An Introduction to Labs (2)

Aaron Zhao, Imperial College London

Overview

- Lab 3:
  - Mixed Precision Search
- Lab 4:
  - Software stream: Automated and manual low-level optimizations
  - Hardware stream: Emitting SystemVerilog from Pytorch models

Mixed Precision Search

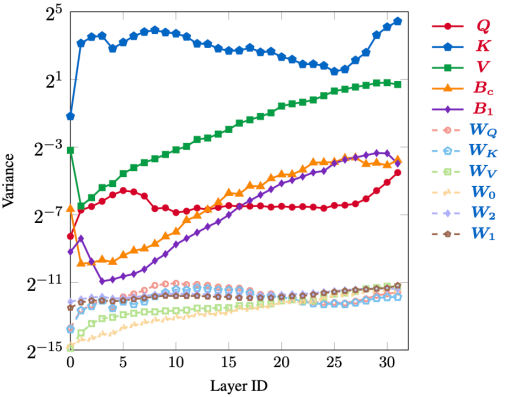
Mixed precision search

- Different blocks of a neural network have different numerical characteristics.
- It naturally makes sense to use different precisions for different blocks.

Algorithm 1 Transformer layer

Require: X ▷ Input features
Require: H ▷ Number of heads

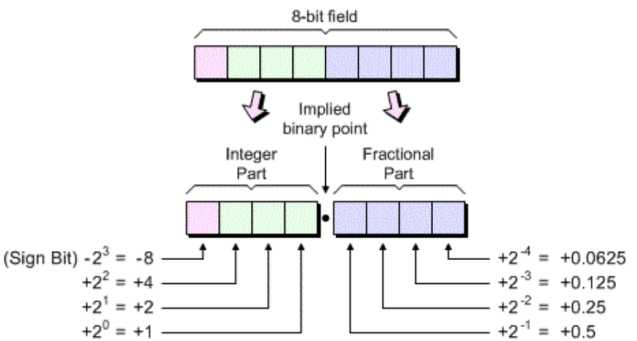
$\boldsymbol{X_n} \leftarrow LayerNorm(X)$
for $i \in [0, H)$ **do**
  $\boldsymbol{Q_i} \leftarrow \boldsymbol{X_n} W_{Q_i}$ ①
  $\boldsymbol{K_i} \leftarrow \boldsymbol{X_n} W_{K_i}$ ②
  $\boldsymbol{V_i} \leftarrow \boldsymbol{X_n} W_{V_i}$ ③
  $A_i \leftarrow \frac{\boldsymbol{Q_i} \boldsymbol{K_i}^T}{\sqrt{d_k}}$ ④
  $\hat{\boldsymbol{A}}_i \leftarrow softmax(A_i, axis \leftarrow -1)$
  $B_i \leftarrow \hat{\boldsymbol{A}}_i \boldsymbol{V_i}$ ⑤
end for
$\boldsymbol{B_c} \leftarrow concat(B_0, \ldots, B_{H-1})$
$B_0 \leftarrow \boldsymbol{B_c} W_0 + b_0$ ⑥
$\boldsymbol{B_n} \leftarrow LayerNorm(B_0 + X)$
$\boldsymbol{B_1} \leftarrow ReLU(\boldsymbol{B_n} W_1 + b_1)$ ⑦
$B_2 \leftarrow \boldsymbol{B_1} W_2 + b_2$ ⑧
$O \leftarrow B_2 + B_0 + X$
return O



Integer quantization

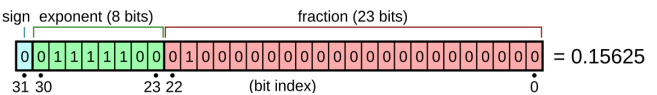
Depending on your background, the definition of integer quantization can be different. We define integer quantization as the process of converting a floating-point number to an integer number with a fixed decimal point (same as fixed-point arithmetic).

These are also known as linear quantization.



Non-linear quantization

- Float based
  - Sign (Blue)
  - Exponent (Green)
  - Mantissa (Red)
  - Exponent bias (a constant)
- Logarithmic based



Software stream: Automated and manual low-level optimizations

torch.compile

`torch.compile` is a tool that can be used to compile PyTorch models to a lower-level representation. It is currently in the Pytorch ecosystem, and it is a tool that can be used to optimize PyTorch models.

One can see this as an automated way to perform low-level optimizations, replying on optimization flows that are already implemented.

FX-Graph: FX-Graph is a symbolic representation of PyTorch models that is designed to be more amenable to optimization

TorchDynamo: TorchDynamo is responsible for JIT compiling arbitrary Python code into FX graphs

TorchInductor: TorchInductor i*TorchInductor: TorchInductor is responsible for lowering FX graphs into a efficient kernel representations

Triton: the language that is used to actually implement these low-level kernel representations.

Fused kernels

In this lab, we take a look at what is a fused kernel that implements ScaledDotProductAttention.

Kernel fusion can happen at different levels, we can fuse kernels at the level of the computation graph, or we can fuse kernels at the level of the actual implementation.

In this example in the lab, we fuse at at the pythonic level, which is a higher level of abstraction.

Porting a custom kernel

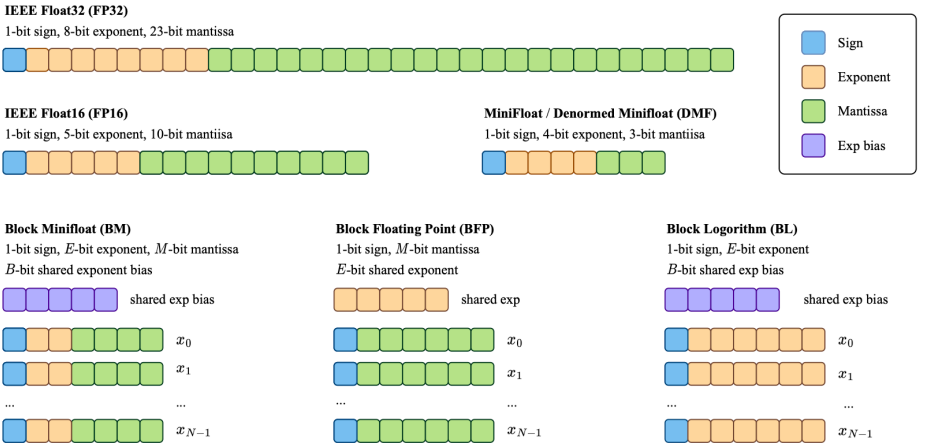
We aim to port an MXInt kernel in this lab.

What would you do?

- It is a custom kernel that is implemented in CUDA (CUTLASS CUTE)
- CUTLASS is a template library that provides highly tuned matrix multiplication kernels for NVIDIA GPUs using CUDA.
- This requires a matching GPU CC (compute capability).
- You will have to build this kernel from scratch

Porting a custom kernel

What is MXInt?



Hardware stream: Emitting SystemVerilog from Pytorch models

Porting a custom kernel (iii)

The goal of this lab is to automatically generate a fully-connected layer in SystemVerilog, and test it using Cocotb.

Cocotb is a COroutine based COsimulation TestBench environment for verifying VHDL and SystemVerilog RTL using Python.

We use the Cocotb with the **Verilator** backend.

Cocotb: direct testing in Python, no need to write a testbench in SystemVerilog.

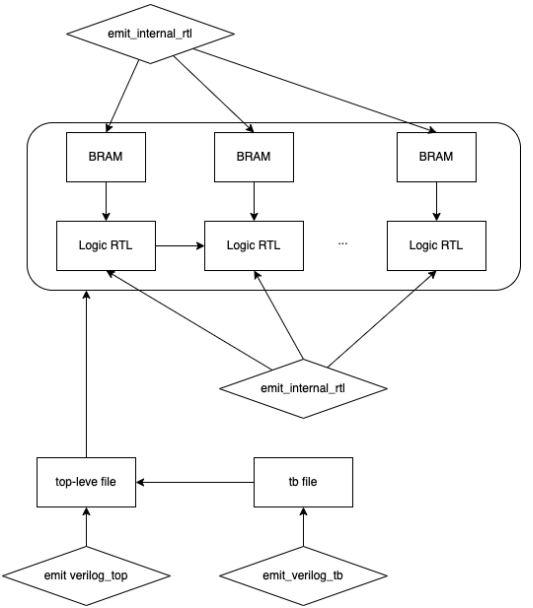
Verilator: 'up-compiles' SystemVerilog into multithreaded C++, lightening fast, no need to open vendor tools when doing behavior level testing.

EmitVerilog Pass in MASE

- Classic source to source generation
- Directly generate SystemVerilog from MaseGraph

```
from chop.passes.graph.transforms import (
emit_verilog_top_transform_pass,
emit_internal_rtl_transform_pass, emit_bram_transform_pass,
emit_verilog_tb_transform_pass)
```

EmitVerilog Pass in MASE (ii)



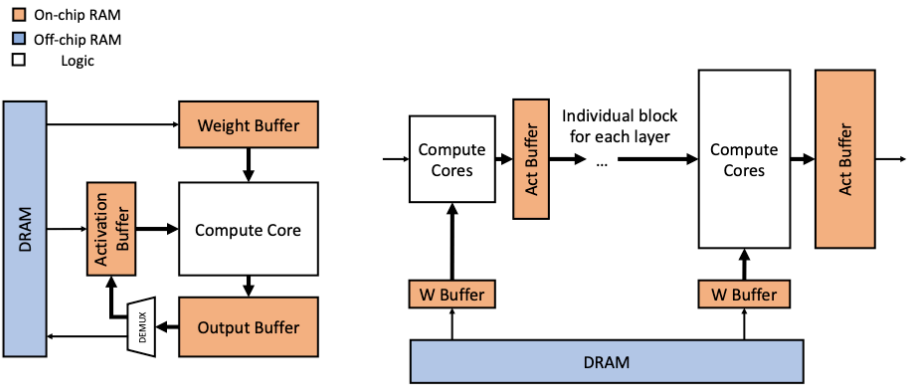
EmitVerilog Pass in MASE (iii)

EmitVerilog generates dataflow designs

- Generate functional elements (RTL)
- Generate memory components (BRAM)
- Dataflow accelerator design without making use of the DRAM

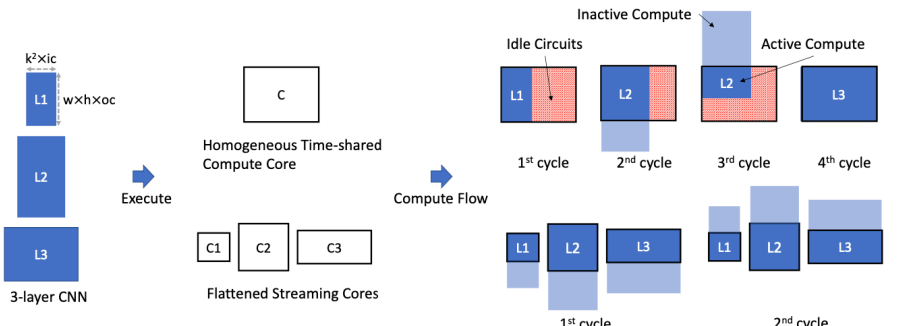
Dataflow accelerator designs

- A homogeneous Big Compute Core (normal design, ASIC)
- A series of tailored small compute cores (dataflow design, FPGA)



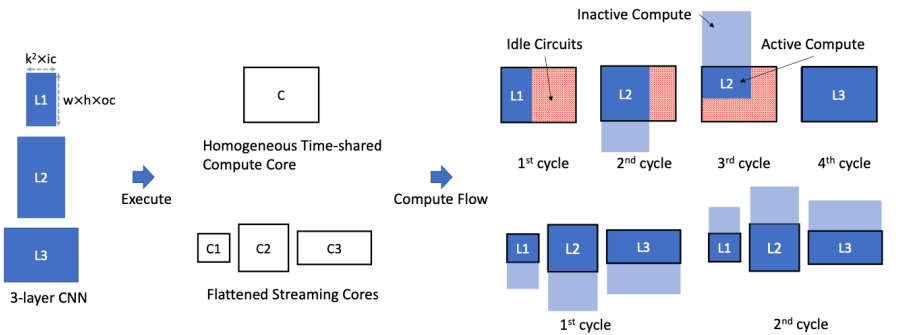
Dataflow accelerator design: Advantages

- No complex control flow (minimal or no ISA design)
- (Almost) no waste of resources
- (Almost) fixed memory access pattern
- Deeply pipelined



Dataflow accelerator design: Disadvantages

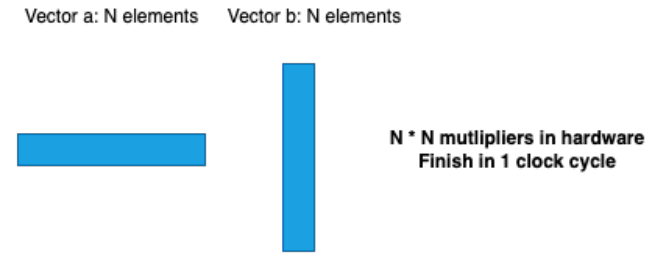
- Re-program hardware for each new network
- Scalability issues
- If DRAM is utilized, hard to achieve great performance by filling up all pipeline stages

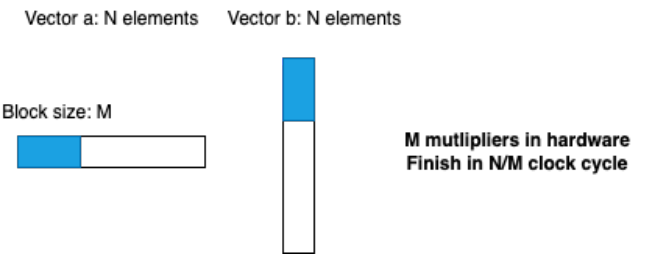


The compute pattern Simple blocking

```
# Breaking the vector into blocks
for i in range(0, n, block_size):
  # calculate end val considering the last block which can be
smaller than block_size
  end_val_i = min(i + block_size, n)
  # Retrieving block of a
  sub_a = a[i:end_val_i]
  # Retrieving corresponding elements from the vector
  sub_b = b[i:end_val_i]
  # multiplication, actual hardware dimension is (block_size, 1)
  result += np.dot(sub\_a, sub\_b)
```

The compute pattern





The compute pattern (ii)

- $N >> M$, this gives you a chance to do a trade-off between resources and latency simply by changing M.
- Blocking can happen in a 2D shape!

2D Blocking

M * M mutlipliers in hardware
Finish in N*N/(M*M) clock cycle

- Parallel Multipliers (M^2).
- M Adder Trees ($log_2(M)$).
- M Accumulators (M).