2025.02.01

tinygrad decomposes them so there's only "reciprocal", but it's a slight loss of expressivity (and convenience!) wrt. "to-power-of"

in OCANNL it would be slightly harder to do complex gradients like "to-power-of" and "division"

reduces the scope of the tensor class a bit but tensors still have grad field

no need to centralize for that

inspiring: instead of tracking backprop at the assignments level, track it at the tensor expression level like in tinygrad

tinygrad's centralized gradient

we get rid of the diff / non-diff distinction, replace `diff : diff option` by mutable diff : diff option, and `type diff = { grad : t; zero_grads : asgns }`

it shouldn't reduce flexibility since the new `grad.forward` would function like the old `diff.bprop`

we will still use the global session state: to update the diff field

i.e. there will still be an implicit set of differentiated roots; later, we could refactor that too

2025.02.02

includes a mini primer on MLIR

To Be Continued

PLISS 2019 Polyhedral Compilation lecture slides

PolyTOPS: Reconfigurable and Flexible Polyhedral Scheduler

Tiling

Loop Fusion and Fission

optimizing nested loops by affine transformations of loop variables

here dataflow perspective!

polyhedra are a technical detail

quick glance at polyhedral optimization

T: topology with nodes-cols and buffers-rows
v: node firings
T v = 0: no deadlock or buffer overflow
$v_i > 0 \; \forall i$: no starvation

ISL: integer set library

schedule: map from timestamp to iteration domain (using lexicographic order)

for example,
minimize: q_p, q_c, d_p, d_c
subject to:
$q_c * x + d_c \geq q_p * \frac{x}{2} + d_p$

construct a producer/consumer dataflow diagram, reorder for locality, and compactify buffers by reusing alread-read cells

simpler -- synchronous dataflow:
+ finds efficient, deadlock free schedules
- limited solutions: $S_p(x) = q_p * x + d_p$
- same amount of data at each node
q: rate, d: dealy, p: producer, c: consumer

optimal solving is too hard, and efficient code generation from a solution is too hard

2025.02.03

iteration domain: space of all executions of statements i.e. single-cell assignments

actually, times are also lexicographic: output dims? input dims

reuse

cache lines

temporal vs spatial locality

schedule = maps iteration domain to execution times preserving lexical order

loops introduce one dimension to the iteration domain per nesting depth

valid schedule: dependencies have lexico-positive distance, schedule is invertible

anyway, causality preserved

statement sequences introduce one dimension per sequence at nesting depth ("unrolled loop")

code generation: invert the schedule; affine eqs. and ineqs. schedules invertible (Gauss Elimination, Fourier-Motzkin)

Parallelism: benefit distance-0 dependences

(called B-vectors if every other dimension is for sequencing, and uniquely identify statements)

polyhedral optimization framework

optimizing schedule: ILP using heuristic objectives (trying to approx. performance on idealized hardware); or discrete search with benchmarking JITted solutions

Reuse: benefit accesses not using iterators from the last schedule dimension

Fusion (when the sequence dim difference is zero): penalty for fissioned statements sharing a data cell, OR penalty for fused statements not sharing any data cells

when ILP, ensure invertibility e.g. iteratively optimize over rows adding constraints to keep full row rank

Vectorization: penalty for iterator in the fastest-varying array subscript and no others

2025.02.04

a statement can have multiple relations if disjunctive conditions in control flow

change in a scheduling relation = program transformation

"Opening Polyhedral Compiler's Black Box" from Inria

requirements -- invertibility not necessary:
- all relations have the same iteration domain
- union of relations of a statement covers the iteration domain and is 1-to-1
- all dates are (vectors of) integers

implies linear independence of dimensions

decodes optimized scheduling relations into a sequence of syntactic loop transformations

encodes loop transformations as updates of scheduling relations

"High performance compilers for parallel computing" M. Wolfe 1996

transforms not needed for decoding:
Interchange, Collapse, Linearize, Parallelize

syntactic transformations

Reorder
changes sequencing

FuseNext
fuses a loop with its successor

Skew
scales the given **output** loop's iterator by given factor

Distribute
splits a loop in two

Interchange
swaps loops at given 2 depths in the given nesting

Reverse
reverses the iteration order for the given loop(s)

Shift
moves given statements by given amount in the given loop

Reshape
scales the given **input** loop's iterator by given factor

IndexSetSplit
partitions a scheduling relation into two disjoint ones, e.g. duplicates a loop with one up to midpoint, the next from midpoint

StripMine
decomposes the given loop into 2 nested loops, with one exec of the inner loop iterating for at most the given steps

+Interchange = Tiling

Collapse
reverse of IndexSetSplit

Grain **Densify**
change how long it takes between two executions of the given statements along the given loop

Linearize
reverse of StripMine

Unroll/Parallelize
not a real transformation, marks the given output dimension for unrolling or vectorizing

Grain + Reshape = Rotation: aligns a dependence along a loop

2025.02.05

another glance at MLIR: Multi-Level Intermediate Representation

all within MLIR

Lowering

TensorFlow repr.

XLA HLO

Polyhedral/Affine

LLVM IR

nvgpu dialect
PTX specific operations

amdgpu dialect

llvm dialect

tensor type dialect

memref type dialect views on arrays

"Polyhedral" / for-loop lang with first-class loop ranges
Affine dialect

GPU dialect
launching kernels

shape dialect
shape inference

Linalg dialect
high-level hierarchical optimization for linalg ops, especially convolutions

Structured Abstraction

Transformation

many improvements over LLVM

dialects are centrally or externally maintained extensions

typed Static Single Assignment, functional: instead of Phi functions, blocks take arguments

48 dialects in the repository

declarative general programming language with namespaces, pattern matching, rewrite rules

combined with C++ code (tricky rewrite rules, passes, trait implementations etc.)

e.g. can define syntax of operations from within op. def.

altogether 238 passes, e.g.: linalg-fuse-elementwise-ops, gpu-map-parallel-loops, affine-loop-tile, convert-gpu-to-nvvm, convert-amdgpu-to-rocdl, convert-affine-for-to-gpu, buffer-loop-hoisting, ownership-based-buffer-deallocation, loop-invariant-code-motion

high-level support for H100:
`nvgpu.mbarrier` ops (Arrive/Wait Bariers),
`nvgpu.tma` ops (Tensor Memory Accelerator),
`nvgpu.warpgroup` MMA ops

Lowering sequence

nvgpu --> nvvm --> llvm --> PTX

2025.02.06

iteration domain:

$$D_S = \{ \vec{it} \mid M_S \cdot \begin{pmatrix} \vec{it} \\ \vec{N} \\ 1 \end{pmatrix} \geq 0 \}$$

dependency legality:

$$\delta_{S \to R} = \{ \begin{pmatrix} \vec{it}_S \\ \vec{it}_R \end{pmatrix} \mid M_{S \to R} \cdot \begin{pmatrix} \vec{it}_S \\ \vec{it}_R \\ \vec{N} \\ 1 \end{pmatrix} \geq 0 \}$$

scheduling function:

$$\theta_S = \vec{it} \mapsto (\phi_{S,0}(\vec{it}), \dots, \phi_{S,m-1}(\vec{it}))$$

$$\phi_{S,i}(\vec{it}) = \vec{T_{S,i}} \cdot \begin{pmatrix} \vec{it} \\ \vec{N} \\ 1 \end{pmatrix}$$

(vector)

(symbolic) iterators: $\vec{it}$
(symbolic) constants: $\vec{N}$
dependency S->R: statement S needs to execute before R to preserve semantics

very clean formal setup of polyhedral optimization, nice scheduler algo

supports user-defined constraints, cost functions

tiling, skewing and intra-tile optims are not done by ILP --> postprocessing at each step

PolyTOPS: Reconfigurable and Flexible Polyhedral Scheduler

PolyTOPS builds ILPs one scheduling dimension at a time, from outermost.

$$(\vec{it}, \vec{it}') \in \delta_{S \to R} \Rightarrow \phi_{R,i}(\vec{it}') \geq \phi_{S,i}(\vec{it})$$

More efficient but doesn't encode global cost functions.

predefined cost funcs

proximity: temporal locality

contiguity: spatial locality

feautrier: find sequential outer dimensions that carry the most dependencies -- enables inner loop parallelism / vectorization

AutoVectorization: which loops should be innermost and unfused, based on mem stride and accesses

bigLoopsFirst: loops with largest domains outermost

directives: specify that a loop should be parallel / vectorized / sequential, if possible

2025.02.07

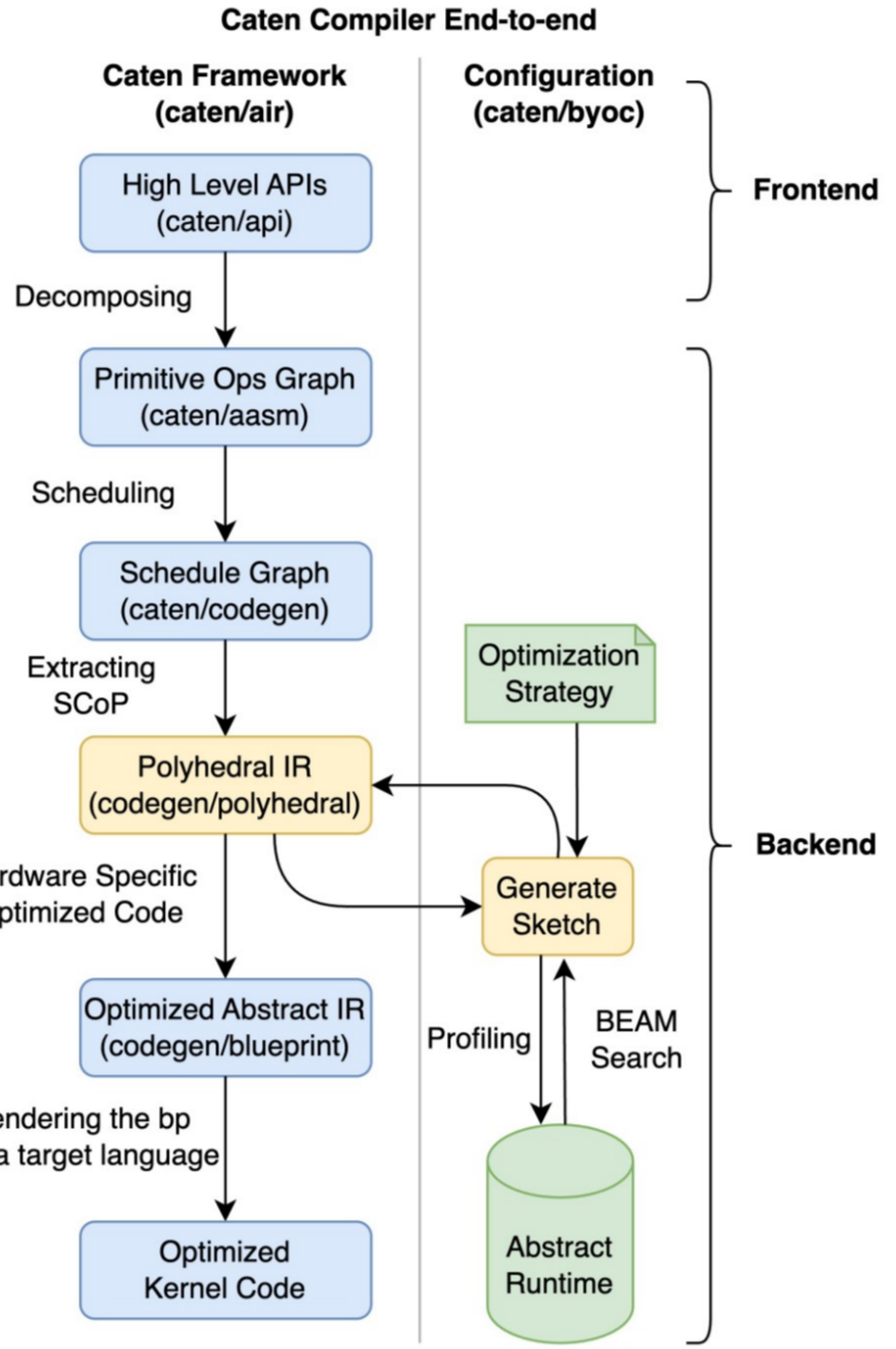
C, Lisp, JS/WebGPU runtimes

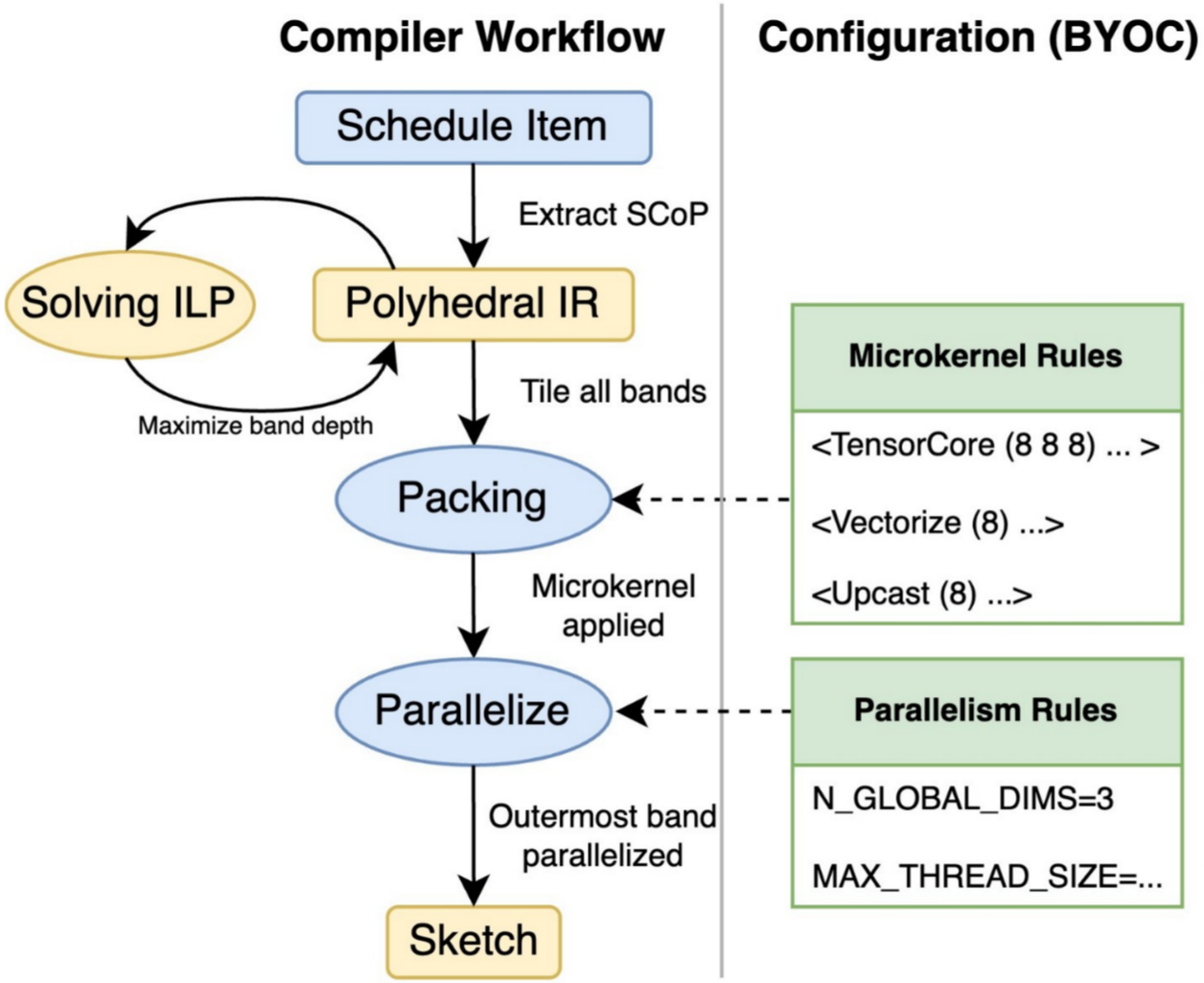
renamed shape-inference to type-relay, handles shape, stride, iter spaces (i.e. OCANNL's projections

has frontend but can also load ONNX graphs

second glance at Caten

https://github.com/hikettei/Caten





https://github.com/hikettei/Caten/pull/422

2025.02.10

for loop optimization →

Integer Set Library

well documented: a manual, and a tutorial on polyhedral compilation

continuously since mid-2008

actively developed, by Sven Verdoolaege, until release of 0.27 early September 2024

distributed with a Python interface

written in C, easy to bind deeply

and generation from Haskell to OCaml :-)

I might rewrite the bindings incompatibly from CamelCase to Snake_case

outdated bindings for OCaml, half-auto-generated

three distributions / library options

isl: the core polyhedral manipulation and solution / optimization

barvinok: dynamically-typed domain-specific programming language; tool **iscc**

pet: extracting (parts of) a polyhedral model from C code

could be interacted with via binaries (tools) but currently I prefer to integrate more tightly

integrate auto-tuning

try to figure out integrating inlining and memory mode decisions, so we get rid of code interpretation on the "pure" OCANNL side

2025. 02. 11

1: instructions and producer-consumer dependences

2: computation order; iteration space transforms (e.g. tiling); architecture (GPU, distributed)

inlining

written in C++

4 layer IR: algorithm, computation, data, communication

3: mem layout; buffer alloc and free

handles sparsity

4: sync and comms, within and between architectures

integrated with ISL to for example express ASTs for CUDA; Halide IR is used for CPU ASTs

Tiramisu polyhedral compiler

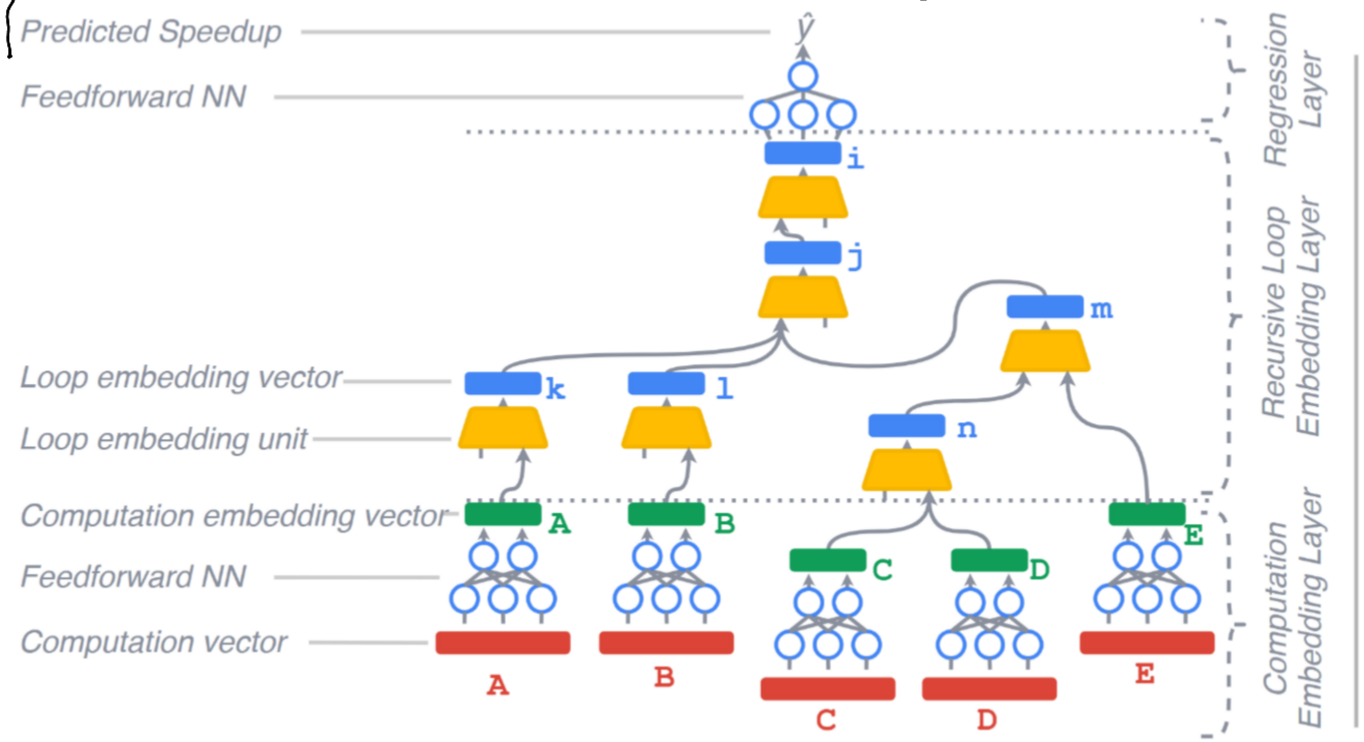
via skewing

handles recurrent NNs and cyclic dataflows

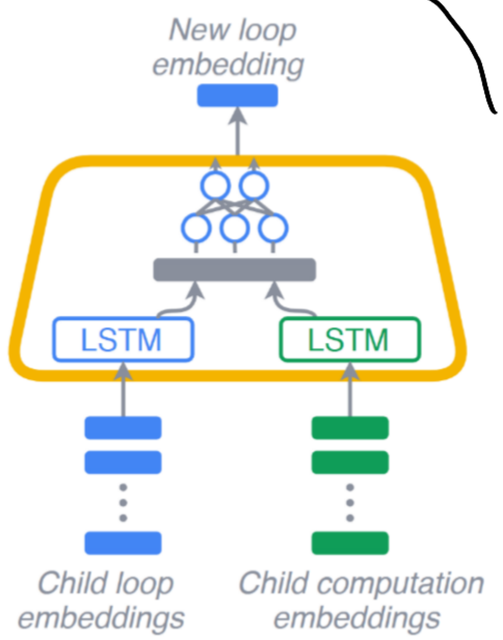
actively developed May 2016-Feb 2023

optimizes distributed code

"A Deep Learning Based Cost Model for Automatic Code Optimization"



(a) Processing the program presented in Figure 1 through the three layers of the cost-model.



(b) Loop embedding unit.

2025.02.12-13

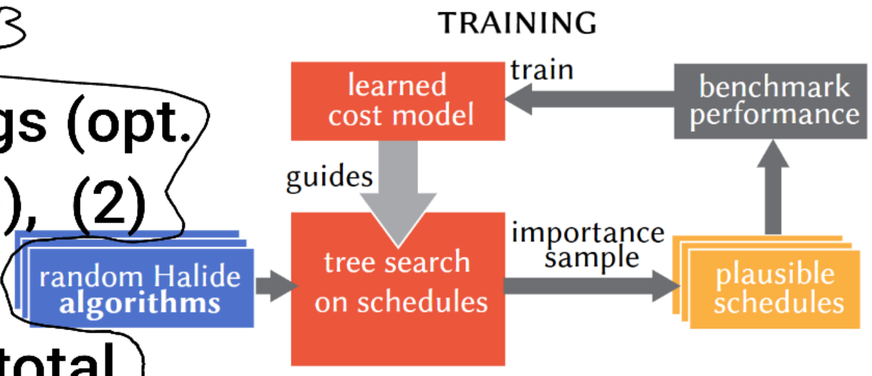
add extra tiling to previous layer if it enables more choices

compute and storage granularity: how many loops to fuse, memory reuse: fusing all loops = inlining

iterates through network layers from output

(1) add inner tilings (opt. vectorized / SIMD), (2) outer tilings (opt. parallel threads); total *2n* steps for *n* layers

shoosing each tile size



Learning to Optimize Halide with Tree Search and Random Programs

beam search: top-k for each step

cost model trained on full pipelines; implicitly assume unscheduled layers are inputs; when interdependent, assume max locality and dense vector loads (optimistic)

autotuning / program search: here Halide (2019), next time Tiramisu (2021)

see figure from yesterday

To Be continued

pruning: limit search space, e.g. loops compute->storage should be single-dimensional (best results for line-buffer)

heuristic instead of search: loop unrolling, no nest reordering except vector dims pushed inside, memory space selection

coarse-to-fine: diversify the beam by prioritizing one representative of a loop nest class; next pass: only optimize within winning loop nest classes, with finer subclasses

cost model and runtime To Be Continued

2025.02.14

algo and schedule features, computed statically:
- histograms of operations to compute a single point
- Jacobians of input/prev-stage acesses wrt. loop dimensions (non-constant ignored assuming worst case)
- count of evaluations of a region, of allocations of storage for their result, of accesses of this storage: both in bytes and in contiguous segments
- number of parallel tasks launched
- number of whole SIMD vectors computed, and of the corresponding scalar values
- number of dense vectors and scalars loaded per vector computed (amortized over sharing by unrolled loops)

region shapes via symbolic interval arithmetic

a small cost network:
features -> coefficients for simple manual heuristics
also built out of the features

beam search with "dropout":
expand k-th best w/prob p^k

Halide program search optimization

pretrained on random algos / networks

autotune: benchmark end-of-beam candidates
finetune: update model while beam-searching

2025.02.16

heuristic / not optimized:
parallelization, vectorization

searches for a sequence of
transformations: fusion,
tiling, interchanges, unrolling

Tiramisu program search

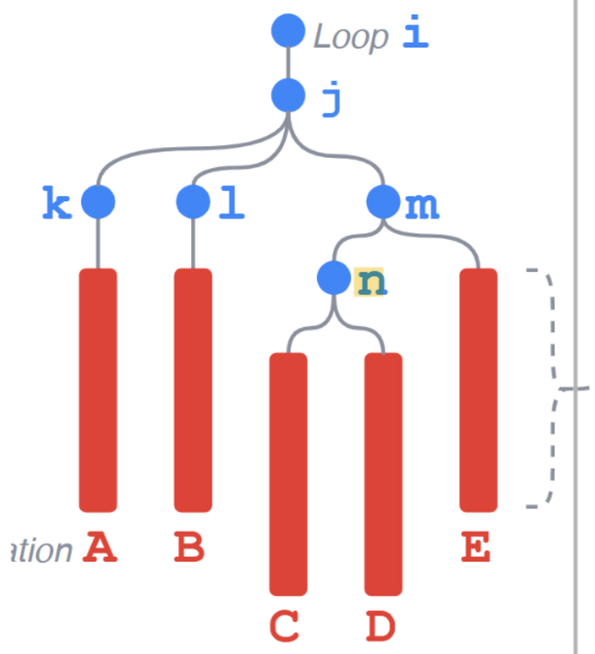
recursive NN over a loop embedding:
combines outputs of LSTMs

loop children:
LSTM over loop
transformations

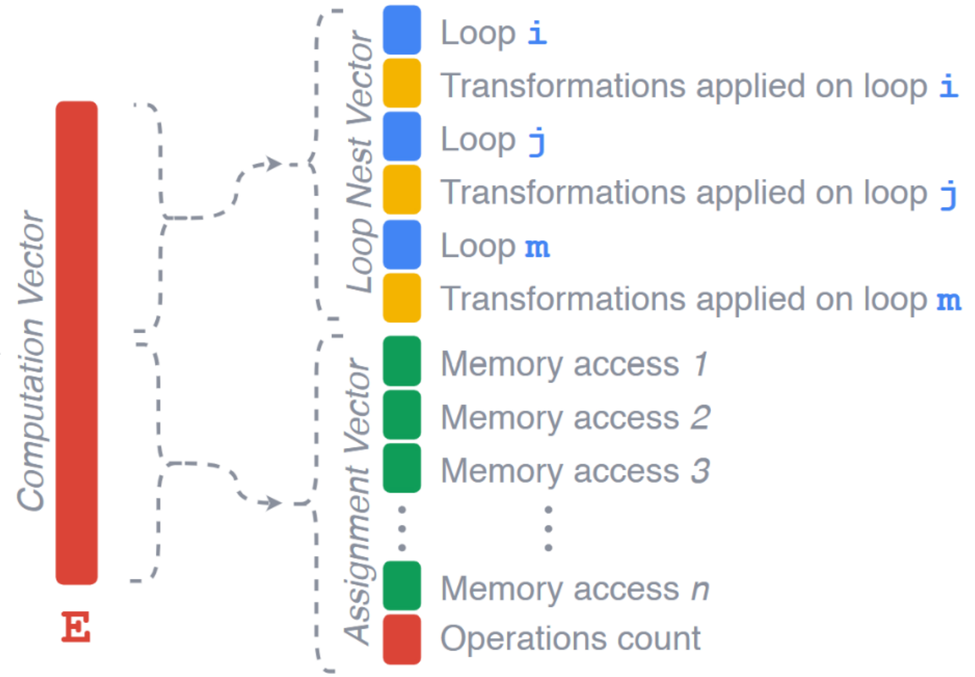
use of two LSTMs loses
ordering info between
nested loops and statements

statement children:
LSTM over computation vectors

computation vectors composed of loop
nestings including loop transformations,
and memory accesses, but padded to 1235
dims for feedforward NN embedding



Program tree representation.



(c) Computation vector.

2025.02.16

sibcehss.com

matrix multiplication on multicore CPU

the machine FMA is accumlating!
c := a*b + c; FMA = fused multiply add = MAC = multiply accumulate

uses vectorized FMA instructions

Intel's MKL has dedicated SGEMM functions / kernels for each microarchitecture

SGEM = single-precision general matrix multiply: a*A*B + b*C

CPUs are compute bound even for small matrices

`-march=native` specializes C compiler's output to microarchitecture it compiled on

big gain from loop reordering: non-contiguous iterator should never be innermost

much better than assigning to a register in innermost loop

tiling the reduce loop (middle) with outermost replication

idea: tile width so that a single tile of the reduce loop fits in cache; in practice the best size is bigger

tiling on multiple dimensions

it's two elementary loop transforms: strip-mine and reorder

multithreading strategy

tile rows of A and columns of B so that there's no need to communicate except to gather the results

2025.02.17

matrix multiplication in CUDA without tensor cores

we implement FMA / MAC:
C := alpha A @ B + beta C

cuBLAS at FP32 doesn't use tensor cores, only at TF32 and BF16

should be compute bound if well designed

kernel 1: each cell of C a separate thread doing reduction, thread ids end up increasing along columns of C; 1.3% eff.

no change in assembly, access coalescing at runtime, requires aligned access; compiler does partial inner loop unrolling

kernel 2: ensure thread ids increase along rows of C; 8.5% eff.

kernel 3: threads first populate shared mem, then sync, then reduce the shared mem block; 12.8% eff.

To Be Continued

kernel 4: kernel 3's thread-parallel A loop strip-mined to new innermost non-parallel loop, smaller tiling of reduce loop than kernel 3 to lower smem; 36.5% eff.

it's a strip-mining of the reduce loop, but tricky as pure loop transforms -- kinda also strip-mining of A and B's non-shared axis (into block-parallel and thread-parallel) and promoting tiles to shared mem

combats shared mem access bottleneck

why does the strip-mining halve threads per threadblock?

kernel 5: also strip-mine kernel 4's thread-parallel B loop to new innermost non-parallel loop; now too few threads wrt. tile size: smem copy loops

68.7% eff.

__syncthreads separates two implicit thread-parallel loops, but logically the thread-parallel iterators are different (the first loop's iter for A becomes the innermost non-parallel loop for A)

siboehm.com

2025.02.18

siboehm.com

kernels 7 and 8: shared-mem bank conflicts

kernel 6: transpose a tile of A while copying to shared mem; vectorize copying via `reinterpret_cast<float4 *>`; 78.4% eff.

smem bank conflicts and double buffering continued after SGEMV notes on AMD GPUs

matrix multiplication in CUDA without tensor cores

kernel 9: autotuning:
how much data cached:
- from global to shared mem
- from shared mem to register file
  i.e. local mem

kernel 10: warptiling: GPU
--GMEM-to-SMEM-> SM
--SMEM-to-RegisterFile-> Warp
--RegisterFile-to-core-> Thread

each level of tiling is: strip-mining, reordering, preparatory loops for mem level transfer (sometimes with transpose)

Levels of parallelism:
- blocktiling: different Streaming Microprocessors, syncing and shared mem within tile
- warptiling: different warp schedulers, mem bank benefits, warp-level ops (eg. tensor cores)
- threadtiling: Instruction Level Parallelism

CUDA compiler can do the ILP vectorization

kernels 11 and 12: double buffering

2025.02.19
seb-v.github.io

matrix multiplication on AMD RDNA3 GPUs

To Be Continued

kernel 1 is identical to CUDA; 3.3% eff.

like before, we do C := a*A*B + b*C in FP32 prec.

32*4 (b/c FP32) = 128 bytes mem transaction

kernel 2: LDS tiling; square As, Bs 32x32 of A / B, strip-mines reduce dim and reorders, loads the tiles As, Bs w/ transpose into LDS along rows of A, B; 13.1% eff.

nomenclature:
Compute Unit = Streaming Multiprocessor
Stream Processor = CUDA Core
wavefront (32 or 64) = warp (32 threads)
workgroup = block
Local Data Storage = Shared MEMory
The programming model is almost the same, uses the same keywords e.g. `__shared__`.

kernel 3: register tiling adds 2 (3) more tiling levels: 1-per-thread increases arithmetic complexity and 2x2-per-wave enables vectorized mem transfers; loads a col,row of LDS tile to registers to reduce, strip-mines both tile dims;

the wave tile is composed of 2x2 subtiles sized 8x4 that each fit in a wavefront (8*4=32) (why?) doesn't prevent mem bank conflicts b/c accessed contiguously?

also unlocks `v_dual_fmac_f32`: parallel vector multiply-accumulate operations

this is kinda swizzled layout

kernel 7 from siboehm.com prevents bank conflicts by dispersing consecutive Bs, so on reading to regs they're from the same bank

I don't see how kernel 8 from siboehm.com prevents bank conflicts?

swizzled layouts often use bitwise ops on indices

2025.02.20 seb-v.github.io

needs __launch_bounds__ , otherwise compiler uses scratch mem instead of registers

while some threads queue for loading from GMEM, others continue computing

kernel 4: double buffering; load the next block's GMEM data into registers at the beginning of a step; sync at end of step and write to SMEM; also adds unroll pragmas; 83.7% eff.

matrix multiplication on AMD GPUs: double buffering and bank conflicts

kernel 11 from siboehm.com: unlike kernel 4 above because loading always from GMEM to SMEM, doubles the size of SMEM buffers and alternates the halves

kernel 5: bank conflicts --> adds padding to As rows (smem tile of A) so that accesses are staggered: original BM%32=0, new (BM+4)%32=4 no same-thread same-bank conflicts;
arithmetic intensity --> 2x wider wave tile;
adds `-cumode` (each SIMD32 sticks to its own LDS); 109.8% eff.

theoretically close to 200% eff. might be possible because for unknown reason rocBLAS is not using dual_fmac: 2x multiply-accumulate ops.

To Be Continued

kernel 12 from siboehm.com: uses cuda::memcpy_async with a cooperative_groups barrier, with double size buffers; compiler can better use available hardware: copying can happen in parallel to all computations

2025.02.21 seb-v.github.io

nomenclature:
HIP = CUDA
ISA = SASS (assembly)
AMDIL / ROCm llvm IR = PTX

matrix multiplication on AMD GPUs: from HIP to ISA

kernel 6: optimize vector arithmetics using v_dual_fmac_f32: maximize number of vec registers read per instruction; maximize use of cache; maintain a consistent symmetric access pattern; don't worry about contiguous use of registers: reorder before writing to LDS

kernel 6 details: ensure C_regs contiguous, assign A_col and B_row to non-overlapping banks (0-1 and 2-3), update LDS loads and inner loop v_dual_fmac accordingly, permute registers as needed by code writing to global memory; 123.7% eff.

kernel 7: loop unrolling; didn't work with the HIP implementation b/c the compiler prefetched more values from the LDS; now just duplicate ISA loop body with address increments, remove branch instructions; 135.1% eff.

kernel 8: batched GMEM loads; precompute global load offsets into scalar registers (saving on vec registers); in the loop, update A,B index vec registers, global_load_b32 output-reg, index-reg, offset-reg format; spread the loads accross the unrolled inner loop; 160.6% eff.

remaining possible optimization: double buffering for LDS

this approach is not scalable but gives insight into optimal execution on RDNA3

2025.02.22

The FineWeb recipe

OpenAI, Anthropic etc. use their own web crawlers and data

Common Crawl is a non-profit org extracting data for researchers; currently over 250 billion pages; new crawls w/ 200-400TiB text every 1-2 months

FineWeb-Edu: 1.3 trillion and 5.4 trillion tokens (very) high quality

filtered by "educational content" classifier distilled from Llama-3-70B-instruct prompted to focus on grade/middle-school

FineWeb is derived from 96 CommonCrawl snapshots; has 15 trillion tokens; 44TB

reduce

CommonCrawl text extraction (WET) is poor quality; but text extraction is costly

data quality: train small models and evaluate on early-signal benchmarks

CommonSense QA, HellaSwag, OpenBook QA, PIQA, SIQA, WinoGrande, ARC, MMLU

base filtering: URL blocklists; English only (fastText classifier); repetition filter (many repeated lines); filters from MassiveText (used to train Gopher); 36T tokens

deduplicating: MinHash based: 5-garms, 112 has functions, 14 buckets; 0.85 similarity -> 98.8% discard

filters from C4 dataset: no "lorem ipsum" and no curly brackets

within-snapshot only; cross-snapshot-repeated data has higher quality, global dedup upsamples bad quality

C4 dataset inspired filters: docs with fraction of lines ending with punctuation > 0.12; docs with fraction of chars in duplicated lines > 0.1; docs with fraction of lines shorter than 30 chars < 0.67

MassiveText filters: docs between 50 and 100K words, mean word length from 3 to 10 chars, hash|ellipsis-to-word ratio < 0.1; less than 90% lines bullet points; less than 30% lines end with ellipsis; 80% words contain alphabetic char; contains at least 2 of: *the, be, to, of, and, that, have, with*