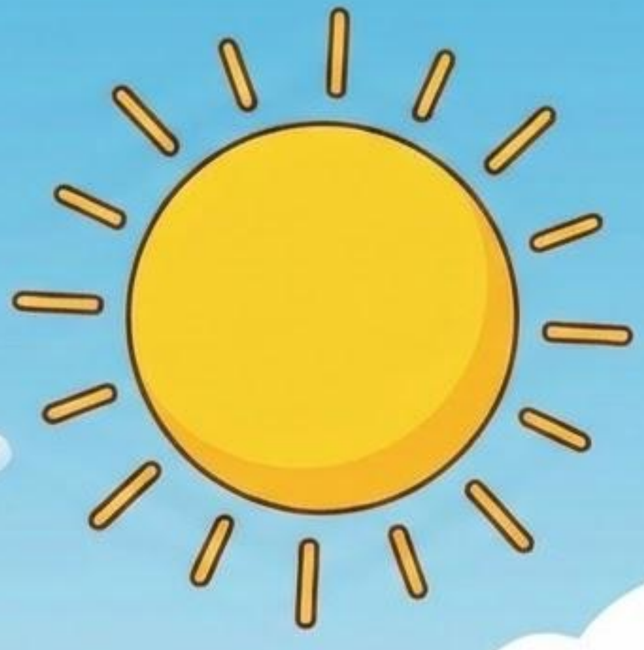
CONDUCTING THE GPU SYMPHONY 🎶 WITH RUST 🦀 AND TRITON ⚡

KUBA SOŁTYS

SOLUTIONS ARCHITECT



HTTPS://WWW.YOUTUBE.COM/@QOOBA

HTTPS://GITHUB.COM/QOOBA

HTTPS://WWW.LINKEDIN.COM/IN/QOOBA

BIELIK LM CONTRIBUTOR

WRITING GPU KERNELS & RUST CODE

HTTPS://WWW.YOUTUBE.COM/@QOOBA

HTTPS://GITHUB.COM/QOOBA

HTTPS://WWW.LINKEDIN.COM/IN/QOOBA

THE OVERTURE

FROM AGENTS TO KERNELS

TRITON VS CUDA

C++

INSIDE BIELIK 1.5B

MatMul, Softmax, RMSNorm

espressivo, C++

THE GRAND FINALE

JIT VS AOT TRITON

JIT

AOT

Ahead of Time

Compilate

DEEP DIVE: TRITON TO PTX

con fuoco

RUST BINDINGS

mf sicuro

BENCHMARK:

RUST VS PYTHON

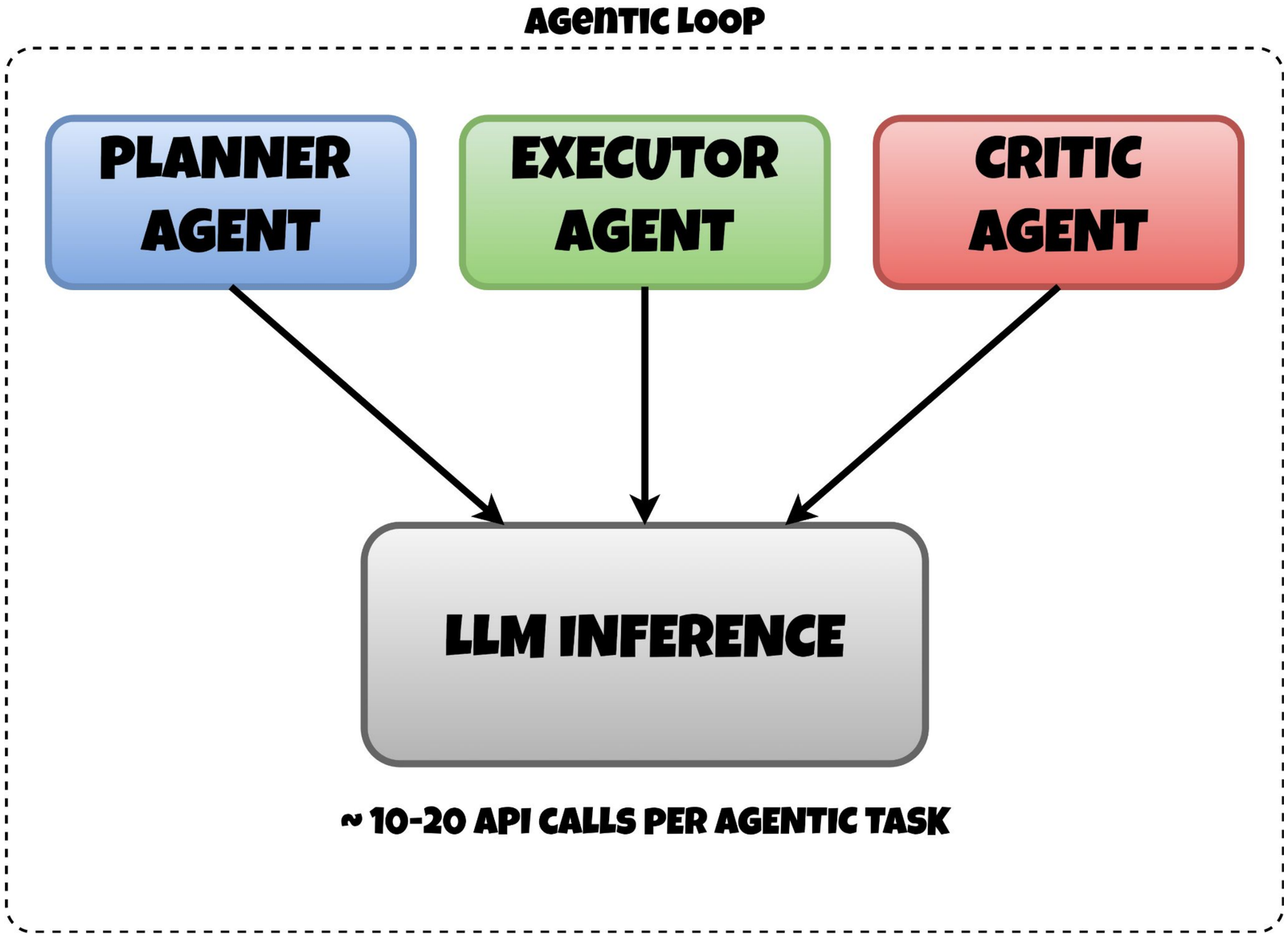
RUST

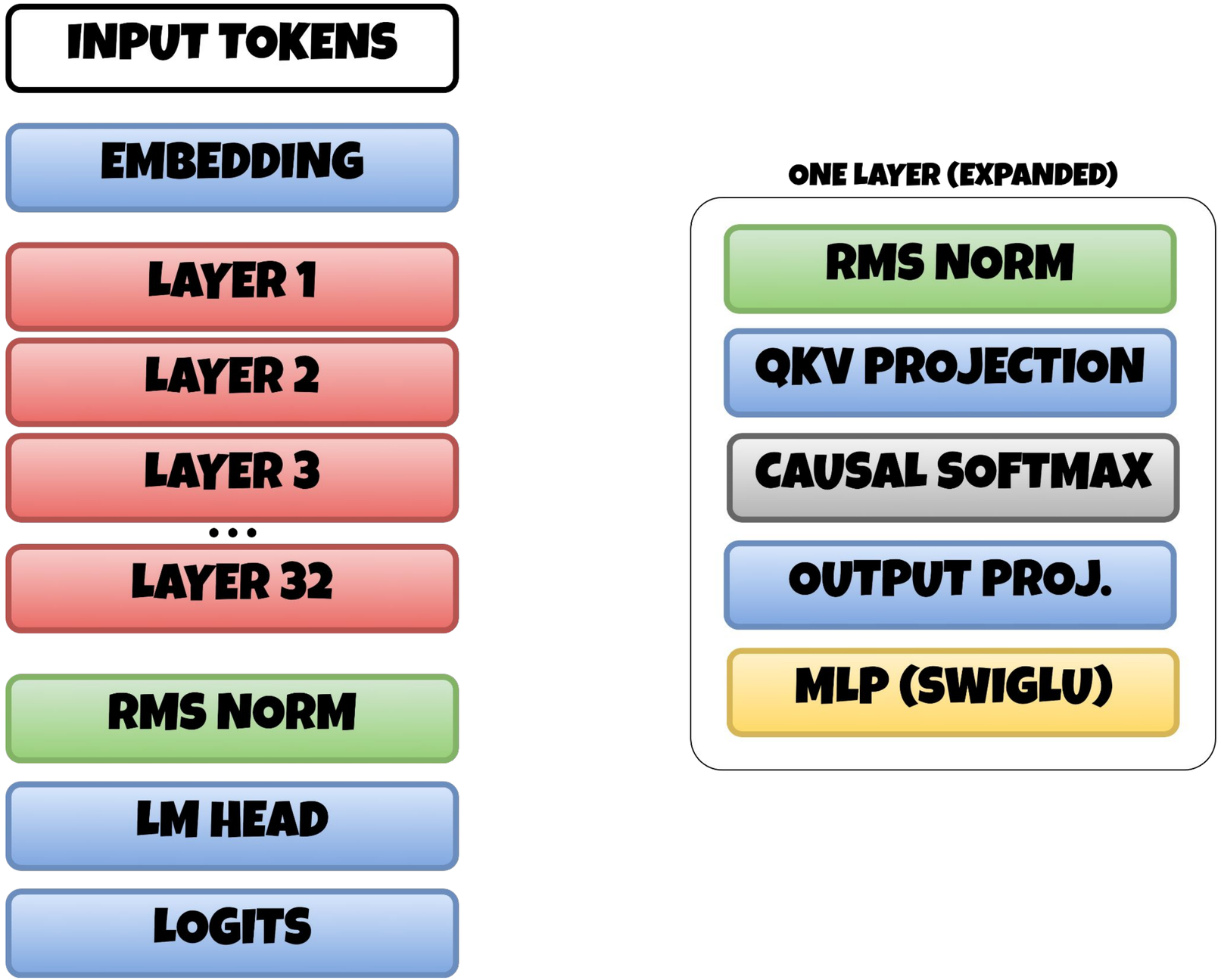
PYTHON

Speed (x)

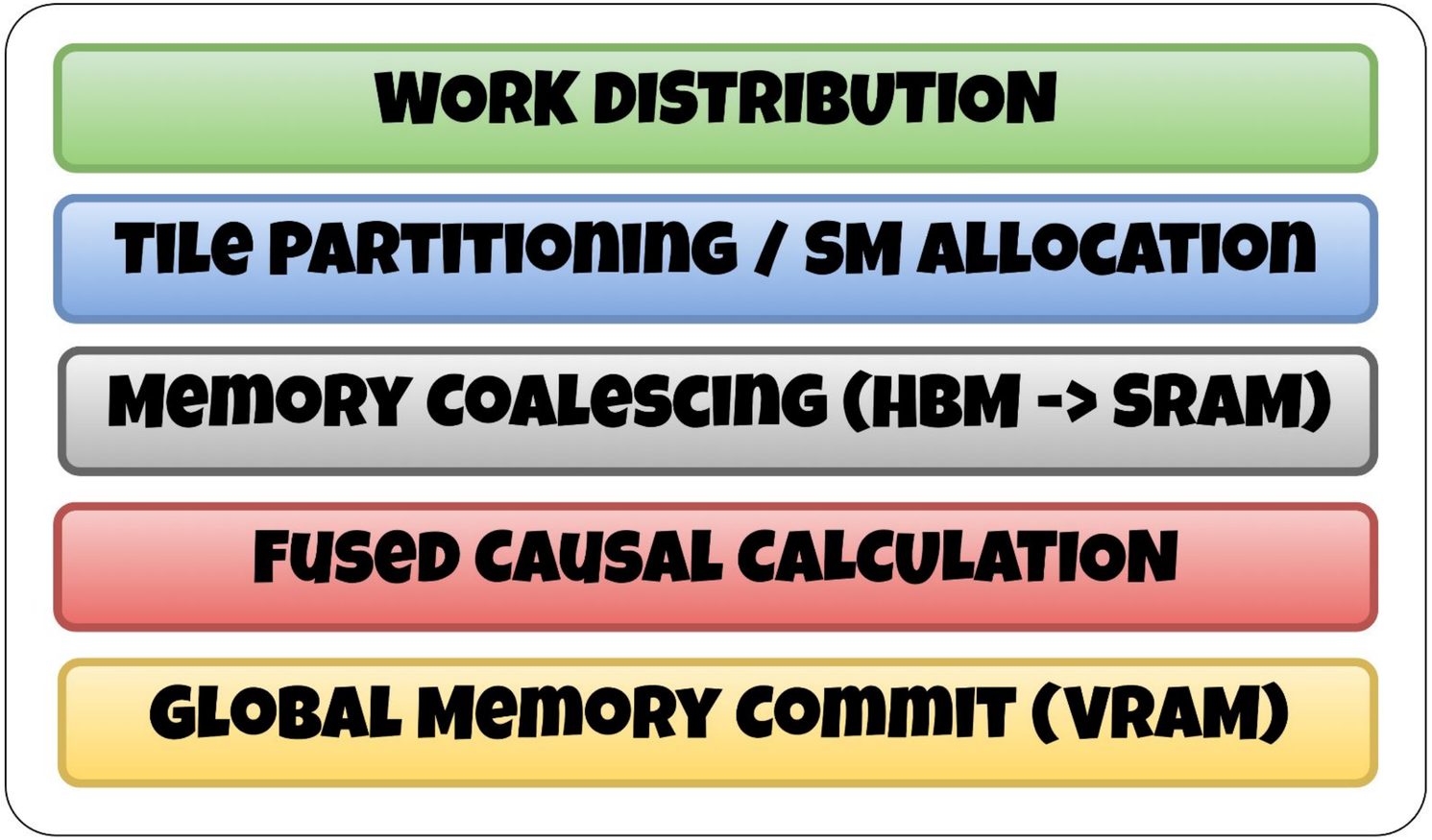
Rust Python

WHY GPU KERNELS ?


Agentic Loop
PLANNER AGENT
EXECUTOR AGENT
CRITIC AGENT
LLM INFERENCE
~ 10-20 API CALLS PER AGENTIC TASK


INPUT TOKENS
EMBEDDING
LAYER 1
LAYER 2
LAYER 3
...
LAYER 32
RMS NORM
LM HEAD
LOGITS
ONE LAYER (EXPANDED)
RMS NORM
QKV PROJECTION
CAUSAL SOFTMAX
OUTPUT PROJ.
MLP (SWIGLU)

GPU EXECUTION



FASTER KERNEL -> FASTER TOKEN -> FASTER CALL -> FASTER TASK

CUDA VS TRITON: ABSTRACTION LEVEL

RAW CUDA C++

```cpp
// You manage every thread

__global__ void softmax_kernel(
    float* x, float* out, int n
) {
  int tid = blockIdx.x * blockDim.x
          + threadIdx.x;

  // manual warp sync
  __syncwarp();

  // manual shared mem
  __shared__ float shmem[1024];

  // … 200 more lines

}
```

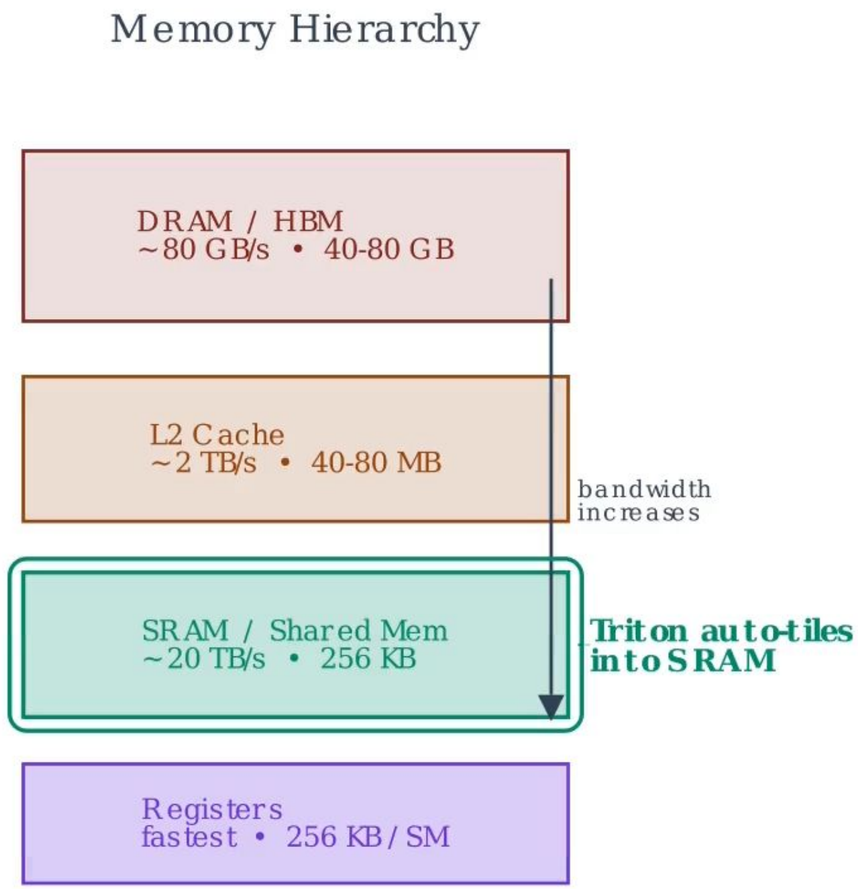
THREAD-LEVEL CONTROL. WARP DIVERGENCE, BANK CONFLICTS, COALESCING - ALL YOURS.

TRTION

```python
@triton.jit
def softmax_kernel(
    x_ptr, out_ptr, n_rows, n_cols,
    BLOCK_SIZE: tl.constexpr
):
    row  = tl.program_id(0)   # block idx
    offs = tl.arange(0, BLOCK_SIZE)
    x    = tl.load(x_ptr + row * n_cols
                   + offs,
                   mask=offs < n_cols,
                   other=-float('inf'))
    x   -= tl.max(x, axis=0)
    e    = tl.exp(x)
    out  = e / tl.sum(e, axis=0)
    tl.store(out_ptr + row * n_cols
             + offs, out,
             mask=offs < n_cols)
```

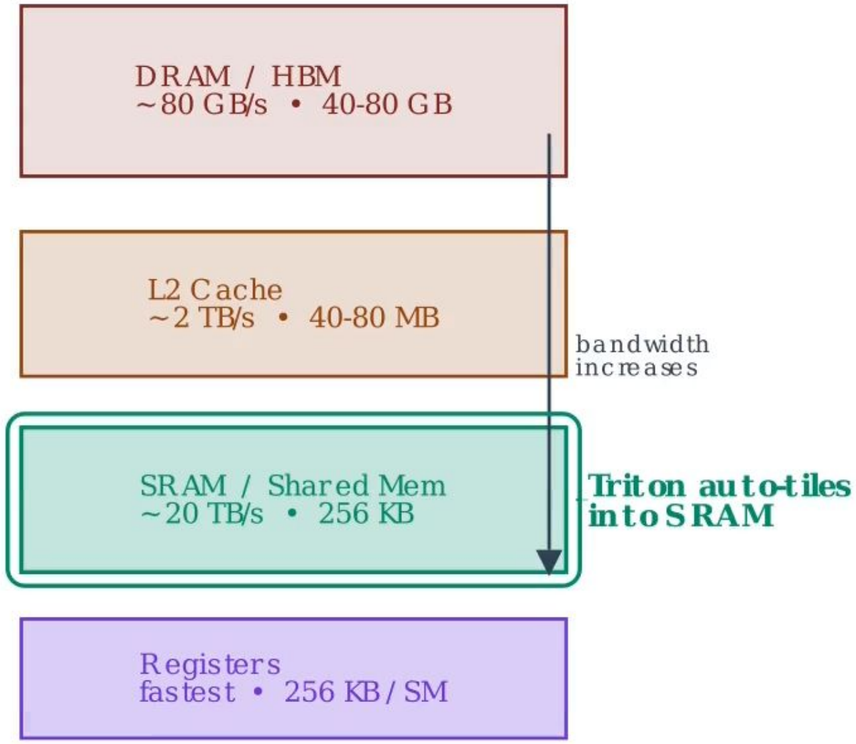
BLOCK-LEVEL. TRITON HANDLES SRAM, COALESCING & SYNC.

GPU Memory Hierarchy & Triton's Abstraction



GPU Memory Hierarchy & Triton's Abstraction

Memory Hierarchy

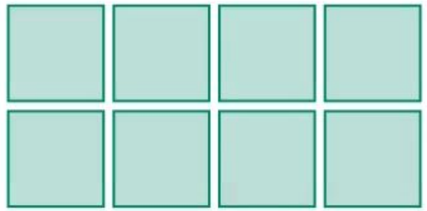


Triton vs CUDA Programming

CUDA: thread-level

Each thread → 1 element
You manage tiling, sync, banks

Triton: block-level



Each program → 1 tile (BLOCK_SIZE)
Auto SRAM, coalescing, sync

Triton kernel — block-level thinking

```
pid  = tl.program_id(0)          # block index
offs = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
x    = tl.load(ptr + offs)       # load tile → SRAM
y    = tl.exp(x - tl.max(x))  # compute in regs
```

Triton hides warp/thread management & SRAM scheduling

WHAT TRITON GIVES US

- ✓ **Automatic SRAM management** - tiles loaded to shared memory without explicit `__shared__` arrays
- ✓ **Global memory coalescing** - Triton ensures consecutive threads access consecutive addresses
- ✓ **Latency hiding** - instruction-level pipelining between memory ops and arithmetic
- ✓ **Tensor cores for free** - `tl.dot(a, b, allow_tf32=True)` emits `mma.sync` instructions automatically; no manual warp-level intrinsics
- ✓ **Auto-tuning** - `@triton.autotune` tries multiple `BLOCK_SIZE` values, picks best

Triton's insight: optimal GPU kernels differ only in *tile size*, not structure. Write the structure once; let the compiler tune the rest.

MATMUL, SOFTMAX, RMSNORM

MATMUL

```python
import torch
import triton
import triton.language as tl

@triton.jit
def matmul_tiled_kernel(
    a_ptr, b_ptr, c_ptr,
    M, N, K,
    BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr,
):
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)

    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)

    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)

    for k_start in range(0, K, BLOCK_K):
        offs_k = k_start + tl.arange(0, BLOCK_K)

        a_tile = tl.load(a_ptr + offs_m[:, None] * K + offs_k[None, :],
            mask=(offs_m[:, None] < M) & (offs_k[None, :] < K),
            other=0.0,
        )

        b_tile = tl.load(
            b_ptr + offs_k[:, None] * N + offs_n[None, :],
            mask=(offs_k[:, None] < K) & (offs_n[None, :] < N),
            other=0.0,
        )

        acc += tl.dot(a_tile, b_tile, allow_tf32=True)

    out_mask = (offs_m[:, None] < M) & (offs_n[None, :] < N)
    tl.store(
        c_ptr + offs_m[:, None] * N + offs_n[None, :],
        acc,
        mask=out_mask,
    )
```

CUDA Core vs Tensor Core

CUDA Core

$$c = a \times b + c$$

Example:

$$3.5 \times 2.1 + 0 = 7.35$$

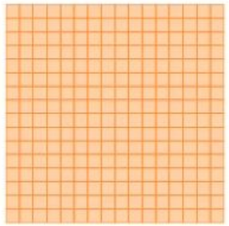
Performance:

~19 TFLOPS (FP32)

16× FASTER!

Tensor Core

$$D_{16\times16} = A_{16\times16} @ B_{16\times16} + C_{16\times16}$$



Performance:

~312 TFLOPS (FP16)

```python
import triton
import triton.language as tl

@triton.jit
def softmax_causal_simple_kernel(
    input_ptr, output_ptr, n_rows, n_cols, seq_len,
    input_row_stride, output_row_stride,
    BLOCK_SIZE: tl.constexpr,
):
    row_idx = tl.program_id(axis=0)

    if row_idx >= n_rows:
        return

    position = row_idx % seq_len

    input_row_start = input_ptr + row_idx * input_row_stride
    output_row_start = output_ptr + row_idx * output_row_stride

    col_offsets = tl.arange(0, BLOCK_SIZE)
    mask = col_offsets < n_cols

    row_vals = tl.load(input_row_start + col_offsets, mask=mask, other=-float('inf'))
    is_future = col_offsets > position
    row_vals = tl.where(is_future, -float('inf'), row_vals)

    row_max = tl.max(row_vals, axis=0)
    numerator = tl.exp(row_vals - row_max)
    denominator = tl.sum(numerator, axis=0)
    softmax_output = numerator / denominator

    tl.store(output_row_start + col_offsets, softmax_output, mask=mask)
```

CAUSAL MASK

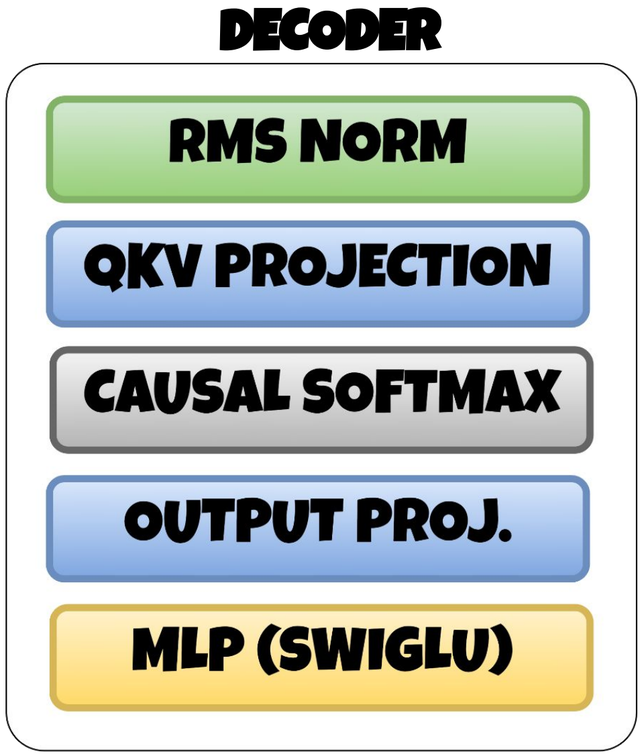
DECODER

- RMS NORM
- QKV PROJECTION
- CAUSAL SOFTMAX
- OUTPUT PROJ.
- MLP (SWIGLU)

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i - \max(\mathbf{z})}}{\sum_{j=1}^{N} e^{z_j - \max(\mathbf{z})}}$$

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

```python
import triton
import triton.language as tl


@triton.jit
def rms_norm_simple_kernel(
    X, W, Y,
    stride_x_row, stride_y_row,
    n_cols, eps,
    BLOCK_SIZE: tl.constexpr,
):
    row_idx = tl.program_id(axis=0)

    X += row_idx * stride_x_row
    Y += row_idx * stride_y_row

    col_offsets = tl.arange(0, BLOCK_SIZE)
    mask = col_offsets < n_cols

    x = tl.load(X + col_offsets, mask=mask, other=0.0)
    w = tl.load(W + col_offsets, mask=mask, other=1.0)

    x_squared = x * x
    mean_x_squared = tl.sum(x_squared) / n_cols
    rms = tl.sqrt(mean_x_squared + eps)
    x_normalized = x / rms
    y = x_normalized * w

    tl.store(Y + col_offsets, y, mask=mask)
```



$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\text{RMS}(\mathbf{x}) + \epsilon} \odot \mathbf{w}$$

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}$$

JIT → AOT: THE MINDSET SHIFT

JIT (PyTorch standard)

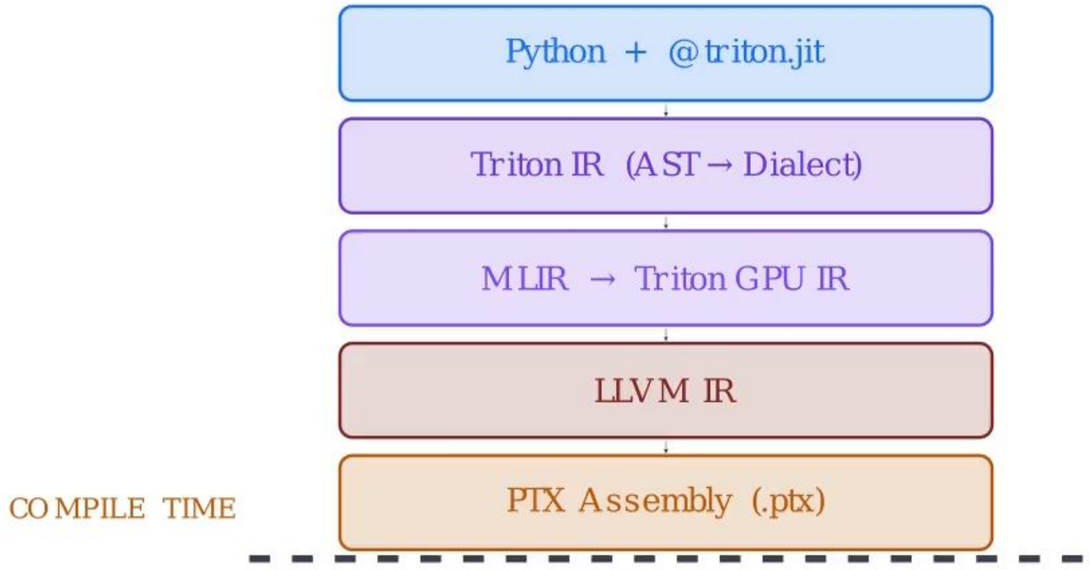
- ✓ Convenient - kernel compiled on first call
- ✓ BLOCK_SIZE tuned per input shape
- ⚠ Requires Python + Triton at runtime
- ⚠ First-call latency ("cold start")
- ⚠ 500+ MB Python dependency tree

AOT (our approach)

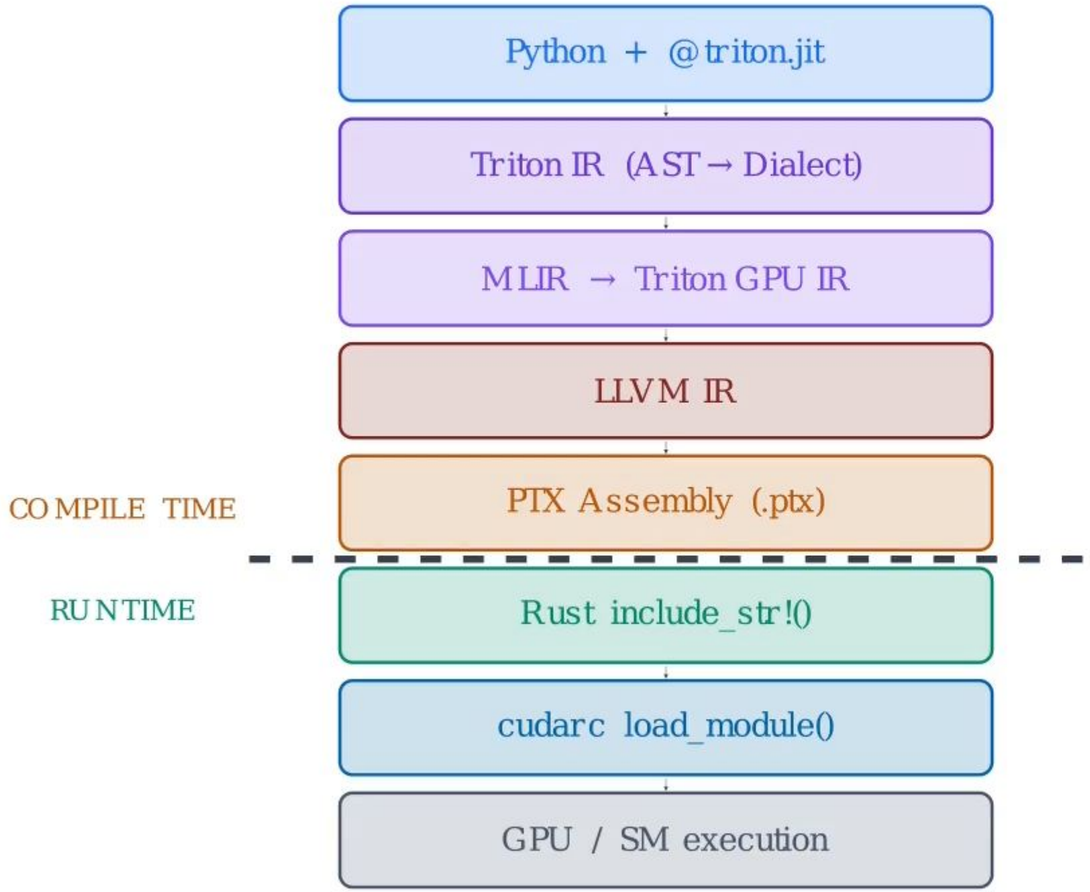
- ✓ PTX compiled *once*, embedded in binary
- ✓ Zero Python at runtime
- ✓ Instant load - no compilation step
- ⚠ BLOCK_SIZE *frozen* at compile time
- ⚠ Must target specific GPU (SM arch)

TRITON COMPILATION DEEP DIVE

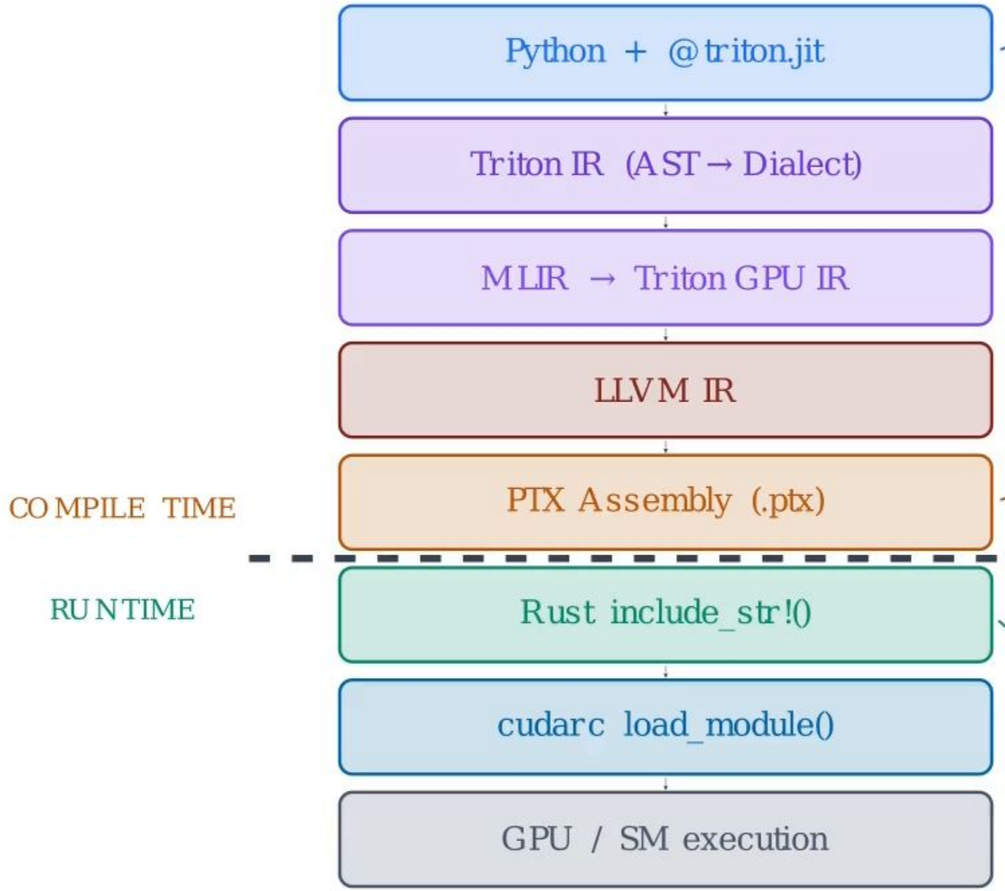
AoT Compilation: Python → GPU Assembly



AoT Compilation: Python → GPU Assembly



AoT Compilation: Python → GPU Assembly



Python / Triton source

```
@triton.jit
def softmax_kernel(
    input_ptr, output_ptr,
    n_rows, n_cols,
    BLOCK_SIZE: tl.constexpr
):
```

PTX output (no Python needed!)

```
.visible .entry softmax_kernel(
  .param .u64 input_ptr,
  .param .u64 output_ptr,
  .param .u32 n_rows,
  .param .u64 meta1, // Triton meta
  .param .u64 meta2  // Triton meta
)
```

Rust: embed at compile time

```
const PTX: &str =
    include_str!("softmax.ptx");

// Zero Python overhead at runtime
```

```python
import triton
import triton.language as tl
import triton.compiler as tc

@triton.jit
def add_kernel(x_ptr, y_ptr, ouptut_ptr, n_elements, BLOCK_SIZE:
tl.constexpr):
    pid = tl.program_id(0)
    block_start = pid * BLOCK_SIZE

    offset = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offset < n_elements

    x = tl.load(x_ptr + offset, mask=mask, other=0.0)
    y = tl.load(y_ptr + offset, mask=mask, other=0.0)

    output = x + y

    tl.store(ouptut_ptr + offset, output, mask=mask)

```


```python
target = triton.runtime.driver.active.get_current_target()

src = tc.ASTSource(
    fn=add_kernel,
    signature={"x_ptr": "*fp32", "y_ptr": "*fp32",
               "ouptut_ptr": "*fp32", "n_elements": "i32"},
    constexprs={"BLOCK_SIZE": 256}
)

compiled_kernel = triton.compile(src, target=target)

print("# TRITON IR")
print("---")
print(compiled_kernel.asm["ttir"])
print("# TRITON-GPU IR")
print("---")
print(compiled_kernel.asm["ttgir"])
print("# LLVM IR")
print("---")
print(compiled_kernel.asm["llir"])
print("# PTX")
print("---")
print(compiled_kernel.asm["ptx"])
print("# SASS")
print("---")
print(compiled_kernel.asm["sass"])
```

Step 1: Triton IR

GPU-agnostic tile operations. Parses Python AST, resolves tl.constexpr (BLOCK_SIZE baked in). No threads, no warps yet.

```
#loc = loc("/home/qooba/add_kernel_compile.py":6:0)
#loc15 = loc("x_ptr"(#loc))
#loc16 = loc("y_ptr"(#loc))
#loc17 = loc("ouptut_ptr"(#loc))
#loc18 = loc("n_elements"(#loc))
module {
  tt.func public @add_kernel(%x_ptr: !tt.ptr<f32> loc("x_ptr"(#loc)), %y_ptr: !tt.ptr<f32>
 loc("y_ptr"(#loc)), %ouptut_ptr: !tt.ptr<f32> loc("ouptut_ptr"(#loc)), %n_elements: i32
 loc("n_elements"(#loc))) attributes {noinline = false} {
    %cst = arith.constant dense<0.000000e+00> : tensor<256xf32> loc(#loc1)
    %c256_i32 = arith.constant 256 : i32 loc(#loc1)
    %pid = tt.get_program_id x : i32 loc(#loc19)
    %block_start = arith.muli %pid, %c256_i32 : i32 loc(#loc20)
    %offset = tt.make_range {end = 256 : i32, start = 0 : i32} : tensor<256xi32> loc(#loc21)
    %offset_0 = tt.splat %block_start : i32 -> tensor<256xi32> loc(#loc22)
    %offset_1 = arith.addi %offset_0, %offset : tensor<256xi32> loc(#loc22)
    %mask = tt.splat %n_elements : i32 -> tensor<256xi32> loc(#loc23)
    %mask_2 = arith.cmpi slt, %offset_1, %mask : tensor<256xi32> loc(#loc23)
    %x = tt.splat %x_ptr : !tt.ptr<f32> -> tensor<256x!tt.ptr<f32>> loc(#loc24)
    %x_3 = tt.addptr %x, %offset_1 : tensor<256x!tt.ptr<f32>>, tensor<256xi32> loc(#loc24)
    %x_4 = tt.load %x_3, %mask_2, %cst : tensor<256x!tt.ptr<f32>> loc(#loc25)
    %y = tt.splat %y_ptr : !tt.ptr<f32> -> tensor<256x!tt.ptr<f32>> loc(#loc26)
    %y_5 = tt.addptr %y, %offset_1 : tensor<256x!tt.ptr<f32>>, tensor<256xi32> loc(#loc26)
    %y_6 = tt.load %y_5, %mask_2, %cst : tensor<256x!tt.ptr<f32>> loc(#loc27)
    %output = arith.addf %x_4, %y_6 : tensor<256xf32> loc(#loc28)
    %0 = tt.splat %ouptut_ptr : !tt.ptr<f32> -> tensor<256x!tt.ptr<f32>> loc(#loc12)
    %1 = tt.addptr %0, %offset_1 : tensor<256x!tt.ptr<f32>>, tensor<256xi32> loc(#loc12)
    tt.store %1, %output, %mask_2 : tensor<256x!tt.ptr<f32>> loc(#loc13)
    tt.return loc(#loc14)
  } loc(#loc)
```

Step 2: TritonGPU IR

Tiles mapped to hardware. Decides warps, threads, shared memory. Each tensor gets a layout annotation.

```
#blocked = #ttg.blocked<{sizePerThread = [1], threadsPerWarp = [32], warpsPerCTA = [4], order = [0]}>
#loc = loc("/home/qooba/add_kernel_compile.py":6:0)
#loc15 = loc("x_ptr"(#loc))
#loc16 = loc("y_ptr"(#loc))
#loc17 = loc("ouptut_ptr"(#loc))
#loc18 = loc("n_elements"(#loc))
module attributes {"ttg.num-ctas" = 1 : i32, "ttg.num-warps" = 4 : i32, ttg.target = "cuda:89", "ttg.threads-per-warp" = 32 : i32} {
  tt.func public @add_kernel(%x_ptr: !tt.ptr<f32> loc("x_ptr"(#loc)), %y_ptr: !tt.ptr<f32> loc("y_ptr"(#loc)), %ouptut_ptr: !tt.ptr<f32> loc("ouptut_ptr"(#loc)), %n_elements: i32 loc("n_elements"(#loc))) attributes {noinline = false} {
    %c256_i32 = arith.constant 256 : i32 loc(#loc1)
    %cst = arith.constant dense<0.000000e+00> : tensor<256xf32, #blocked> loc(#loc1)
    %pid = tt.get_program_id x : i32 loc(#loc19)
    %block_start = arith.muli %pid, %c256_i32 : i32 loc(#loc20)
    %offset = tt.make_range {end = 256 : i32, start = 0 : i32} : tensor<256xi32, #blocked> loc(#loc21)
    %offset_0 = tt.splat %block_start : i32 -> tensor<256xi32, #blocked> loc(#loc22)
    %offset_1 = arith.addi %offset_0, %offset : tensor<256xi32, #blocked> loc(#loc22)
    %mask = tt.splat %n_elements : i32 -> tensor<256xi32, #blocked> loc(#loc23)
    %mask_2 = arith.cmpi slt, %offset_1, %mask : tensor<256xi32, #blocked> loc(#loc23)
    %x = tt.splat %x_ptr : !tt.ptr<f32> -> tensor<256x!tt.ptr<f32>, #blocked> loc(#loc24)
    %x_3 = tt.addptr %x, %offset_1 : tensor<256x!tt.ptr<f32>, #blocked>, tensor<256xi32, #blocked> loc(#loc24)
    %x_4 = tt.load %x_3, %mask_2, %cst : tensor<256x!tt.ptr<f32>, #blocked> loc(#loc25)
    %y = tt.splat %y_ptr : !tt.ptr<f32> -> tensor<256x!tt.ptr<f32>, #blocked> loc(#loc26)
    %y_5 = tt.addptr %y, %offset_1 : tensor<256x!tt.ptr<f32>, #blocked>, tensor<256xi32, #blocked> loc(#loc26)
    %y_6 = tt.load %y_5, %mask_2, %cst : tensor<256x!tt.ptr<f32>, #blocked> loc(#loc27)
    %output = arith.addf %x_4, %y_6 : tensor<256xf32, #blocked> loc(#loc28)
    %0 = tt.splat %ouptut_ptr : !tt.ptr<f32> -> tensor<256x!tt.ptr<f32>, #blocked> loc(#loc12)
    %1 = tt.addptr %0, %offset_1 : tensor<256x!tt.ptr<f32>, #blocked>, tensor<256xi32, #blocked> loc(#loc12
    tt.store %1, %output, %mask_2 : tensor<256x!tt.ptr<f32>, #blocked> loc(#loc13)
    tt.return loc(#loc14)
  } loc(#loc)
```

Step 3: LLVM IR

Per-thread code. Tile abstraction gone. Register allocation, loop unrolling, inline PTX for masked loads.

```
define ptx_kernel void @add_kernel(ptr addrspace(1) %0, ptr addrspace(1) %1, ptr addrspace(1) %2, i32 %3, ptr addrspace(1) readnone captures(none) %4, ptr addrspace(1) readnone captures(none) %5) local_unnamed_addr #0 !dbg !4 {
  %7 = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x(), !dbg !7
  %8 = shl i32 %7, 8, !dbg !8
  %9 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x(), !dbg !9
  %10 = and i32 %9, 127, !dbg !9
  %11 = or disjoint i32 %8, %10, !dbg !10
  %12 = or disjoint i32 %11, 128, !dbg !10
  %13 = icmp slt i32 %11, %3, !dbg !11
  %14 = icmp slt i32 %12, %3, !dbg !11
  %15 = sext i32 %11 to i64, !dbg !12
  %16 = getelementptr float, ptr addrspace(1) %0, i64 %15, !dbg !12
  %17 = sext i32 %12 to i64, !dbg !12
  %18 = getelementptr float, ptr addrspace(1) %0, i64 %17, !dbg !12
  %19 = tail call i32 asm sideeffect "mov.u32 $0, $1;\0A\09@$3 ld.global.b32 { $0 }, [ $2 + 0 ];", "=r,r,l,b"(i32 0, ptr addrspace(1) %16, i1 %13) #2, !dbg !13
  %20 = bitcast i32 %19 to float, !dbg !13
  %21 = tail call i32 asm sideeffect "mov.u32 $0, $1;\0A\09@$3 ld.global.b32 { $0 }, [ $2 + 0 ];", "=r,r,l,b"(i32 0, ptr addrspace(1) %18, i1 %14) #2, !dbg !13
  %22 = bitcast i32 %21 to float, !dbg !13
  %23 = getelementptr float, ptr addrspace(1) %1, i64 %15, !dbg !14
  %24 = getelementptr float, ptr addrspace(1) %1, i64 %17, !dbg !14
  %25 = tail call i32 asm sideeffect "mov.u32 $0, $1;\0A\09@$3 ld.global.b32 { $0 }, [ $2 + 0 ];", "=r,r,l,b"(i32 0, ptr addrspace(1) %23, i1 %13) #2, !dbg !15
  %26 = bitcast i32 %25 to float, !dbg !15
  %27 = tail call i32 asm sideeffect "mov.u32 $0, $1;\0A\09@$3 ld.global.b32 { $0 }, [ $2 + 0 ];", "=r,r,l,b"(i32 0, ptr addrspace(1) %24, i1 %14) #2, !dbg !15
  %28 = bitcast i32 %27 to float, !dbg !15
  %29 = fadd float %20, %26, !dbg !16
  %30 = fadd float %22, %28, !dbg !16
  %31 = getelementptr float, ptr addrspace(1) %2, i64 %15, !dbg !17
  %32 = getelementptr float, ptr addrspace(1) %2, i64 %17, !dbg !17
  %33 = bitcast float %29 to i32, !dbg !18
  tail call void asm sideeffect "@$2 st.global.b32 [ $1 + 0 ], { $0 };", "r,l,b"(i32 %33, ptr addrspace(1) %31, i1 %13) #2, !dbg !18
  %34 = bitcast float %30 to i32, !dbg !18
  tail call void asm sideeffect "@$2 st.global.b32 [ $1 + 0 ], { $0 };", "r,l,b"(i32 %34, ptr addrspace(1) %32, i1 %14) #2, !dbg !18
  ret void, !dbg !19
}

; Function Attrs: mustprogress nocallback nofree nosync nounwind speculatable willreturn memory(none)
declare noundef range(i32 0, 2147483647) i32 @llvm.nvvm.read.ptx.sreg.ctaid.x() #1

; Function Attrs: mustprogress nocallback nofree nosync nounwind speculatable willreturn memory(none)
declare noundef range(i32 0, 1024) i32 @llvm.nvvm.read.ptx.sreg.tid.x() #1

attributes #0 = { nounwind "nvvm.reqntid"="128" }
attributes #1 = { mustprogress nocallback nofree nosync nounwind speculatable willreturn memory(none) }
attributes #2 = { nounwind }
```

Step 4: PTX — GPU

Portable assembly. This is what we save and embed in Rust.

Note: 6 params, not 4 — Triton adds 2 metadata pointers!

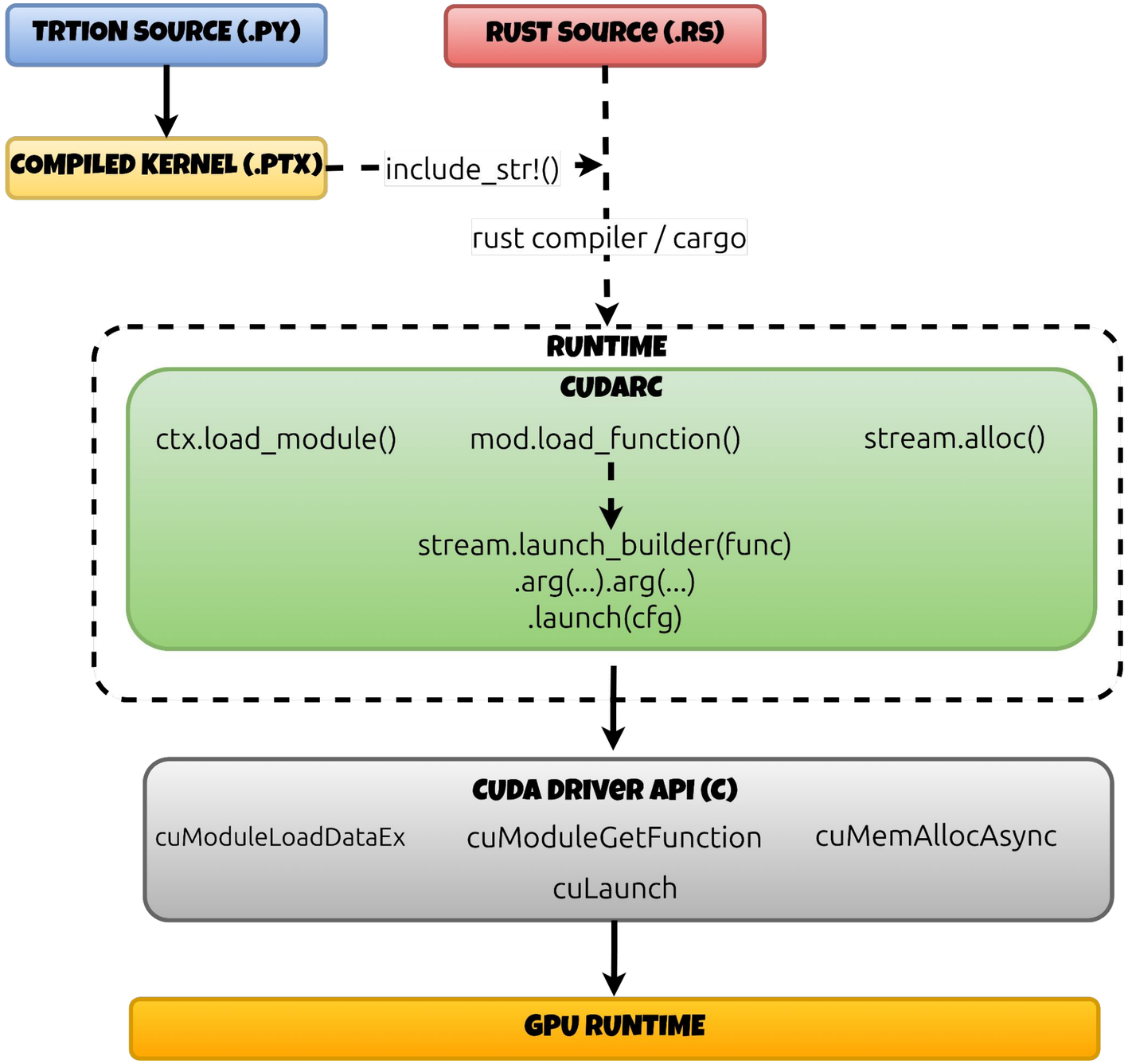
```
.version 8.7
.target sm_89
.address_size 64

    // .globl   add_kernel              // -- Begin function add_kernel
                                        // @add_kernel
.visible .entry add_kernel(
    .param .u64 .ptr .global .align 1 add_kernel_param_0,
    .param .u64 .ptr .global .align 1 add_kernel_param_1,
    .param .u64 .ptr .global .align 1 add_kernel_param_2,
    .param .u32 add_kernel_param_3,
    .param .u64 .ptr .global .align 1 add_kernel_param_4,
    .param .u64 .ptr .global .align 1 add_kernel_param_5
)
.reqntid 128
{
    .reg .pred  %p<7>;
    .reg .b32   %r<18>;
    .reg .b64   %rd<11>;
    .loc    1 6 0                       // add_kernel_compile.py:6:0
$L__func_begin0:
    .loc    1 6 0                       // add_kernel_compile.py:6:0

// %bb.0:
    ld.param.b64    %rd7, [add_kernel_param_0];
    ld.param.b64    %rd8, [add_kernel_param_1];
$L__tmp0:
    .loc    1 7 23                      // add_kernel_compile.py:7:23

 ...
```

Step 5: SASS — Machine Code

GPU-specific binary. CUDA driver compiles PTX -> SASS at runtime on the target GPU. We never ship SASS, only PTX.

```
---
Function:add_kernel
--:-:-:-:2  MOV R1, c[0x0][0x28];
--:-:0:-:1  S2R R0, SR_TID.X;
--:-:-:-:1  MOV R11, 0x4;
--:-:-:-:1  CS2R R6, SRZ;
--:-:-:-:1  ULDC.64 UR4, c[0x0][0x118];
--:-:1:-:1  S2R R3, SR_CTAID.X;
--:-:-:-:1  CS2R R8, SRZ;
01:-:-:Y:4  LOP3.LUT R0, R0, 0x7f, RZ, 0xc0, !PT;
02:-:-:Y:4  LEA R0, R3, R0, 0x8;
--:-:-:-:1  ISETP.GE.AND P0, PT, R0.reuse, c[0x0][0x178], PT;
--:-:-:-:1  IMAD.WIDE R4, R0.reuse, R11, c[0x0][0x168];
--:-:-:Y:4  IADD3 R2, R0, 0x80, RZ;
--:-:-:-:1  ISETP.GE.AND P1, PT, R2, c[0x0][0x178], PT;
--:-:-:Y:6  IMAD.WIDE R2, R0, R11, c[0x0][0x160];
--:-:2:-:4  @!P0 LDG.E R6, [R2.64];
--:-:2:-:4  @!P0 LDG.E R7, [R4.64];
--:-:3:-:4  @!P1 LDG.E R8, [R2.64+0x200];
--:-:3:-:1  @!P1 LDG.E R9, [R4.64+0x200];
04:-:-:-:1  FADD R13, R7, R6;
--:-:-:Y:5  IMAD.WIDE R6, R0, R11, c[0x0][0x170];
--:0:-:-:1  @!P0 STG.E [R6.64], R13;
08:-:-:-:1  FADD R9, R9, R8;
--:-:-:-:6  @P1 EXIT;
--:-:-:-:1  STG.E [R6.64+0x200], R9;
--:-:-:-:5  EXIT;
...
```

TRITON + RUST


TRTION SOURCE (.PY)
RUST SOURCE (.RS)
COMPILED KERNEL (.PTX)
include_str!()
rust compiler / cargo
RUNTIME
CUDARC
ctx.load_module()
mod.load_function()
stream.alloc()
stream.launch_builder(func)
.arg(...).arg(...)
.launch(cfg)
CUDA DRIVER API (C)
cuModuleLoadDataEx
cuModuleGetFunction
cuMemAllocAsync
cuLaunch
GPU RUNTIME

CUDARC: THE GPU BRIDGE

CUDA Runtime API (libcudart)

- ✗ Requires nvcc compiler
- ✗ C++ ABI complexity in Rust FFI
- ✗ Runtime & driver must match

CUDA Driver API (libcuda) via cudarc

- ✓ Load any `.ptx` at runtime
- ✓ No nvcc needed at deploy time
- ✓ Pure Rust, no unsafe in caller code
- ✓ Lazy module caching built-in

```rust
// CudaDevice wraps the driver handle
let device = CudaDevice::new(0)?;   // GPU 0

// Alloc a typed buffer on the device
let buf: CudaSlice<f32> = unsafe { device.alloc(n)? };

// Load a PTX module and get a function handle (cached)
let func = device.get_or_load_custom_func(
    "kernel_name", "module_name", PTX_SOURCE
)?;
```

Option 1: Pure cudarc

Raw GPU buffers — minimal dependencies, full control:

```rust
const PTX: &str = include_str!("../kernels/matmul/compiled/matmul_tiled_kernel.ptx");

pub struct PureMatmulKernel {
    stream: Arc<CudaStream>,
    func:   CudaFunction,
}

impl PureMatmulKernel {
    pub fn new(stream: Arc<CudaStream>) -> Result<Self> {
        let func = stream.context().get_or_load_custom_func(
            "matmul_tiled_kernel", "matmul_module", PTX)?;
        Ok(Self { stream, func })
    }

    pub fn forward(&self, a: &CudaSlice<f32>, b: &CudaSlice<f32>,
                   m: usize, k: usize, n: usize) -> Result<CudaSlice<f32>> {
        let mut c = unsafe { self.stream.alloc::<f32>(m * n)? };
        let cfg = LaunchConfig {
            grid_dim:         (m.div_ceil(32) as u32, n.div_ceil(32) as u32, 1),
            block_dim:        (128, 1, 1),   // matches .reqntid 128 in PTX
            shared_mem_bytes: 16384,          // 2 × (BLOCK_M×BLOCK_K + BLOCK_K×BLOCK_N) × 4
        };
        unsafe {
            self.stream.launch_builder(&self.func)
                .arg(a).arg(b).arg(&mut c)
                .arg(&(m as u32)).arg(&(n as u32)).arg(&(k as u32))
                .arg(&0u64).arg(&0u64)  // ← Triton metadata (must not be omitted!)
                .launch(cfg)?;
        }
        Ok(c)
    }
}
```

OPTION 2: CANDLE CUSTOMOP

Type-safe tensors — shape tracking, layout offsets, device binding handled automatically:

```rust
pub struct MatmulKernel;

impl MatmulKernel {
    pub fn forward(&self, a: &Tensor, b: &Tensor) -> Result<Tensor> {
        a.apply_op2_no_bwd(b, self)
    }
}

impl CustomOp2 for MatmulKernel {
    fn name(&self) -> &'static str { "matmul_tiled" }

    fn cuda_fwd(&self,
        a_s: &CudaStorage, a_l: &Layout,
        b_s: &CudaStorage, b_l: &Layout,
    ) -> Result<(CudaStorage, Shape)> {
        let dev = a_s.device();
        let (m, k) = (a_l.dim(0)?, a_l.dim(1)?);
        let n = b_l.dim(1)?;

        let a = f32::as_cuda_slice(a_s)?.slice(a_l.start_offset()..);
        let b = f32::as_cuda_slice(b_s)?.slice(b_l.start_offset()..);
        let mut c = unsafe { dev.alloc::<f32>(m * n)? };

        let func = dev.get_or_load_custom_func(
            "matmul_tiled_kernel", "matmul_module", PTX)?;
        let cfg = LaunchConfig {
            grid_dim:         (m.div_ceil(32) as u32, n.div_ceil(32) as u32, 1),
            block_dim:        (128, 1, 1),
            shared_mem_bytes: 16384,
        };
        unsafe {
            dev.launch_builder(&func)
               .arg(&a).arg(&b).arg(&mut c)
               .arg(&(m as u32)).arg(&(n as u32)).arg(&(k as u32))
               .arg(&0u64).arg(&0u64)
               .launch(cfg)?;
        }
        Ok((f32::wrap_cuda_slice(c, dev.clone()), Shape::from(&[m, n])))
    }
}
```

Pure cudarc vs Candle CustomOp

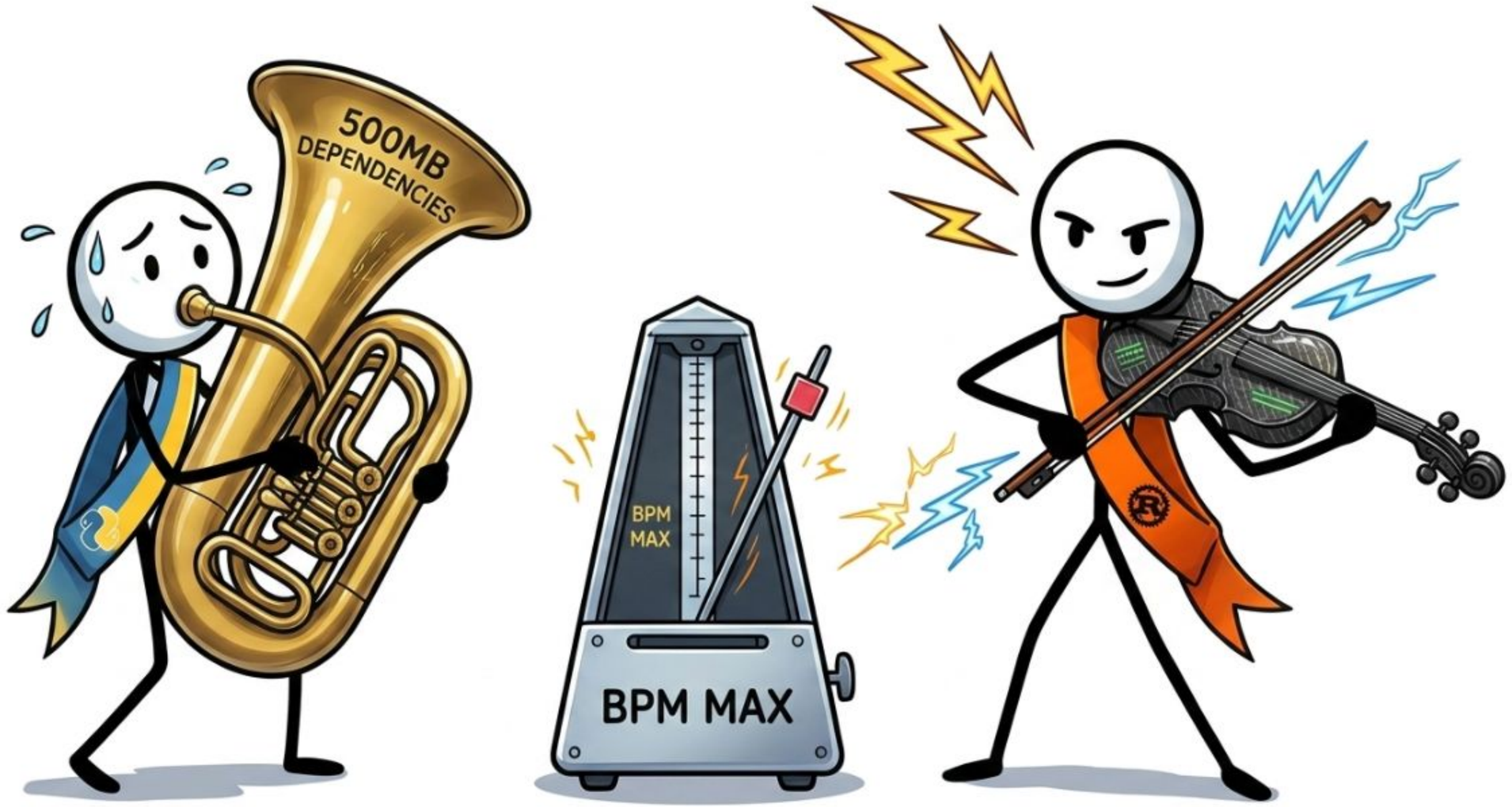
Pure cudarc

- ✓ Minimal dependencies
- ✓ Direct buffer control
- ✓ Works outside Candle ecosystem
- ⚠ Manual shape tracking
- ⚠ No layout offset handling
- ⚠ Returns `CudaSlice`, not `Tensor`

Candle CustomOp

- ✓ Automatic shape tracking
- ✓ `start_offset()` handles sliced tensors
- ✓ Device binding & grad graph
- ✓ Integrates with Candle model API
- ⚠ Candle dependency required

BENCHMARK

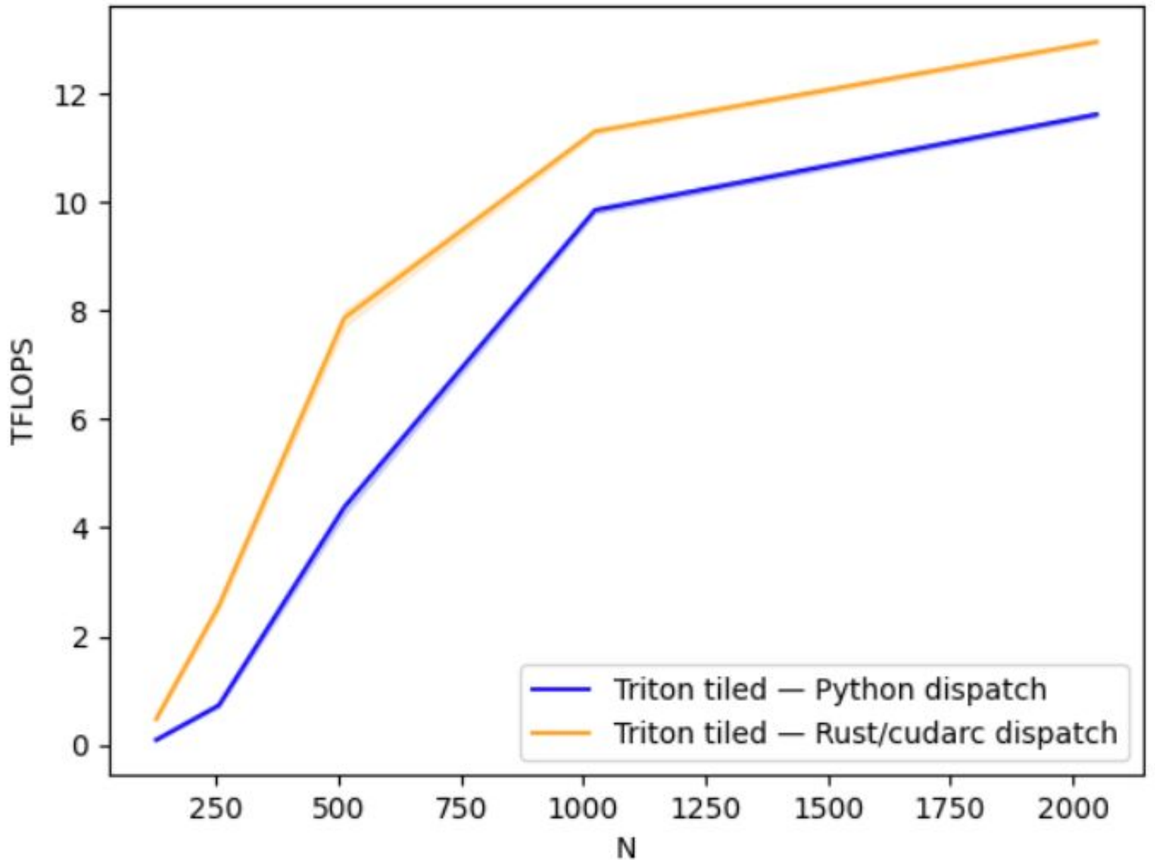


TRITON + PYTHON (RUBATO)

TRITON + RUST (PRESTO CON FUOCO)

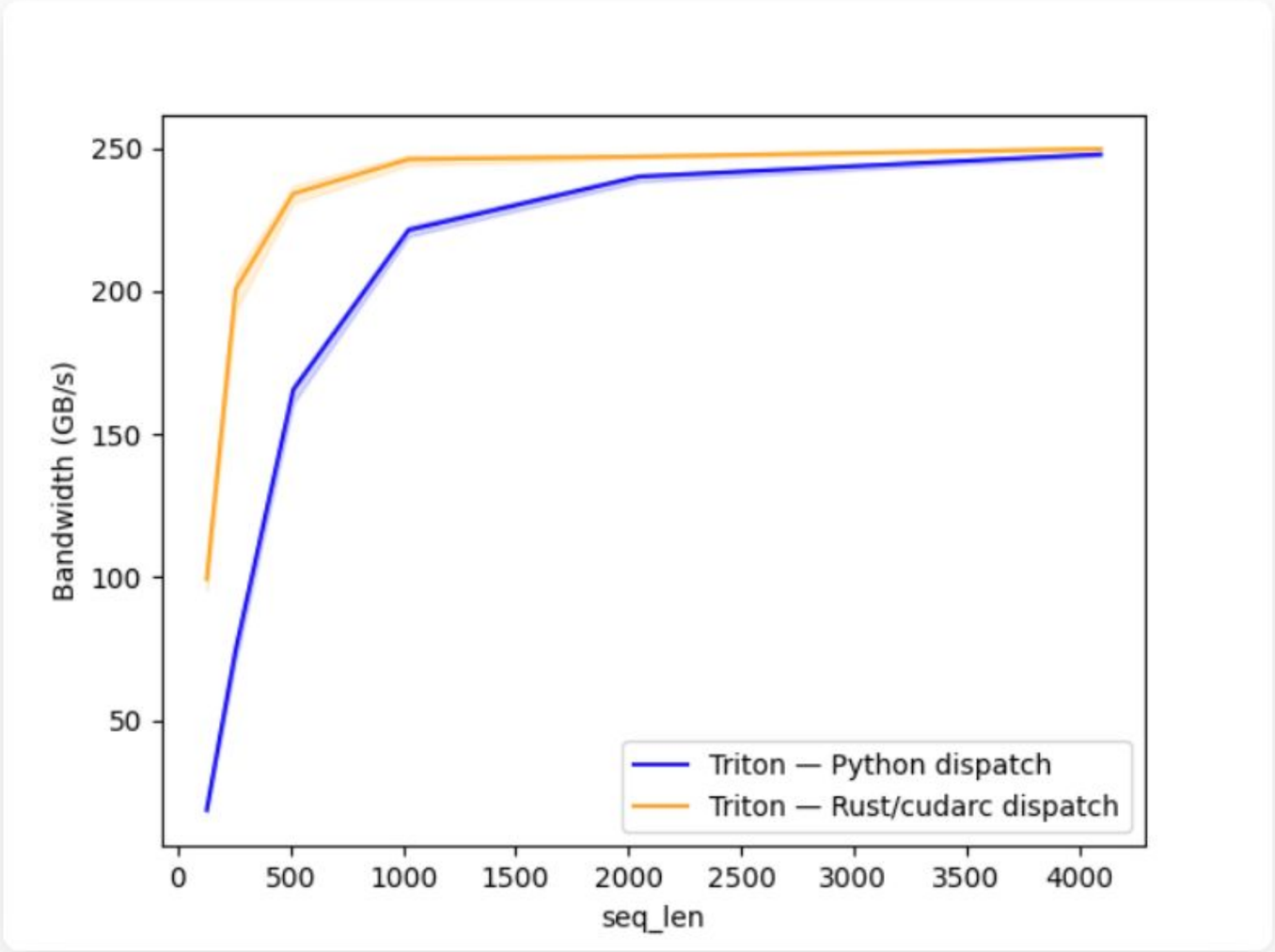
BENCHMARK: TILED MATMUL

TFLOPS (fp32) — Python vs Rust · RTX 4060 Ti



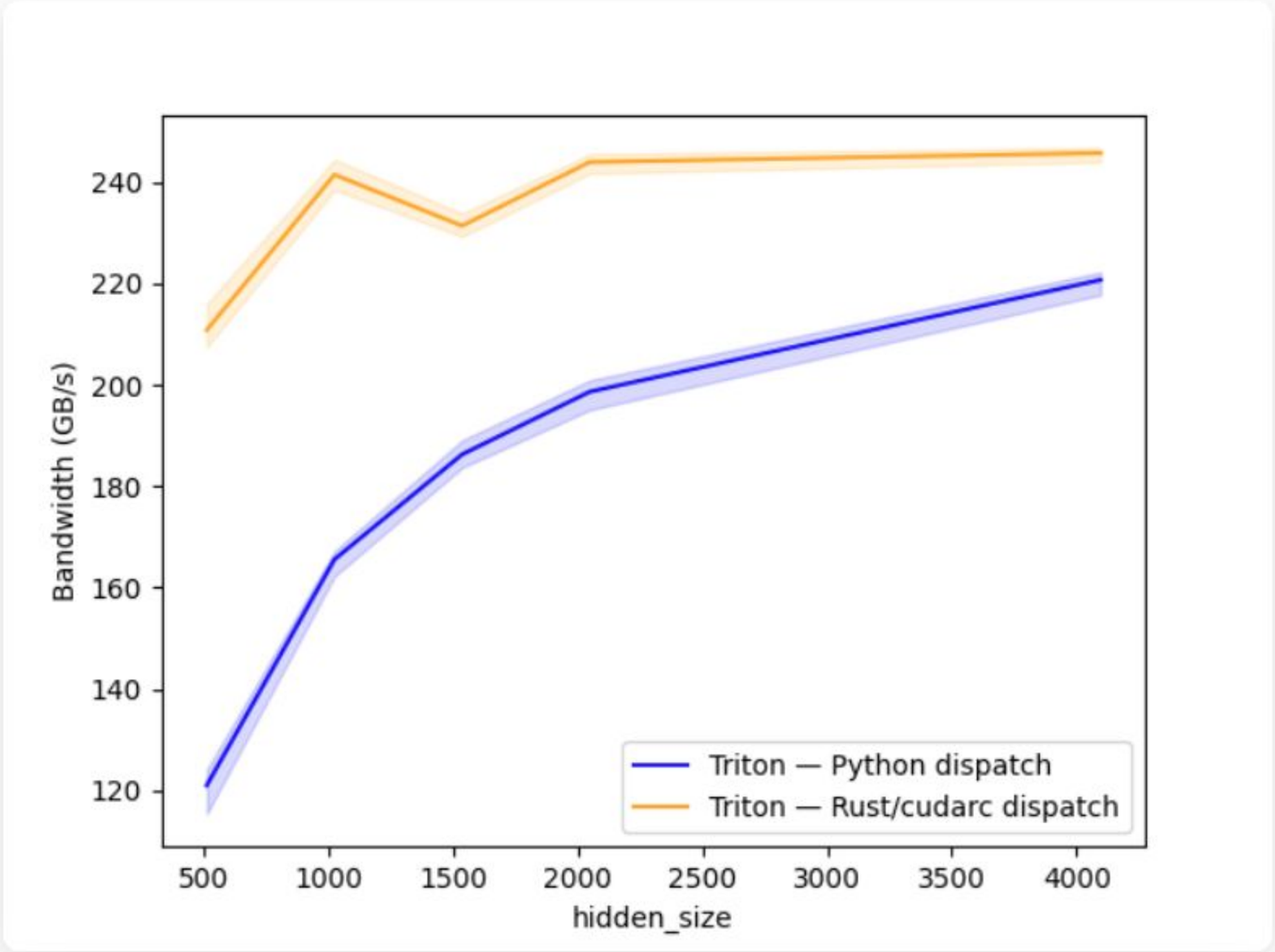
BENCHMARK: CAUSAL SOFTMAX

Cold-cache bandwidth (GB/s) — Python vs Rust · RTX 4060 Ti



BENCHMARK: RMSNORM

Cold-cache bandwidth (GB/s) — Python vs Rust · RTX 4060 Ti



BENCHMARKS (RTX 4060 TI)

Kernel	Size	Python dispatch	Rust/cudarc dispatch	Notes
Causal Softmax	seq_len=128	18.6 GB/s	**99.3 GB/s (5.3×)**	Dispatch overhead dominates
Causal Softmax	seq_len=2048	240 GB/s	247 GB/s (~1.0×)	DRAM-bound — both hit ~250 GB/s ceiling
RMSNorm	hidden=1536 *(Bielik)*	186 GB/s	**231 GB/s**	**Dip: next_pow2(1536)=2048 → 25% wasted BW**
RMSNorm	hidden=2048	199 GB/s	**244 GB/s**	No padding waste — full bandwidth
Tiled MatMul	N=128	0.09 TFLOPS	**0.48 TFLOPS (5.3×)**	Dispatch overhead dominates at small N
Tiled MatMul	**N=2048**	**11.6 TFLOPS**	**13.0 TFLOPS (1.12×)**	**TF32 tensor cores — compute-bound**

Small inputs: Rust dispatch 5× faster (saves 35–80 µs Python overhead)

Large inputs: kernels converge — DRAM/compute is the ceiling, not dispatch

WHEN IS THIS STACK THE RIGHT CHOICE?

Use it when…

- ✓ Serving a *fixed* model on *known* hardware
- ✓ Latency SLAs in the single-digit millisecond range
- ✓ Memory-bound decoding (every microsecond matters)
- ✓ Agentic loops — many sequential kernel launches
- ✓ Edge / embedded deployment (no Python runtime)
- ✓ You need custom ops Candle doesn't have

Skip it when…

- ⚠ Rapid architecture experimentation (use PyTorch)
- ⚠ Model changes frequently (recompile PTX each time)
- ⚠ Team unfamiliar with Rust + GPU programming
- ⚠ Standard cuBLAS/NCCL kernels are sufficient
- ⚠ Many different GPU SKUs in production fleet

SUMMARY

PRESENTATION


THE END