<128×16xf16, #blocked1>>
  %35 = tt.load %34 : !tt.ptr<tensor<128×16xf16, #blocked1>>
  …<similarly, offset calculation for %arg1>…
  %46 = tt.make_tensor_ptr %42, [%44, %43], [%45, %c1_i64], [%arg3, %10] {order = array<i32: 1, 0>} : <tensor<16×256xf16, #blocked>>
  %47 = tt.load %46 : !tt.ptr<tensor<16×256xf16, #blocked>>
  %48 = triton_gpu.convert_layout %35 : tensor<128×16xf16, #blocked1> → tensor<128×16xf16, #triton_gpu.dot_op<{opIdx = 0,  …}>>
  %49 = triton_gpu.convert_layout %47 : tensor<16×256xf16, #blocked> → tensor<16×256xf16, #triton_gpu.dot_op<{opIdx = 1,  …}>>
  %50 = triton_gpu.convert_layout %arg4 : tensor<128×256xf32, #blocked> → tensor<128×256xf32, #blocked2>
  %51 = tt.dot %48, %49, %50, inputPrecision = tf32
      : tensor<128×16xf16, #triton_gpu.dot_op<{opIdx = 0,  …}>> * tensor<16×256xf16, #triton_gpu.dot_op<{opIdx = 1,  …}>> → tensor<128×256xf32, #blocked2>
  %52 = triton_gpu.convert_layout %51 : tensor<128×256xf32, #blocked2> → tensor<128×256xf32, #blocked>
  scf.yield %52 : tensor<128×256xf32, #blocked>
}
%12 = arith.truncf %11 : tensor<128×256xf32, #blocked> to tensor<128×256xf16, #blocked>
 …<offset calculation for %arg2>…
%23 = tt.make_tensor_ptr %19, [%21, %20], [%22, %c1_i64], [%9, %10] {order = array<i32: 1, 0>} : <tensor<128×256xf16, #blocked>>
tt.store %23, %12 : !tt.ptr<tensor<128×256xf16, #blocked>>
```

Thank you!