Lecture 5:

Generating High Performance Kernels for Image Processing and AI/ML Applications

Visual Computing Systems
Stanford CS348K, Spring 2026

Today's themes (mainly a discussion day, few slides)

- Discussion: Halide auto scheduler (Adams et al.)
- A quick intro into ML accelerator hardware
- Discussion: LLM-based kernel generation (AlphaEvolve/OpenEvolve)
- If time: open time for project discussions

Recall: What is the philosophy of Halide

- **Programmer** is responsible for describing an image processing algorithm
- **Programmer** has knowledge to schedule application efficiently on machine (but it's slow and tedious), so give programmer another language to express their high-level scheduling decisions
  - Loop structure of code
  - Unrolling / vectorization / multi-core parallelization

- **The system** (Halide compiler) is not smart, it provides the service of mechanically carrying out the nitty gritty details of implementing the schedule using mechanisms available on the target machine (pthreads, AVX intrinsics, CUDA code, etc.)
  - There are deviations from this philosophy in Halide? What are they?

Automatically generating Halide schedules

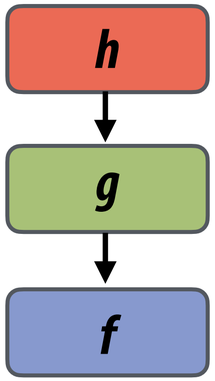
- **Problem: it turned out that very few programmers have the technical ability to write good Halide schedules**
  - Circa 2017… 80+ programmers at Google write Halide
  - Very small number trusted to write schedules

- **Recent work: Halide compiler analyzes the Halide program to automatically generate efficient schedules for the programmer [Mullapudi 2016, Adams 2019]**
  - As of Adams 2019, you'd have to work hard to manually author a schedule that is better than the schedule generated by the Halide autoscheduler for a complex image processing pipeline

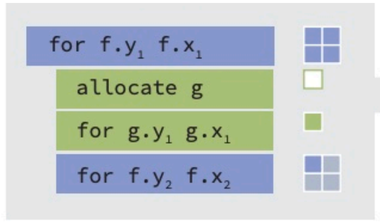
Modeling scheduling as a sequence of choices

- For each node *N* in the program DAG, starting from the end of the DAG…
  - Choose where to place current node *N* in the existing loop nest (determine N.compute_at())
  - Choose a tile sizes for *N* (assume outer dimension is parallel over threads, inner dimension is vectorized)
- Repeat until entire DAG is scheduled

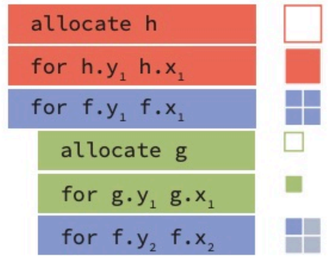
Example Halide DAG



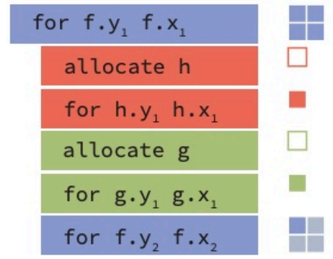
Current state of schedule (after scheduling node f and g)



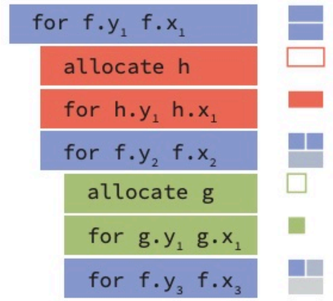
CANDIDATE SUCCESSOR STATES



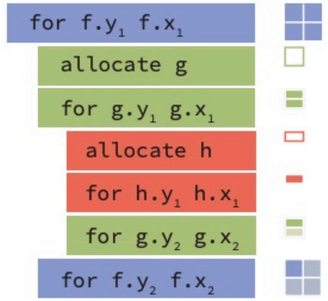
(a) compute at root



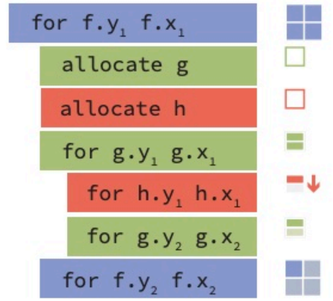
(b) compute h at an existing tiling



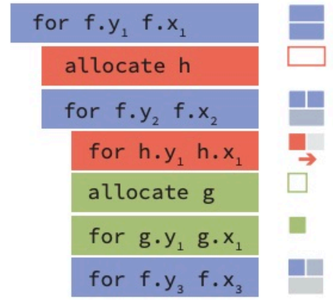
(c) compute h at a new outer tiling of f



(d) compute h at a new sub-tiling of g



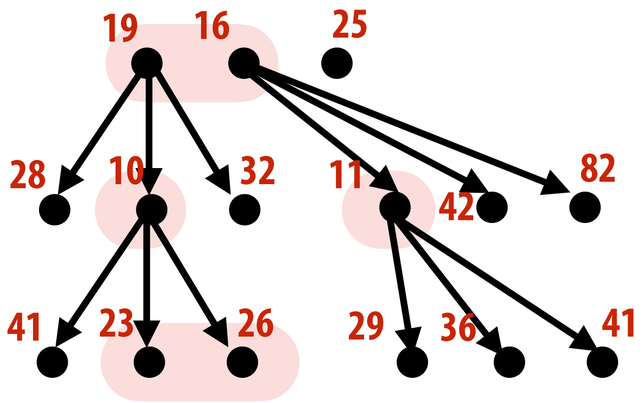
(e) store h at tiles of f, compute h at a new sub-tiling of g



(f) store h at a new outer tiling of f, compute h at sub-tiles of f

Use search to find best performing schedule

- Search over large space of schedules (e.g., greedy search, beam search)



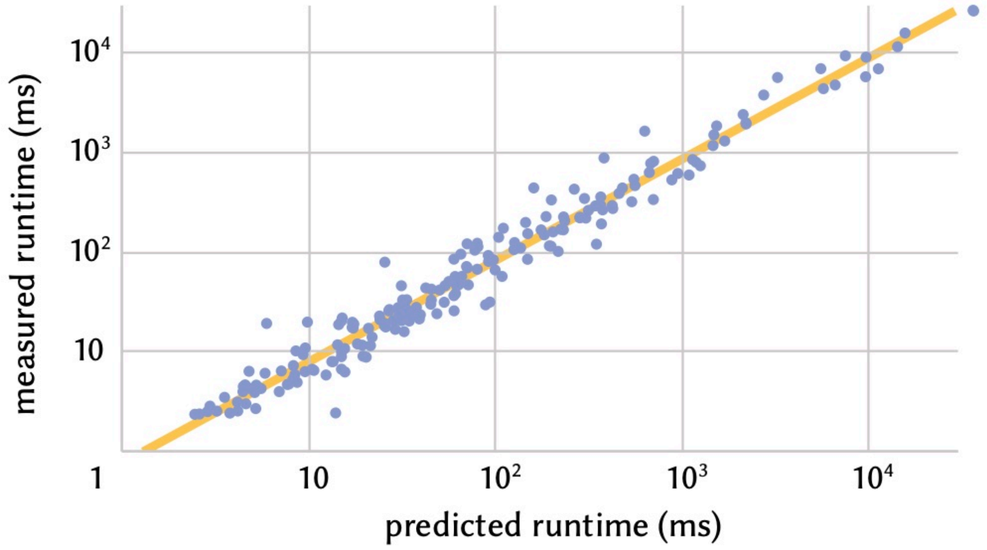
● = a partially scheduled DAG

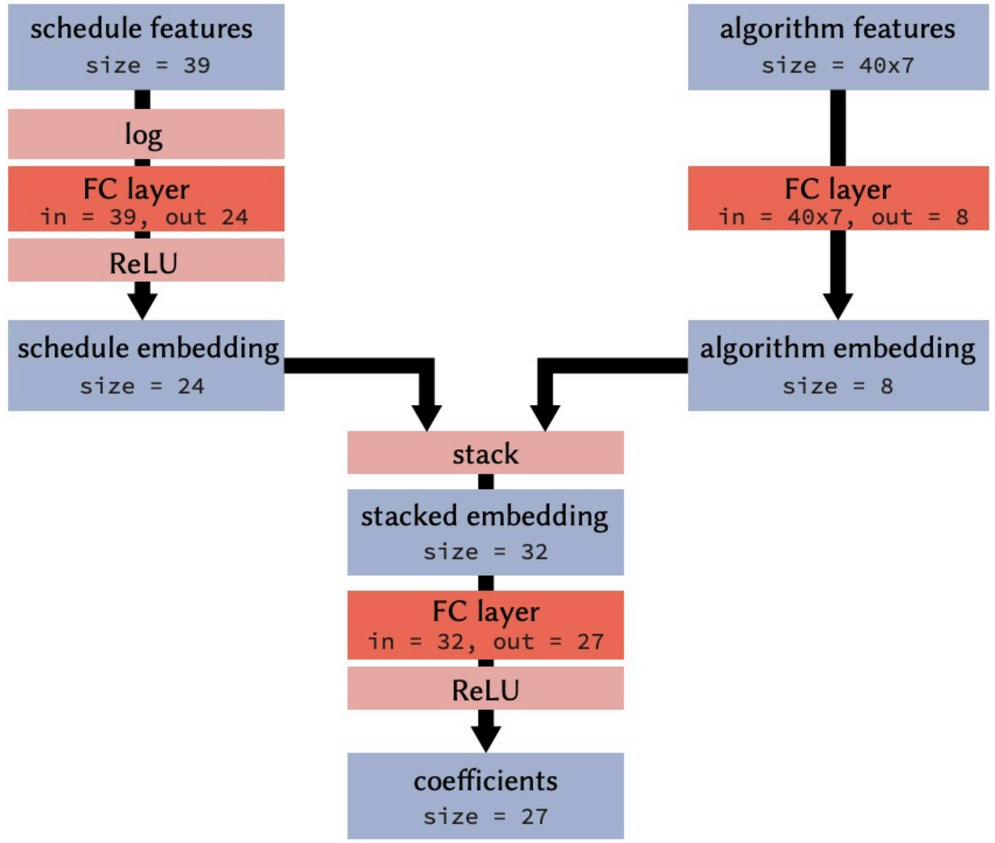
Number = estimated cost of schedule (as given so far)

- Challenge: might need to search over hundreds of thousands of possible schedules… how do we get the cost of a schedule?

Cost estimation using AI

- Given program + schedule… estimate cost *
- Simple MLP that runs in 10's microseconds per schedule (e.g., 1.4M schedules tested in 166 seconds)
  - Trained on a large database of randomly generated Halide programs
  - Training programs compiled and executed to get actual cost

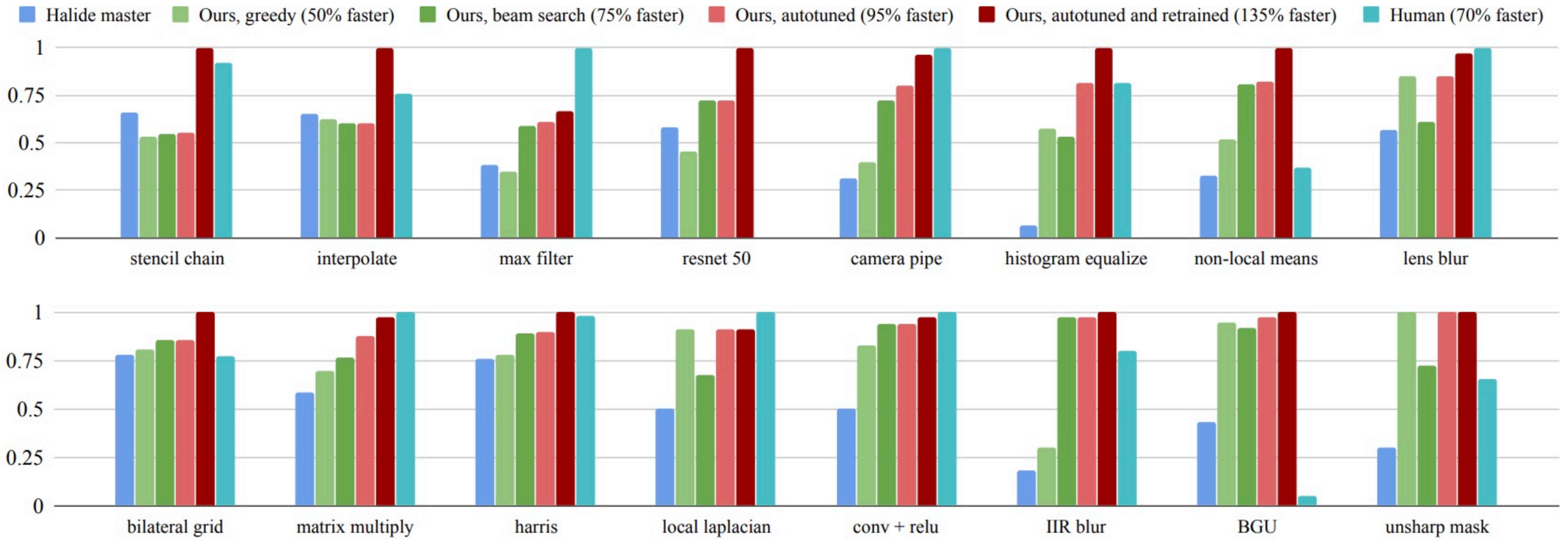




* in practice, doesn't directly compute cost… it outputs 27 coefficients that are plugged into a hand-crafted cost model

Autoscheduler comparable to best known human schedules

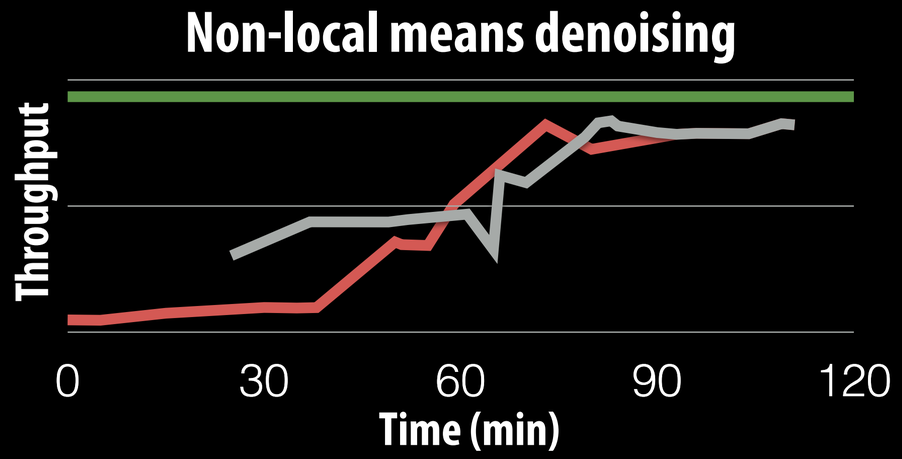
Graphs plot relative throughput (output pixels/second)

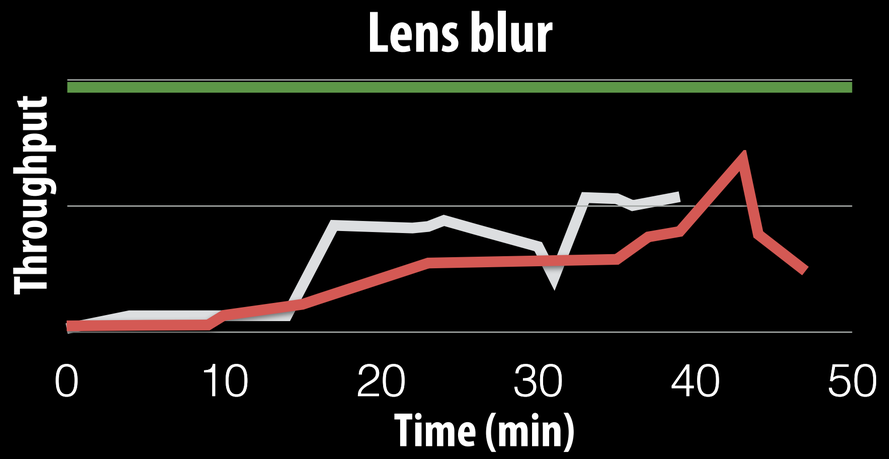


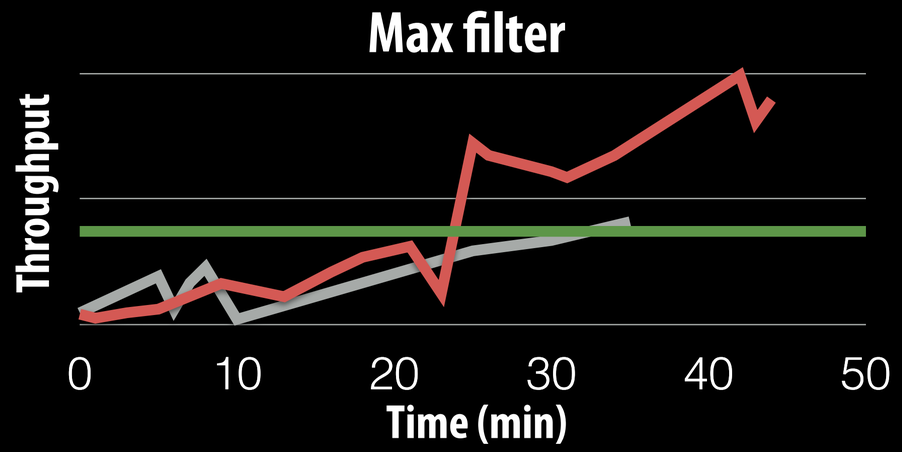
TL;DR - [Adams 2019], you'd have to work pretty hard to manually author a schedule that is better than the schedule generated by the Halide autoscheduler for image processing applications on CPUs

Autoscheduler saves time for experts

Earlier results from [Mullapudi 2016], not [Adams 2019]







Influence on code generation for ML applications

Example: Apache TVM

Apache TVM

An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators

- Schedule Primitives in TVM
  - split
  - tile
  - fuse
  - reorder
  - bind
  - compute_at
  - compute_inline
  - compute_root
  - Summary
- Reduction

Tuning Parameters of Thread Numbers

How to schedule the workload, say, 32x32 among the threads of one cuda block? Intuitively, it should be like t

```
num_thread_y = 8
num_thread_x = 8
thread_y = tvm.thread_axis((0, num_thread_y), "threadIdx.y")
thread_x = tvm.thread_axis((0, num_thread_x), "threadIdx.x")
ty, yi = s[Output].split(h_dim, nparts=num_thread_y)
tx, xi = s[Output].split(w_dim, nparts=num_thread_x)
s[Output].reorder(ty, tx, yi, xi)
s[Output].bind(ty, thread_y)
s[Output].bind(tx, thread_x)
```

There are two parameters in the schedule: `num_thread_y` and `num_thread_x`. How to determine the optimal Below is the result with Filter = [256, 1, 3, 3] and stride = [1, 1]:

Case	Input	num_thread_y	num_thread_x
1	[1, 256, 32, 32]	8	32
2	[1, 256, 32, 32]	4	32
3	[1, 256, 32, 32]	1	32
4	[1, 256, 32, 32]	32	1

Many interesting observations from above results:

Motivation for AI Accelerators

Data movement has high energy cost

- Rule of thumb in system design: always seek to reduce amount of data transferred from memory

- "Ballpark" numbers
  - Integer op: ~ 1 pJ *
  - Floating point op: ~20 pJ *
  - Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ

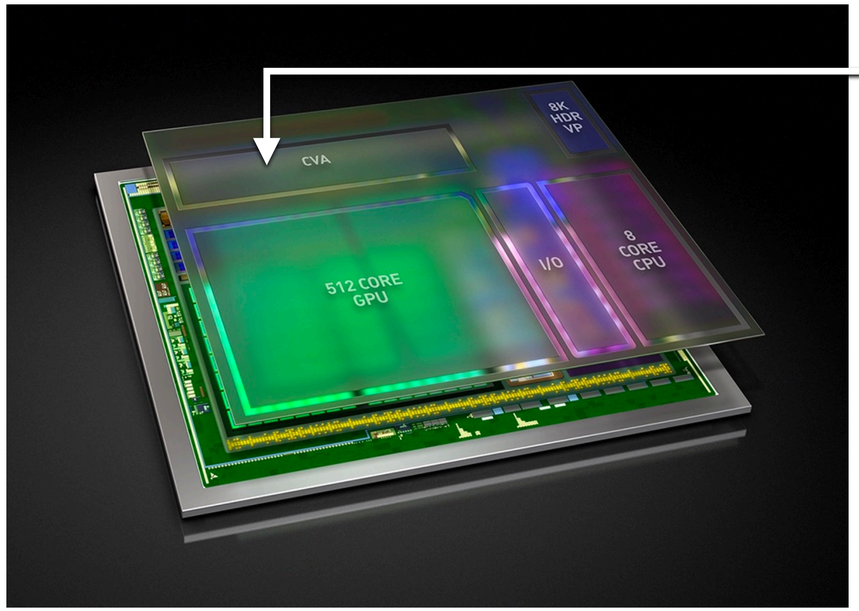
[Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]

- Implications
  - Reading 10 GB/sec from memory: ~1.6 watts
  - Entire power budget for mobile GPU: ~1 watt
    (remember phone is also running CPU, display, radios, etc.)
  - iPhone 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
  - Exploiting locality matters!!!

* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

Efficiency estimates of "coarser granularity" math ops *

- Estimated overhead of programmability (instruction stream, control, etc.)
  - Half-precision FMA (fused multiply-add) — 2000%
  - Half-precision DP4 (vec4 dot product) — 500%
  - Half-precision 4x4 MMA (matrix-matrix multiply + accumulate) — 27%



NVIDIA Xavier (SoC for automotive domain)

Features a Computer Vision Accelerator (CVA), a custom module for deep learning acceleration (large matrix multiply unit)

~ 2x more efficient than NVIDIA V100 MMA instruction despite being highly specialized component. (includes optimization of gating multipliers if either operand is zero)

* Estimates by Bill Dally using academic numbers, SysML talk, Feb 2018

Ampere GPU SM (A100)

Each SM core has:

64 fp32 ALUs (mul-add)

32 int32 ALUs

4 "tensor cores"

Execute 8x4 x 4x8 matrix mul-add instr

A x B + C for matrices A,B,C

A, B stored as fp16, accumulation with fp32 C

There are 108 SM cores in the GA100 GPU:

6,912 fp32 mul-add ALUs

432 tensor cores

1.4 GHz max clock

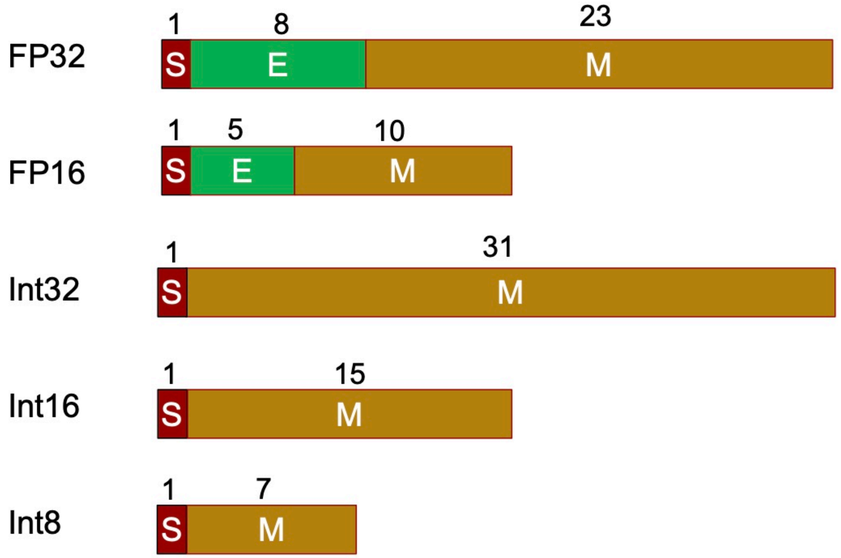
= 19.5 TFLOPs fp32

+ 312 TFLOPs (fp16/32 mixed) in tensor cores

Single instruction to perform 2x8x4x8 FP16 + 8x8 TF32 ops

The NVIDIA tensor core approach is an evolutionary design: add DNN-specific instructions to a traditional programmable processor ("evolve, don't replace")

Numerical data formats



Reminder:

$-1^{S} \times (1 + (M \times 2^{-23})) \times 2^{(E-127)}$

BF16: Same range as FP32, but lower accuracy

0 - 448

0 - 57344

Slide credit: Bill Dally

Summary: implications to SW developers

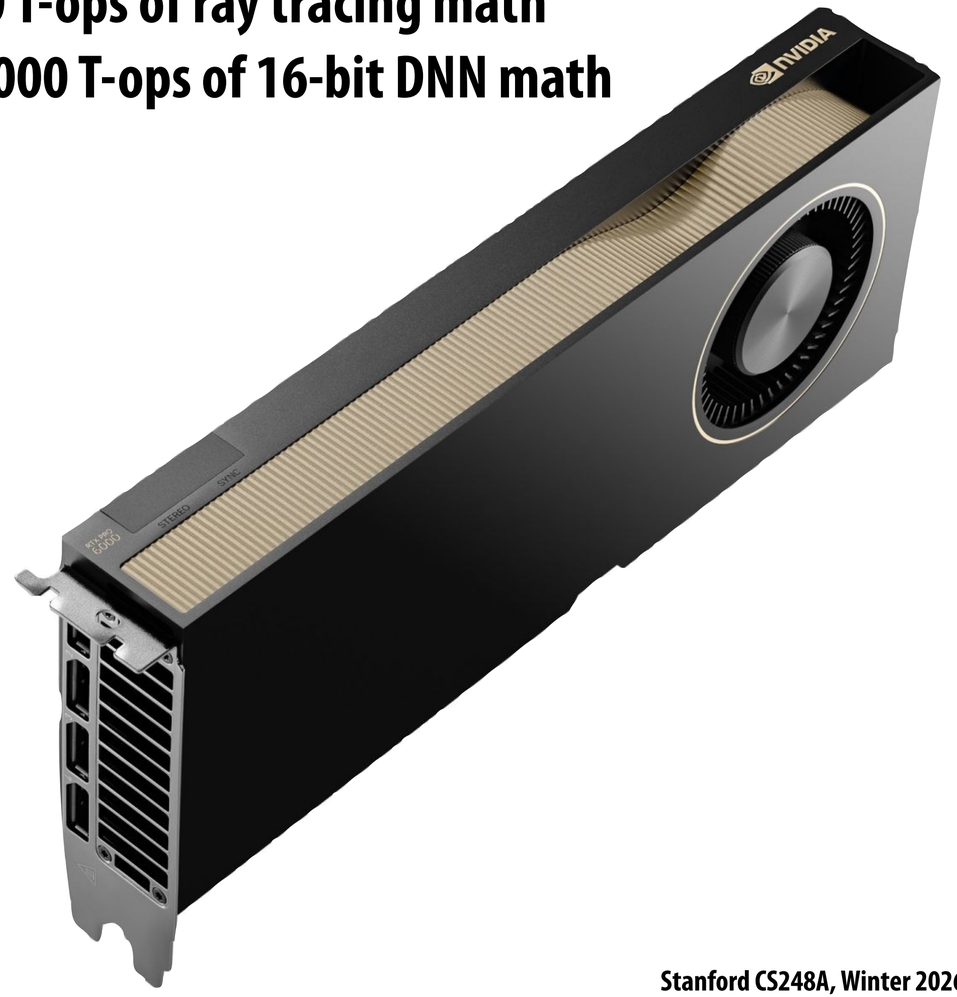
- A significant fraction of the compute capability of a modern GPU is in the special purpose acceleration of neural networks (Tensor Cores)

- So "runs on the GPU" is quickly becoming paramount to "runs on the tensor cores"

- The more of your algorithm you can offload to the tensor cores, the more you get to use this highly-efficient custom hardware
  - Something to keep in mind later in the class… Overall there will be wins when approximating algorithms via neural approximations is not too much (10-50x) more inefficient than original algorithm

NVIDIA RTX 6000 Pro GPU

125 T-ops of FP32 math
380 T-ops of ray tracing math
~1000 T-ops of 16-bit DNN math



Generating efficient code for AI accelerators

What are the right primitives?

- Halide adopts low-level primitives (mathematical expressions on values) for expressing image processing applications

- DNN operations often built on higher-level primitives like matrix multiplication ops, movement of matrices between locations in memory

- Could a Halide-like scheduling approach apply to different sets of primitives?

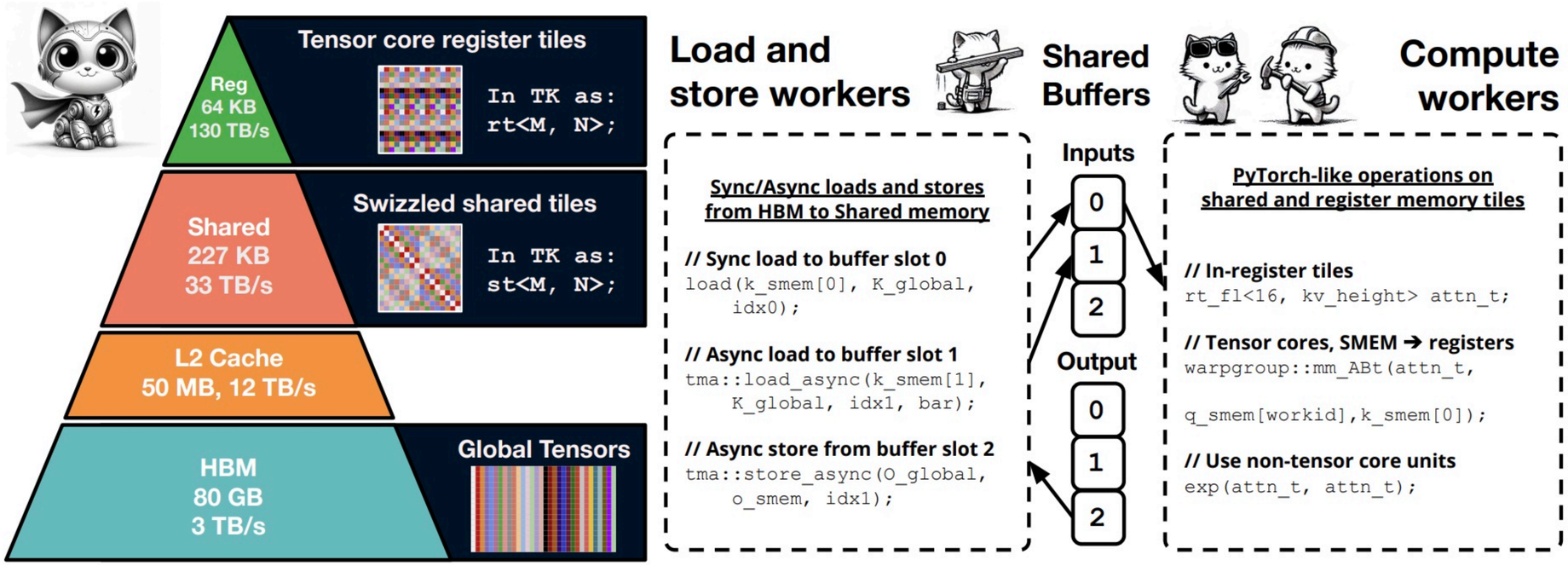
ThunderKittens: A Simple Embedded DSL for AI kernels

Benjamin Spector, Aaryan Singhal, Simran Arora, Chris Re

(This post is an extended introduction to a longer post we're releasing concurrently.)

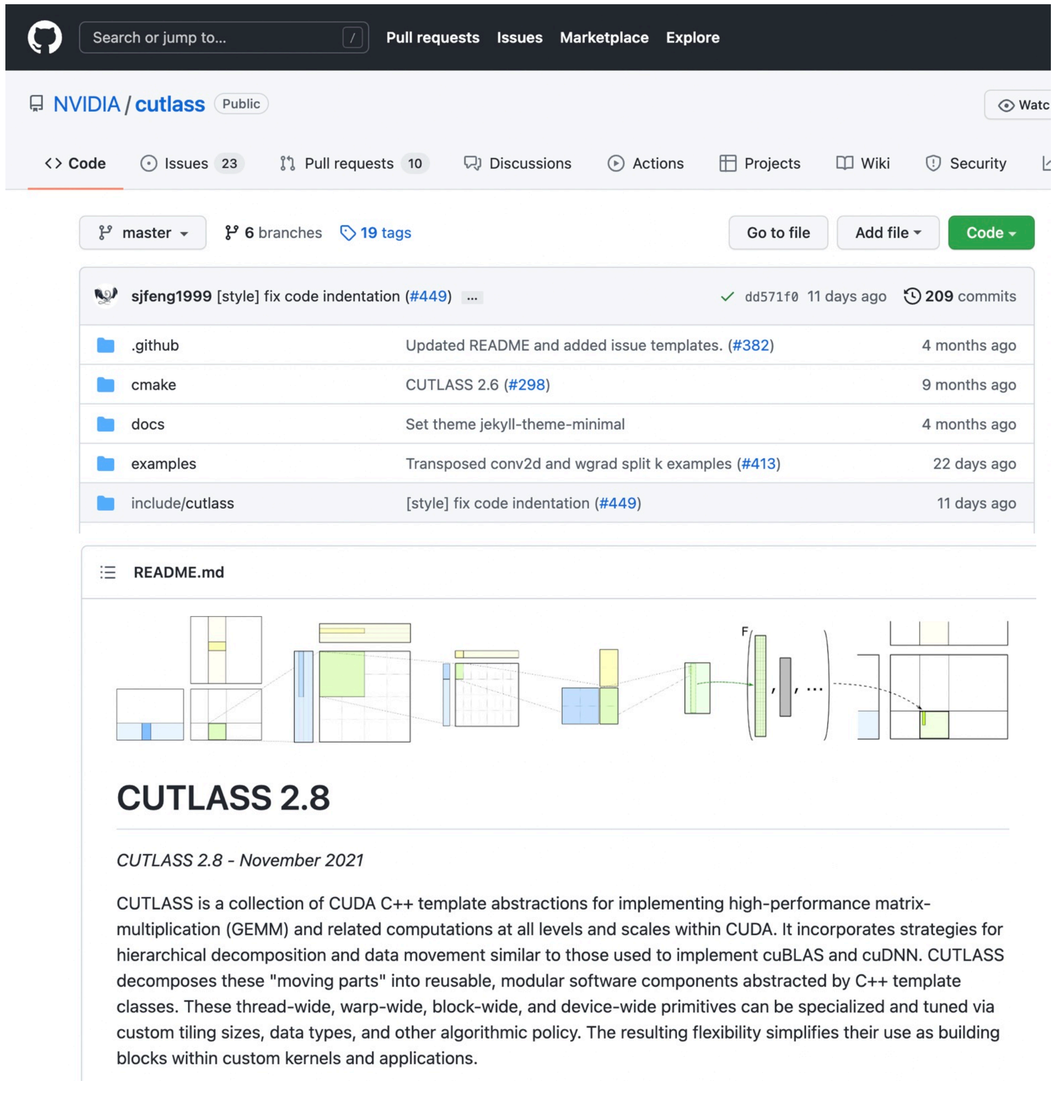
Thunderkittens

- CUDA library of useful tile-based programming primitives
- Intended to make advanced developers (CS149-level folks) more productive writing blocked code
  - Async load/store of tiles
  - Support for advanced memory layouts (blocked tiles, interleaved elements, etc.)



NVIDIA CUTLASS

Basic primitives/building block for implementing your custom high performance DNN layers. (e.g, unusual sizes that haven't been heavily tuned by cuDNN)



Fast (in-shared memory) GEMM

Fast WARP level GEMMs

Iterators for fast block loading/tensor indexing

Tensor reductions

Etc.

CuTile

cuTile Python

cuTile is a parallel programming model for NVIDIA GPUs and a Python-based DSL. It automatically leverages advanced hardware capabilities, such as tensor cores and tensor memory accelerators, while providing portability across different NVIDIA GPU architectures. cuTile enables the latest hardware features without requiring code changes.

cuTile kernels are GPU programs that are executed in parallel on a logical grid of blocks. The `@ct.kernel` decorator marks a Python function as a kernel's entry point. Kernels cannot be called directly from the host code; the host must queue kernels for execution on GPU using the `ct.launch()` function:

```
import cuda.tile as ct
import cupy

TILE_SIZE = 16

# cuTile kernel for adding two dense vectors. It runs in parallel on the GPU.
@ct.kernel
def vector_add_kernel(a, b, result):
    block_id = ct.bid(0)
    a_tile = ct.load(a, index=(block_id,), shape=(TILE_SIZE,))
    b_tile = ct.load(b, index=(block_id,), shape=(TILE_SIZE,))
    result_tile = a_tile + b_tile
    ct.store(result, index=(block_id,), tile=result_tile)

# Host-side function that launches the above kernel.
def vector_add(a: cupy.ndarray, b: cupy.ndarray, result: cupy.ndarray):
    assert a.shape == b.shape == result.shape
    grid = (ct.cdiv(a.shape[0], TILE_SIZE), 1, 1)
    ct.launch(cupy.cuda.get_current_stream(), grid, vector_add_kernel, (a, b, result))
```

Kernels move data between arrays and tiles using functions like `ct.load()` and `ct.store()`. Both arrays and tiles are tensor-like data structures: each has a specific shape (i.e., the number of elements

Triton

- Language support for operations that load/store tensors
- Load “blocks” of data into GPU shared memory
- Perform data-parallel operations on those blocks

A simple blocked matrix multiplication

```
# Do in parallel
for m in range(0, M, BLOCK_SIZE_M):
  # Do in parallel
  for n in range(0, N, BLOCK_SIZE_N):
    acc = zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=float32)
    for k in range(0, K, BLOCK_SIZE_K):
      a = A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K]
      b = B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]
      acc += dot(a, b)
    C[m : m+BLOCK_SIZE_M, n : n+BLOCK_SIZE_N] = acc
```

Triton

Full Triton reference implementation: two levels of blocking

```
@triton.jit
def matmul_kernel(
        # Pointers to matrices
        a_ptr, b_ptr, c_ptr,
        # Matrix dimensions
        M, N, K,
        # The stride variables represent how much to increase the ptr by when moving by 1
        # element in a particular dimension. E.g. `stride_am` is how much to increase `a_ptr`
        # by to get the element one row down (A has M rows).
        stride_am, stride_ak,  #
        stride_bk, stride_bn,  #
        stride_cm, stride_cn,
        # Meta-parameters
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,  #
        GROUP_SIZE_M: tl.constexpr,  #
        ACTIVATION: tl.constexpr  #
):
    """Kernel for computing the matmul C = A x B.
    A has shape (M, K), B has shape (K, N) and C has shape (M, N)
    """
    # -----------------------------------------------------------
    # Map program ids `pid` to the block of C it should compute.
    # This is done in a grouped ordering to promote L2 data reuse.
    # See above `L2 Cache Optimizations` section for details.
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m
```

```
    # -----------------------------------------------------------
    # Add some integer bound assumptions.
    # This helps to guide integer analysis in the backend to optimize
    # load/store offset address calculation
    tl.assume(pid_m >= 0)
    tl.assume(pid_n >= 0)
    tl.assume(stride_am > 0)
    tl.assume(stride_ak > 0)
    tl.assume(stride_bn > 0)
    tl.assume(stride_bk > 0)
    tl.assume(stride_cm > 0)
    tl.assume(stride_cn > 0)

    # -----------------------------------------------------------
    # Create pointers for the first blocks of A and B.
    # We will advance this pointer as we move in the K direction
    # and accumulate
    # `a_ptrs` is a block of [BLOCK_SIZE_M, BLOCK_SIZE_K] pointers
    # `b_ptrs` is a block of [BLOCK_SIZE_K, BLOCK_SIZE_N] pointers
    # See above `Pointer Arithmetic` section for details
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)

    # -----------------------------------------------------------
    # Iterate to compute a block of the C matrix.
    # We accumulate into a `[BLOCK_SIZE_M, BLOCK_SIZE_N]` block
    # of fp32 values for higher accuracy.
    # `accumulator` will be converted back to fp16 after the loop.
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        # Load the next block of A and B, generate a mask by checking the K dimension.
        # If it is out of bounds, set it to 0.
        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)
        # We accumulate along the K dimension.
        accumulator = tl.dot(a, b, accumulator)
        # Advance the ptrs to the next K block.
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    # You can fuse arbitrary activation functions here
    # while the accumulator is still in FP32!
    if ACTIVATION == "leaky_relu":
        accumulator = leaky_relu(accumulator)
    c = accumulator.to(tl.float16)

    # -----------------------------------------------------------
    # Write back the block of the output matrix C with masks.
    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)
```

Trial and error via reflection

Starting code (e.g., PyTorch) + LLM prompt

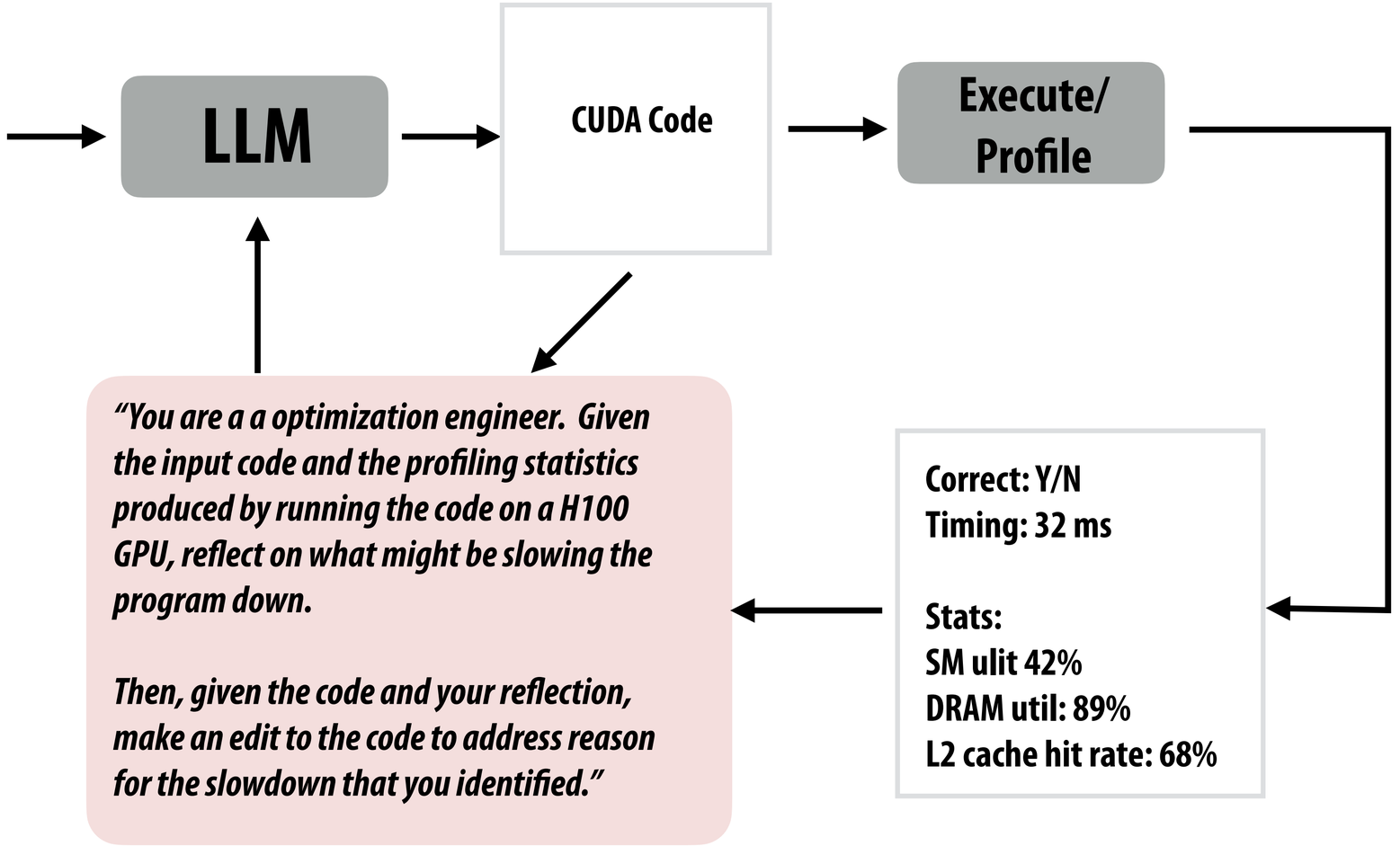
```python
class Model(nn.Module):
    """
    Simple model that performs a matrix multiplication, scales the result, and applies ba
    """
    def __init__(self, in_features, out_features, scale_shape, eps=1e-5, momentum=0.1):
        super(Model, self).__init__()
        self.gemm = nn.Linear(in_features, out_features)
        self.scale = nn.Parameter(torch.randn(scale_shape))
        self.bn = nn.BatchNorm1d(out_features, eps=eps, momentum=momentum)

    def forward(self, x):
        x = self.gemm(x)
        x = x * self.scale
        x = self.bn(x)
        return x

batch_size = 16384
in_features = 4096
out_features = 4096
scale_shape = (out_features,)
```

"You are a performance optimization engineer in CS149. Please rewrite the following PyTorch code as high performance code in CUDA."

Keep in mind the following code optimization principles we discussed in class…

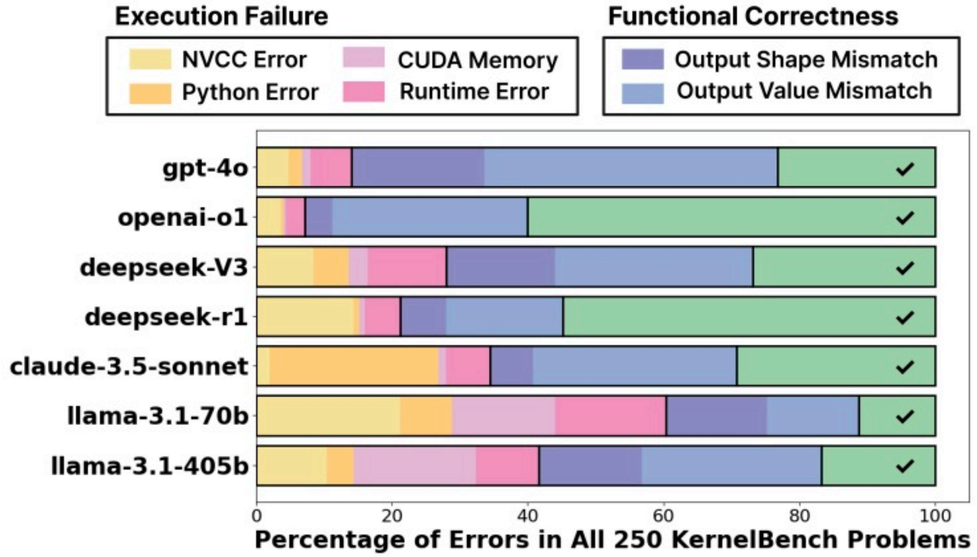


KernelBench

- **A benchmark of hundreds of PyTorch kernels**
- **LMM agent's goal is to automatically produce fast and correct CUDA kernels**

We construct KernelBench to have 4 Levels of categories:

- **Level 1** 🧱: Single-kernel operators (100 Problems) The foundational building blocks of neural nets (Convolutions, Matrix multiplies, Layer normalization)
- **Level 2** 🔗: Simple fusion patterns (100 Problems) A fused kernel would be faster than separated kernels (Conv + Bias + ReLU, Matmul + Scale + Sigmoid)
- **Level 3** ⚛️: Full model architectures (50 Problems) Optimize entire model architectures end-to-end (MobileNet, VGG, MiniGPT, Mamba)
- **Level 4** 🤗: Level Hugging Face Optimize whole model architectures from HuggingFace



Discussion time!

- LLM-based kernel generation
- Vs.
- Brute force search
- Vs.
- Combining the two!