

CSC290/420 ML Systems for Efficient AI Work and Parallelism in ML Programs

Sreepathi Pai

September 2, 2026

URCS

Outline

Work in ML Programs

The von Neumann CPU

Modern Times: Pipelined Execution

Slipping the Surly Bonds of In-order Execution

The Search for Independent Instructions

Alternative Designs

Work in ML Programs

The von Neumann CPU

Modern Times: Pipelined Execution

Slipping the Surly Bonds of In-order Execution

The Search for Independent Instructions

Alternative Designs

- To figure out Time or Energy, we need to identify units of work performed by the ML program
- Work exists at multiple levels of abstraction
 - Math formulae
 - Kernels or “subroutines” (e.g., matrix multiplication)
 - Instructions executed by a processor
 - Actions performed for each instruction inside the processor

A ML Program in PyTorch

```
device = torch.accelerator.current_accelerator().type if torch.accelerator.is_available() else 'cpu'
print(f"Using {device} device")
```

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )
```

```
    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

```
model = NeuralNetwork().to(device)
print(model)
```

Excerpt from the PyTorch tutorial.

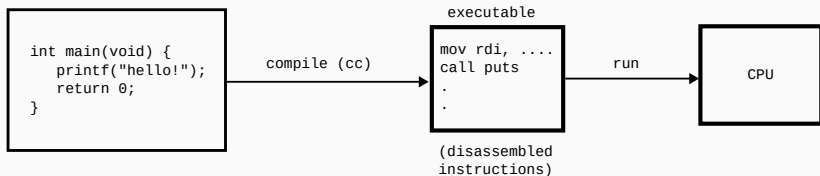
ML Program Execution modalities

- Training mode
 - Forward mode and backward mode
 - The “backward mode” program is automatically derived from a ML program
- Inference mode
 - Forward mode only

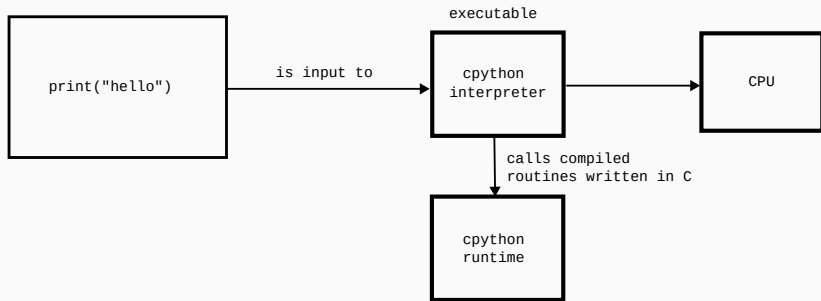
ML Program Execution

- Most ML programs run on CPUs or GPUs
 - Central Processing Unit
 - Graphics Processing Unit
- Neither of these understands “Python”
 - Python programs must be translated to the “native” language of these machines
 - Assembly language, Native code, Machine code
 - There may be multiple layers of translation: Python to CUDA to GPU machine code

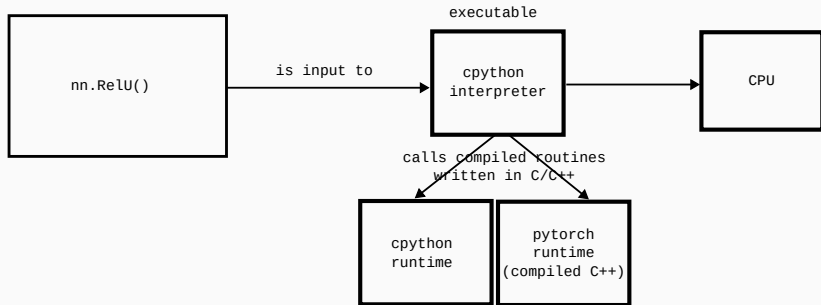
Conventional Program Execution



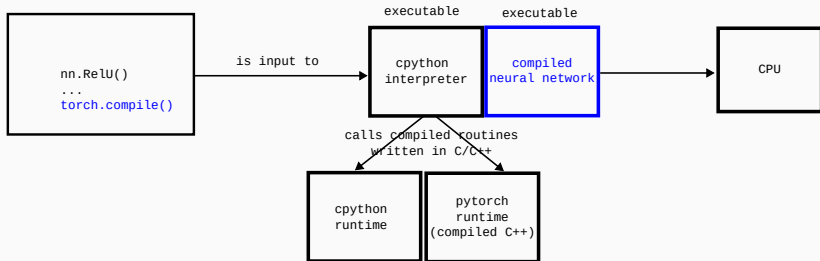
Conventional Python Program Execution



PyTorch 1.0 era



PyTorch 2.0 era



Outline

Work in ML Programs

The von Neumann CPU

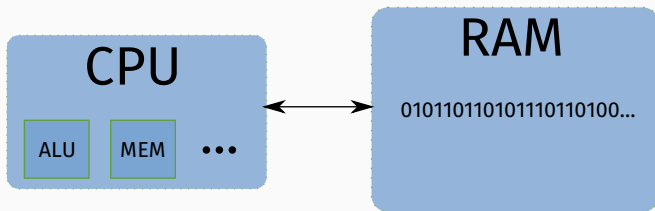
Modern Times: Pipelined Execution

Slipping the Surly Bonds of In-order Execution

The Search for Independent Instructions

Alternative Designs

A von Neumann CPU



- CPU, central processing unit
- Memory, usually RAM, random access memory
 - can also be ROM, read only memory
- Programs, stored initially in memory

John Von Neumann, The First Draft Report on the EDVAC, available at
<https://history-computer.com/Library/edvac.pdf>

CPU Operation

- Instructions are stored in memory
- The CPU FETCHes them from memory
- It then DECODEs the instruction
- The CPU then EXECUTEs it in a functional unit
- Results are WRITTEN BACK (i.e. roughly, made visible)
- Each of these steps is co-ordinated by a clock
 - and occurs over one or more “cycles”
 - Clock, global coordinating mechanism

Instruction Basics

- An instruction is a simple command to a processor

`ADD dst, src1, src2`

- Here ADD is the operation code, or mnemonic
 - Instructs the processor which operation to perform
- `dst` stores the result of the ADD
- `src1`, `src2` are the source (input) operands to ADD
- On most CPUs, `dst`, `src1`, `src2` can be:
 - registers (most common)
 - immediate values (i.e., constants) e.g. `ADD dst, src1, 3`
 - memory addresses

Registers

- Fastest form of memory inside a processor
 - Sequential logic, if you've taken digital logic design
- Usually named, unlike RAM which has addresses
 - e.g., EAX, RBX, R10, etc.
- Few in number
 - Most modern CPUs have tens of registers
- Each register can store 32 to 64 bits of data
- On certain CPUs, operands for instructions must reside in registers
 - only memory instructions can access RAM
- Special registers: program counter (PC)
 - contains address of instruction being executed

Types of Instructions

- Integer Arithmetic instructions
 - IADD, ISUB, IMUL, ...
- Floating-point Arithmetic instructions
 - “floating point” is an approximation of real numbers (future lecture)
 - FMUL, FADD, ...
- Comparison instructions
 - GT, LT, GTE, ...
- Logical and bitwise instructions
 - AND, OR, ...
- Memory instructions
 - LD, ST
- Control flow instructions
 - JMP, JC (conditional)
- Lots of other categories
 - System-specific instructions, etc.

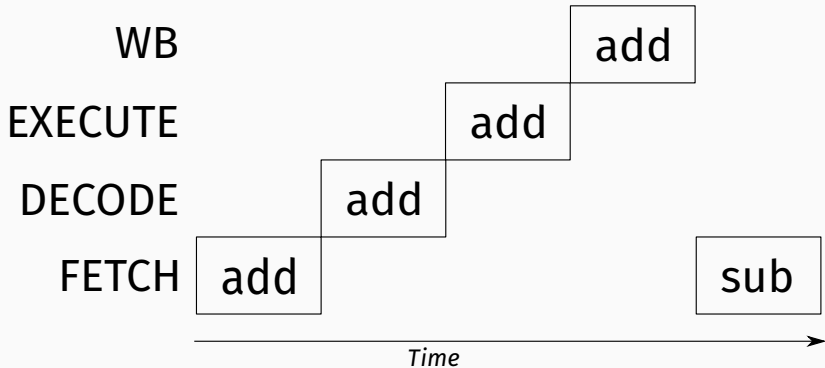
Instruction Set Architecture

- The programmer's interface to processor is known as the "instruction set architecture"
 - e.g., x86, x86-64, ARMv7, RISC-V
- Obsolete categorization: CISC vs RISC
 - Complex Instruction Set Computer (e.g., x86)
 - Reduced Instruction Set Computer (e.g., MIPS)
- The same ISAs can be implemented by different "micro-architectures"
 - Microarchitecture is the internal design of a processor
 - e.g., x86 has been implemented by AMD, Intel, and some other companies

Functional Units

- ALU: Arithmetic and Logic Unit
- MEM: Memory unit
- Other units exist
 - “MMA”: matrix multiply accelerator
 - Tensor “core”

Serial Execution



Outline

Work in ML Programs

The von Neumann CPU

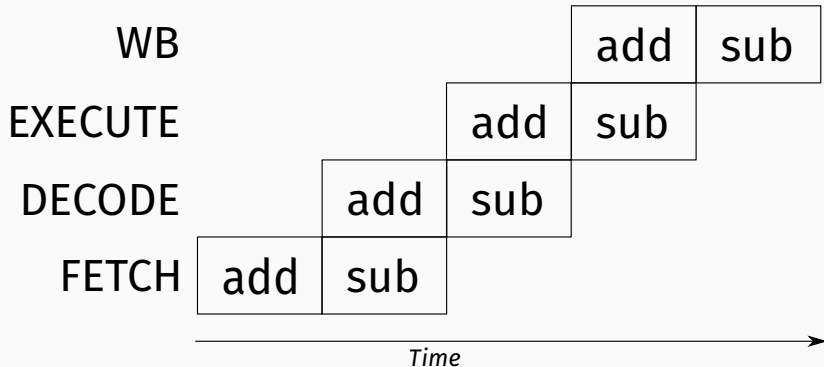
Modern Times: Pipelined Execution

Slipping the Surly Bonds of In-order Execution

The Search for Independent Instructions

Alternative Designs

Illustrating Pipelined Execution



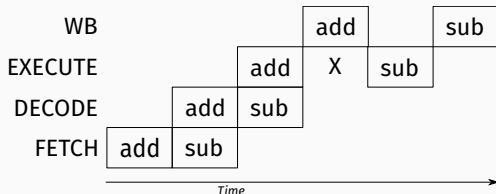
Pipelined Execution

- Pipelined execution splits work into stages
- Here work is “fetch, decode, execute, and write back results for an instruction”
 - But could also be “build a car”, as in assembly line production
 - Or steps of an repeatedly applied algorithm

Dependences and Stalls

```
ADD R3, R1, R2
SUB R4, R3, 1
```

- Here, the SUB instruction depends on the results of ADD instruction
- SUB will read R3 in EX while ADD will write R3 in WB
- This is a *data hazard*, when two instructions attempt to read/write the same register
- The solution is to *stall* the execution of SUB until ADD completes



Handling Different Latencies

DIV R5, R1, R2

DIV R6, R3, R4

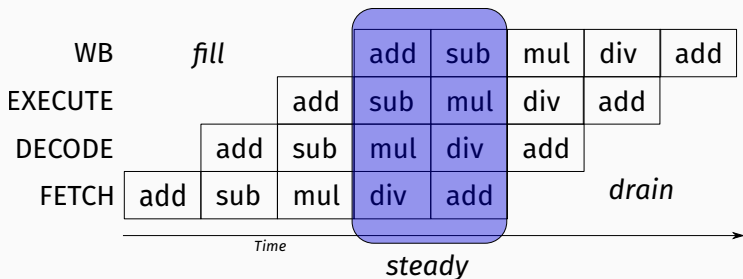
- Each division will take multiple cycles
 - nature of the division algorithm
- Second DIV cannot start until first DIV finishes
 - known as a *structural hazard*
- Different microarchitectural choices:
 - Stall till first DIV finishes
 - Provide multiple ALUs
 - Pipeline the ALU so it can accept an instruction every cycle

Branches

```
addr0: CMP R1, R2, R3
        JL addr1
        SUB R3, R3, 1
        JMP addr0
addr1: MUL R4, R2, R3
```

- After JL has been fetched and decoded, which instruction should be fetched next?
 - SUB or MUL?
- Can't know that until CMP finishes executing
- Known as a *control hazard*
- Stall FETCH until CMP finishes and JL executes

Keeping the Pipeline Full



- Note classic pipelines have five stages
 - Additional MEM stage after EX performs memory operations
- Steady state throughput is 1 instruction per cycle (IPC)
- What can lower this IPC?

Outline

Work in ML Programs

The von Neumann CPU

Modern Times: Pipelined Execution

Slipping the Surly Bonds of In-order Execution

The Search for Independent Instructions

Alternative Designs

Superscalar Execution

- Superscalar execution executes *multiple* instructions at the same time
- Increase resources so as to:
 - Fetch multiple instructions
 - Decode multiple instructions
 - Execute multiple instructions
 - Write back multiple results
- Needs additional hardware resources
 - Moore's Law bounty

Independent Non-Branch Instructions

- Fetching and Decoding multiple instructions requires additional hardware
 - variable length instructions can complicate multiple fetch
 - also can't fetch across conditional branches
- Executing multiple instructions
 - data dependences must be respected
- Writing back multiple results
 - creates new forms of dependences!

Anti-dependence

```
ADD R1, R2, R3  
MUL R2, R4, R5
```

- ADD and MUL are independent instructions
- But, MUL writes to R2 and ADD reads from R2
 - not a problem if executed in order, but simultaneously?

Output dependence

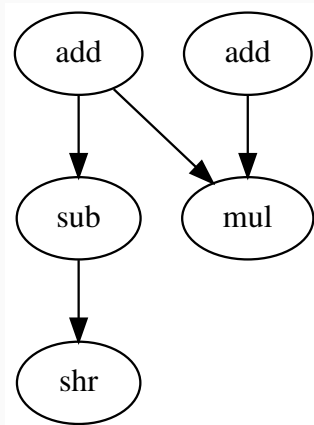
```
ADD R1, R2, R3  
SUB R1, R4, R5
```

- Observe that ADD and SUB are independent instructions
- But, both write to the same output register!

Register Renaming

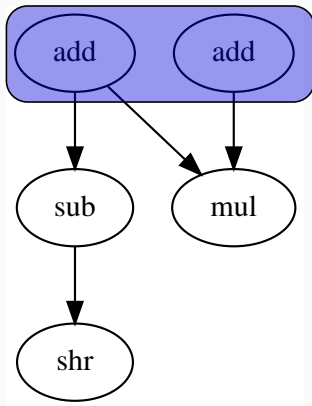
- For both anti-dependence (write-after-read) and output dependence
- ISA registers are *logical register*, that are mapped to *physical* registers on-the-fly by the processor

Instruction Windows + Dataflow Execution = Out-of-order Execution



```
add r3, r1, r2
sub r5, r3, 1
add r4, r1, 3
mul r6, r3, r4
shr r7, r5, 2
```

Instruction Windows + Dataflow Execution = Out-of-order Execution



add r3, r1, r2

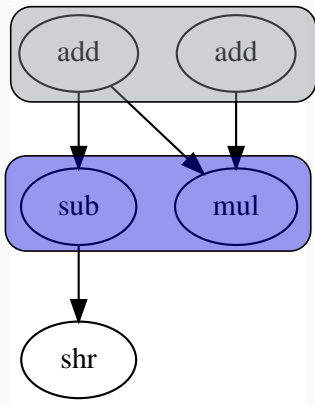
sub r5, r3, 1

add r4, r1, 3

mul r6, r3, r4

shr r7, r5, 2

Instruction Windows + Dataflow Execution = Out-of-order Execution



add r3, r1, r2

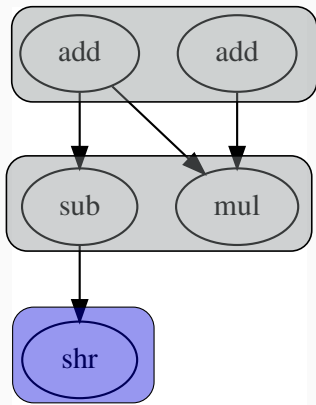
sub r5, r3, 1

add r4, r1, 3

mul r6, r3, r4

shr r7, r5, 2

Instruction Windows + Dataflow Execution = Out-of-order Execution



add r3, r1, r2

sub r5, r3, 1

add r4, r1, 3

mul r6, r3, r4

shr r7, r5, 2

Speculative Execution

- Instruction windows are limited by conditional branches
- Can't fetch and execute until a conditional branch is resolved
- But:
 - can predict which way a branch will go
 - begin executing *speculatively*
 - check speculation when branch ultimately resolves
 - if prediction correct, profit!
 - otherwise, throw away all speculated instructions
 - flush pipeline and refetch from correct location

How far can this go?

- Most modern processors fetch 8 to 16 instructions at a time
- Hundreds of instructions in flight every cycle in steady state
- Why not build ever larger instruction windows?

Outline

Work in ML Programs

The von Neumann CPU

Modern Times: Pipelined Execution

Slipping the Surly Bonds of In-order Execution

The Search for Independent Instructions

Alternative Designs

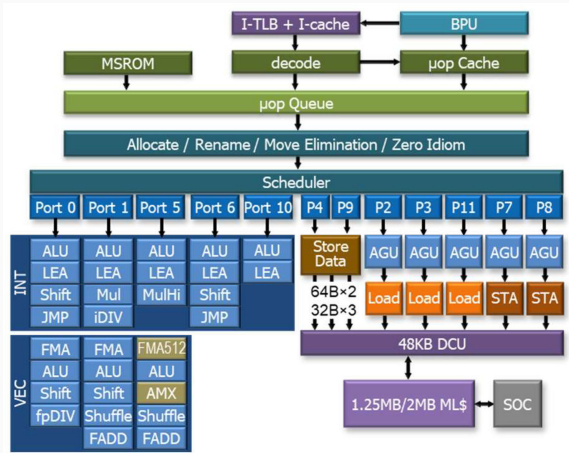
Simultaneous Multithreading

- Instructions from two running programs are inherently independent
 - Browser and Text editor for example
- Different *threads* of execution
- Change FETCH to fetch instructions from different threads at the same time
- Operating system (OS) identifies these threads to processor
- Key idea: Separate front ends for each thread, but common, shared backend
- Known as Simultaneous Multi-threading (SMT), or by its trade name Hyperthreading
- Not always beneficial

Chip Multiprocessors (CMPs)

- Now mostly known as *multicores*
- Replicate processor core across a chip (i.e. multiple processor cores)
- All processor cores share the same memory
- But can run independent programs on each core
- Alternatively, programmers can rewrite their programs to expose multiple “threads” of execution
 - using OS and language facilities

Intel Alder Lake P-core



Rotem, Efraim, Adi Yoaz, Lihu Rappoport, et al. "Intel Alder Lake CPU Architectures." IEEE Micro 42, no. 3 (2022): 13–19. <https://doi.org/10.1109/MM.2022.3164338>

Upper limits on the performance of programs

- If the processor has a max IPC of $IPC_{\max}$
 - across all cores
- And your program has W instructions to execute
- Then, the least amount of time (in cycles) is $W/IPC_{\max}$
 - Multiply this by the frequency of the processor (in Hz) to get seconds
- Alternatively, measure the achieved IPC of your program and compare it to $IPC_{\max}$
 - Subtlety: measure *useful* IPC

Keeping the Beast Fed

To fully utilize a modern CPU, therefore:

- Have enough independent work (i.e. instructions)
- Must keep pipeline full (with or without SMT)
 - avoid stalls
 - avoid mis-speculation (wasted work)
- All cores occupied with threads

Outline

Work in ML Programs

The von Neumann CPU

Modern Times: Pipelined Execution

Slipping the Surly Bonds of In-order Execution

The Search for Independent Instructions

Alternative Designs

Flynn's Taxonomy

Flynn proposed in 1966, a taxonomy of processor architectures

- A simple in-order pipeline is Single Instruction Single Data
 - SISD
- A multicore is Multiple Instruction Multiple Data
 - MIMD
- Multiple Instruction Single Data
 - ?
- Single Instruction Multiple Data
 - vector machines
 - short vector instructions
 - GPUs