Cloud Service Models

Paas: Platform as a Service
- provide SW platform
- eg Googles App-Engine
- avoid worrying about platf. scalability

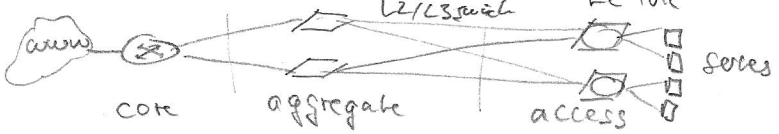
Saas: Software as a service
- Provide licensed apps
- no cost of maintenance, installation

Iaas: Infrastructure
- raw compute, storage, network → AWS

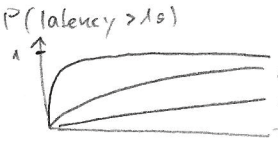
Data centers

Specialized ML cloud HW (Google TPU)

Topology:



Latency Oup depends on all servers finishing. A few slow servers large scale → more prone to high tail latency



$P_{ios,fast} = (1 - P_{slow})^N$ P_{slow}: prob. of slow srvr

TCO

Total cost of Ownership = CapEx + OpEx
- CapEx: HW
- OpEx: electricity

Reliability
- FIT failure in time, failures per billion h of op
- MTTF mean time to fail, time to first error
- MTTR mean time to repair, detect & fix fault

$$\text{Steady state availability} = \frac{MTTF}{MTTF + MTTR}$$

Performance

2010 → 2015: storage & network ×10, CPU not: why?

$P_{dyn} \propto C \cdot V^2 \cdot f$

Why multicore?

[Replicate smaller cores with only little less performance increases total performance and keeps power down.]

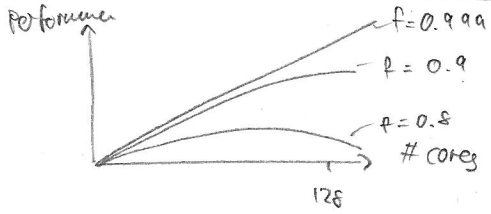
	1 core scaling	2 core	4 core
Core A	A	~ A/2	~ A/4
Core P	W	~ W/2	~ W/4
Chip P	W + 0	W + 0'	W + 0''
Core Perform.	P	0.9 · P	0.8 P
Chip Perform.	P	1.8 · P	3.2 P

Amdahl's law

$$\text{speedup} = \frac{1}{(1-f) + \frac{f}{n}} \xrightarrow{n \to \infty} \frac{1}{1-f}$$

f: fraction running in parallel
1-f: non-parallelizable fractions

- ignores (power) cost of n cores
- more cores → each is slower
- parallel speedup < n



Figures

- response time
- throughput

elapsed time
- total, incl. I/O, OS, idle

CPU time
- spent processing

$$\text{CPU time} = \frac{\text{CPU clk cycles}}{\text{clock rate}}$$

cycles per instruction
- determined by program, ISA, compiler

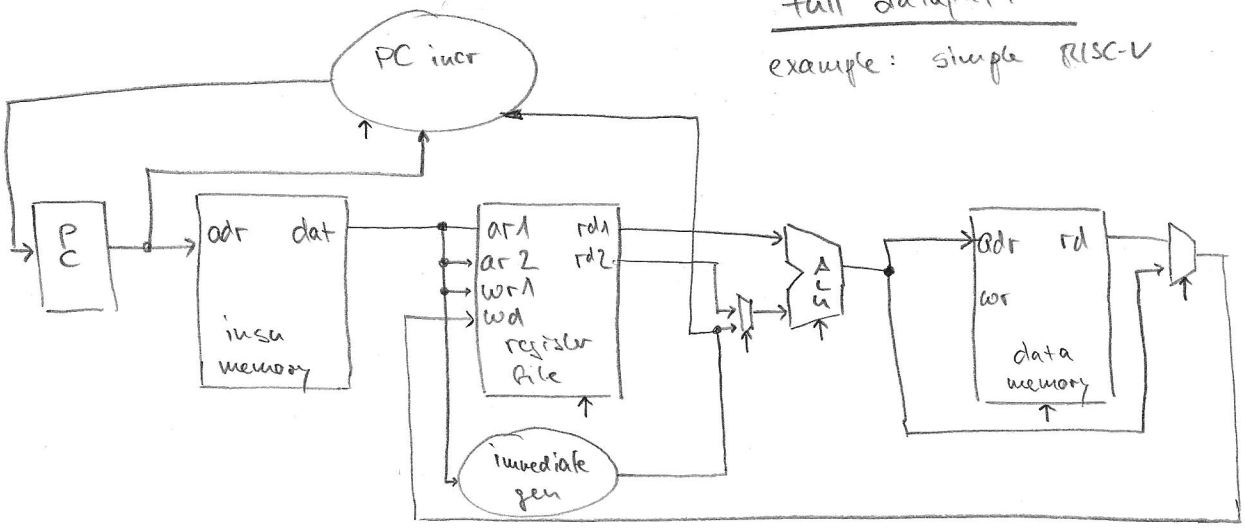
avg. cycles per instruction
- CPU hardware

$$CPI = \frac{\text{CPU time} \times f_{clk}}{\text{Instr. count}}$$

Performance depends on
- ✓ Algorithm
- ✓ Programming
- ✓ Compiler
- ✓ ISA

Full datapath

example: simple RISC-V



PC	Program counter	instr. memory → fetch insn
RF	register file	contains temporary data
ALU	arithmetic logical unit	calculates stuff used for - result for data - mem adr for ld/st - branch compare
↑	control signals	

von Neumann architecture data + instruction memory share same bus, thus can't fetch & run at same time → von Neumann bottleneck

pipeline

parallel improves performance
- run different tasks at same time

example
- IF: insn fetch
- ID: insn decode and RF read
- EX: exec op or calc address
- MEM: access memory
- WB: write back to RF

speedup if balanced all stages take same time

speedup = ×Nstages

non-ideal due to wbal & hazards

```
ld x1,100(x4)   | IF | ID | EX | · | WB |
ld x2,200(x4)   |    | IF | ID | EX | · | WB |
ld x3,400(x4)   |    |    | IF | ID |  | · | WB |
```

RISC-V designed for pipelining
- All insn 32-bit
  - → easy IF/ID in 1 cyc
  - → x86 has 1–17 byte insn
- Few & regular insn fmt
- ld/sd addressing

HAZARDS

situation that prevents starting next insn in cycle.

structural
- conflicting use of resource
- example simultaneous ld/st
- → causes "bubble" - pipelined arch need separate ins/dat mem or separate caches

data insn depends on completion of data access of prev. insn

```
add  x1a,x0,x11
sub  x2, x1a,x3
```

```
IF ID EX MEM WB
   IF ID EX  MEM WB
stall 2 cyc
      IF ID EX  MEM WB
```

solve by
- **forwarding (bypass)**
  - requires extra connection in datapath
  - doesn't always work: ld x1,0(x2) → sub x4,x1,x5
- **code scheduling**
  - compiler re-arranges insn to mitigate data haz.

control fetch depends on last insn (branch)

solve by
- **stall on branch** wait for b insn complete before loading next insn
- **branch predict** pred. outcome, only stall if wrong
  - static: based on typical branch behav.
  - dynamic: HW measures actual branch behav.
    - → used in deeper pipelines
    - 1/2 bit predictor. only change on 2 mispred.
    - → cache branch targets

exceptions & interrupts unexpected events
- save PC & prog state, jump to handler

L1: Memory Architecture

SOC

Memory Hierarchy

Levels
- Block: unit of copy'z may be multiple words
- Hit: if satisfied by upper level
- Miss: if data is absent

Technology
- Static RAM
- Dynamic RAM
- Ideal: access time of SRAM, capacity/cost of disk
- 2.5D: DRAM on same chip
- HBM: on top of die

Principle of Locality
- need only small portion of mem at a time
- recent items likely to be re-used (loops) → temporal
- items near recently accessed are likely to be accessed (arrays) → spatial

Memory hierarchy
- all on disk
- recently/nearby on RAM
- most recent to cache

Performance Factors
- Row buffer: several rd/refresh in same cycle
- Synch DRAM: seq. read in bursts
- DRAM banking: simult. access to multiple DRAM

Increasing Bandwidth
- 4-word wide
  - miss penalty: 1+15+1
  - BW: 16 bytes / 17 cycl: 0.94
- 4-bank interleaved
  - Miss: 1+15+4×1 = 20
  - BW: 16 B / 20 = 0.8

Cache

replacement policy
- direct: no choice
- prefer non-valid
- LRU: least recently used
- random

miss penalty
on miss: stall pipeline, fetch from next level, restart instr fetch/complete data access

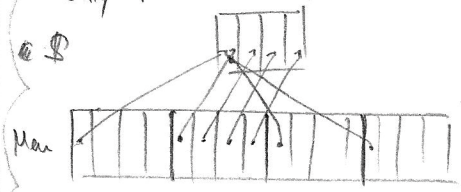
write-through: on write hit also update Mem
- slow: base CPI=1, 10% are sd, mem access 100 clk
  - → effective CPI = 1 + 0.1 · 100 = 11
- sol: write buffer: holds data to be stored, CPU continues normally

write-back: dirty bit, only write to mem if block is replaced

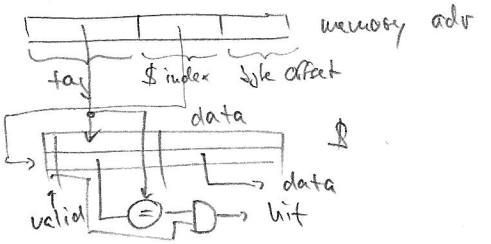
performance
Mem stall cycles = $\frac{\text{mem acc}}{\text{program}}$ · miss rate · miss penalty

avg mem access time = hit time + miss rate · miss penalty

direct mapped
- location determined by address
- only 1 choice: (Blk adr) % (# blocks in cache)



- tags & valid bits: valid if present
- tag: if correct loc

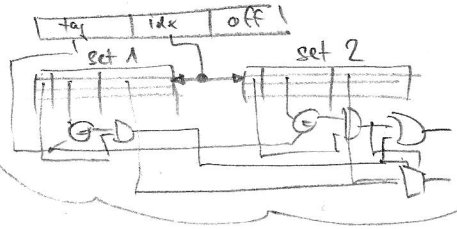


Block size
- larger → reduce miss rate due to spatial locality
- larger → more competition → larger miss penalty

associative cache
Fully Ass: allow mem block to go in any cache entry

N-way: Each set contains n entries, blk no determines set, search all entries in given set, n comparators

2-way associative

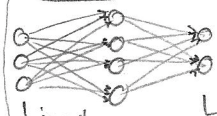


multi-level
L1, L2, L3

$CPI = base + L1_{miss} + L2_{miss}$

L2: ML workloads

Terminology



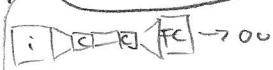
○ : neurons
→ : synapses/activations

L input
L output layer
L hidden

Training

Supervised: i/o is labeled
Unsup.: data is not labeled
reinforce: reward/punishment

Structure



conv: activation (x*y + b)
FC: fully connected matmul
norm: normalize mean/var
pool: reduce dim with max/avg

CNN

ML Workload

Computation

fully connected: "simple" matmul. GEMM, general matmul

conv rearrange to GEMM

e.g. Toeplitz:

GEMM

mtx is toeplitz

transforms

Strassen GEMM $O(N^3) \rightarrow O(N^{2.807})$
by reducing of mult. reduced accum. stability, significantly more mem!

Winograd targets conv. works on small region of inp → uses inputs repeated

FFT and mult. instead of conv. only useful with large matrices

pruning after train remove near-zero multiplications

data

bottleneck data transfer is the limiting factor. data reuse is key!

conv. reuse conv layer only

reuse { - activations - filter weights

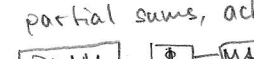
Fmap feature map. conv & FC layer

reuse: activations

Filter conv & FC with batch > 1

reuse: filter wts

cache extra layer of mem to reduce DRAM access. partial sums, activations, ...



HW centric view

ISA extensions

HW loops = zero overhead loop remove branch overhead in for loops. 4 insn: start, end, count, short-cut. +5% area, almost 2x speed

Bit manipulation useful for binarized neural nets. popcount, rotate, find first/last, ...

Auto incr. ld/st incr. address afw accessing array a[i++]
- w/o: ld x2, 0(x10) / addi x10,x10,1
- w/: ld x2, 0(x10!)

SIMD one cycle dot-prod e.g. clock gating to save energy of unused parts

shuffle key for SIMD memory efficiency. rearrange values for next sliding window

Example

```
for (i=0; i<100; i++)
    d[i] = a[i] + s[i];
```

11 → 1.25 cyc/out

Baseline

```
mv   x5,0
mv   x4,100
Lstart:
ld   x2, 0(x10)
ld   x3, 0(x11)
addi x10,x10, 1
addi x11,x11,1
add  x2,x3,x2
ss   x2,0(x12)
addi x4,x4,-1
addi x12,x12,1
bne  x4,x5, Lstart
```

Auto incr ld/st

```
mv   x5,0
mv   x4,100
Lstart:
ld   x2,0(x10!)
ld   x3,0(x11!)
addi x4,x4,-1
add  x2,x3,x2
ss   x2,0(x12!)
bne  x4,x5, Lstart
```

HW loop

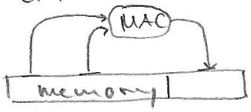
```
lp.setupi 100,Lend
ld   x2, 0(x10!)
ld   x3, 0(x11!)
add  x2,x3,x2
Lend: ss x2, 0(x12!)
```

SIMD

```
lp.setupi 25,Lend
lw   x2, 0(x10!)
lw   x3,0 (x11!)
pv.add.b x2,x3,x2
Lend: ss x2,0 (x12!)
```

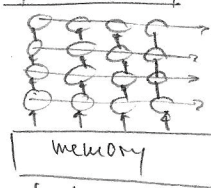
Ser, Par, Systolic

serial high memory access



parallel GPU more serial cores working in local memory

systolic array e.g. google TPU



can handle GEMM very well, less memory usage

must be large enough or else behaves like serial

in-memory

use charge of mos-cap. input from word-line

$$\Delta V = \frac{1}{C_{BL}} \times \int_0^{T_{MAC}} I(t)\,dt$$

bit wise MACC

Out of Order Processors

Terms

Flynn Bottleneck: Scalar pipeline is by design upper bound to IPC≤1

ILP Insn level parallelism

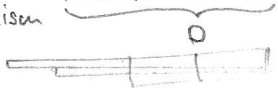
IPC Insn per Cycle

Superscalar:

IP = D·N
OL = 1
Peak IPC = N

D pipelie depth

IP Insn Parallelism



IP = D
OL = 1
Peak IPC = 1

Problems with Superscalar

Forwarding is no longer effective

Pipeline hazards: Resources are stuck

	1	2	3	4	5	6	7	8	9	10	11	12
addf f0,f1,f2	F	D	E+	E+	E+	W						
mulf f2,f3,f2		F	D	d*	d*	E+	E+	E+	E+	E+	W	
subf f0,f1,f4			F	p*	p*	D	E+	E+	E+	W		

E+ is busy so subf must stall.

Solution: ILP

OoO exposes more ILP reordering these insn results in ILP=2. → register renaming required for resolving dependencies.

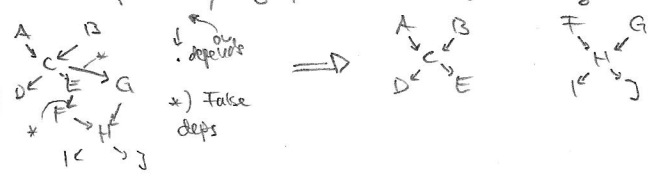
ILP=1
- r1 ← r2+1
- r3 ← r1/17
- r4 ← r0-r3

ILP=1
- r11 ← r12+1
- r13 ← r12+1
- r14 ← r13-r20

ILP=2

Register Renaming

- Dependency exists indep. of HW. It's only a **hazard** if HW has to deal with it
- True data dependency: RAW (read after write) — r1 = r2 + r3; r4 = r1 + r5 } prevent reordering
- false dep: WAW r1 = r2+r3, r1 = r4+r5; WAR r2 = r1+r3, r1 = r4+r5 } WAW, WAR can be fixed by renaming
- Data dependency graph



Reg names are arbitrary and only need to be consistent between writes

r1 = ...
... = r1
... = ...r1
r1 = ...

Ⓐ If reg is W, assign it to physical reg

Ⓑ - until it is written again, translate it

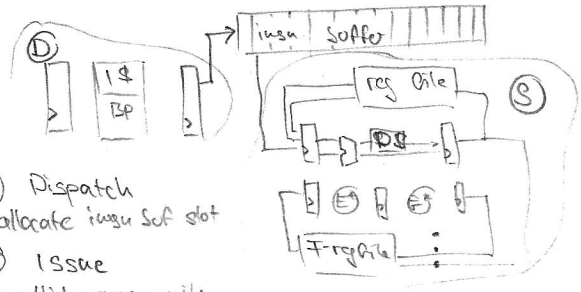
add r2, r3, r1	add p2, p3, p4 Ⓐ
sub r2, r1, r3	sub p2, p4, p5 Ⓑ Ⓐ
mul r2, r3, r3	mul p2, p5, p6 Ⓐ
div r1, 4, r1	div p4, 4, p7 Ⓐ

Keep track in map table and free list.

! false dep. resolved !

OoO & dynamic scheduling

- 100% in HW at runtime
- rename to avoid false dependencies
- exec insn ASAP
- Fetch many ins in insn window (current > 100 !)
- commit insns in order

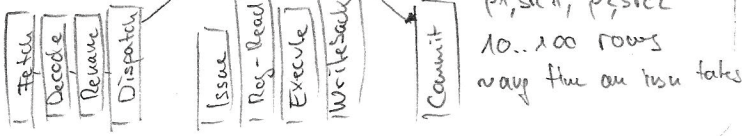


Ⓓ Dispatch
- allocate insn buf slot

Ⓢ Issue
- multiple exec units

Issue policy: If multiple insn are waiting, which to choose? → HW intensive

OoO pipeline



ROB: re-order buffer
entry used in exec
table containing insn #, use, exec, op, p1, src1, p2, src2
10..100 rows
way the insn takes

Speculative Execution

- Branch prediction: - Target address speculation → BTB (Branch target Buffer)
  - Condition resolution
  - Smith predictor, HW complex
- Trace-Cache: Fold out taken br by tracing insns as they commit into fill buffer
- loop-buffer: cache target itself instead of address
- Mispredict: - roll-back ROB - recover register renaming
- Speculative ld/st: - store buffer
  - exec all ld/st in pgm order
- adr speculation

st x1, (x2) } guess x2!=x4
ld x3, (x4) } guess x2==x4

Recap

3 ideas: renaming, speculative exec, in-order commit

complex, enables: OoO exec, multi-issue, loop unrolling, ...

Exec ASAP, commit in order

exception before commit? → flush pipeline

L3: Virtual Memory

How do apps share main memory?
↳ Goal: Each app thinks it has infinite mem

Solution: treat memory as "cache", store overflowed as "swap" on disk, add lvl of indirection

Virtual Memory (VM)

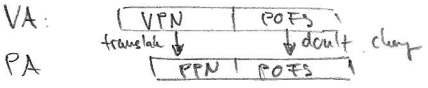
VA: virtual adr PA: physical adr

OS controls VA ↔ PA mapping

VA → PA at ~~page~~ granularity

→ Protection: Apps can't access each others memory

Address Translation



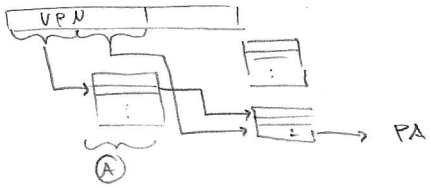
VPN: virtual page number
POFS: page offset
PPN: physical page number

Each process has a PT (page table) that maps VP → PP or to disk (swap)

Multi Level PT

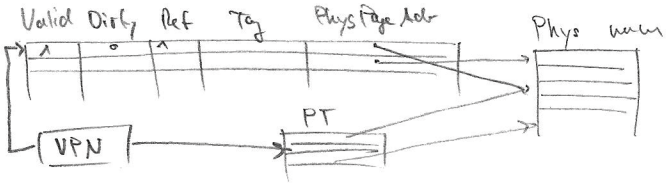
Problem: PT gets very large
Solution: Hierarchy in PT

large parts of (A) are unused → 2nd level entries need not exist and entry in (A) is NULL.



Fast translation using TLB

Use cache to store PTE (PT entries) in CPU, these accesses have good locality. They are called Translation Look-aside Buffer (TLB). Typical 16-512 PTEs, 0.5-1 cyc hit, 10-100 miss, 0.01-1% miss rate.



- TLB is 4+/fully associative between CPU & I$/D$
- Can perform TLB/$ access at same time bcs $ needs tag only at very end
- PT "walker": on TLB miss HW searches/walks table

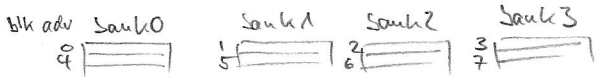
Memory Optimizations

6 basic cache optimizations

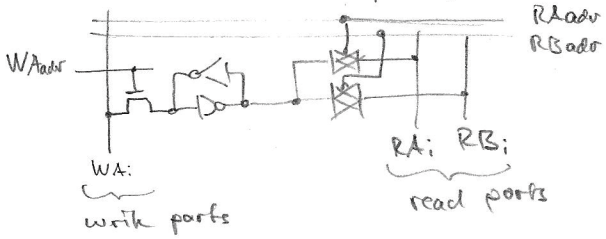
- larger block size
  ↳ reduce compulsory miss
- larger total $ capacity
  ↳ reduce miss rate
- Higher associativity
  ↳ reduce conflicts
- cache levels
  ↳ reduce overall access time
- Prio to read miss over write
  ↳ reduce miss penalty
- avoid adr transl. in $ indexing
  ↳ reduce hit time

Advanced Optimizations

- small & simple 1st lvl $
  - critical timing path
  - lower associativity reduces power
- way prediction
- pipelining cache
  - to improve bandwidth
  - makes it easier for associativity increase
- non-blocking cache
- multibanked cache
  - interleave banks according to block address



- True SRAM multiporting



- incr. area - slower access (wire delay)

- Compiler optimization
  - loop interchange
  - blocking
- HW prefetch
  - fetch two blocks on miss (incl. next blk)

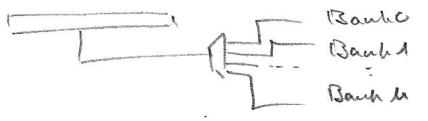
Dependability

Introduce error correcting codes to increase reliability.

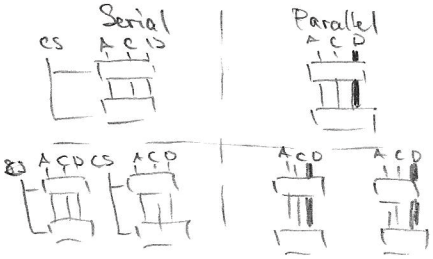
Example: Hamming SEC code. Parity bits

Main memory

Interleaved: - Break into M banks: Word A is in bank = A mod M at offset = A div M
- Banks can operate concurrently & independently



A = address
C = command
D = data



Non-interleaved

Interleaved

Parallel: increase adr width
Goal: Maximize requests to an open row

L4: Multicore

SOC 4

Classification

homogen. cores | hetero cores

- traditional multicore
- manycore
- big.LITTLE: small efficient, large powerful

Challenge: With more cores, memory & core to core communication gets difficult

parallel architecture

CPU CPU CPU ... — interconnect

Two models: Symmetric multi processor and distributed shared memory

SMP

centralized mem w/ uniform access time (UMA)

CPU $ ... — interconnect — memory

DSM

nonuniform access time (NUMA)

CPU $ memory ... — memory

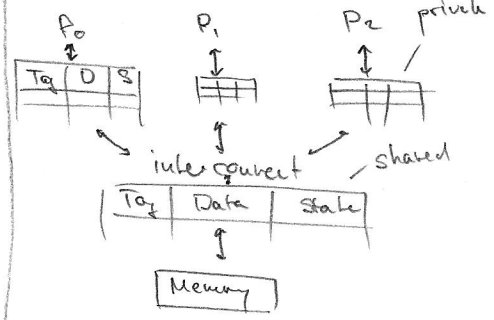
Non-shared memory: message passing
- simpler HW - explicit comm. - implicit synch

Shared memory: Open MP - ease of programming
- sharing large data - supports HW caching

Multicore

Shared memory issues

Cache Coherence



Coherency: A write followed by a read by the same proc P with no writes in between returns the value written

BROADCAST

Snooping: 3 states per cache line: invalid, shared, modified. A write invalidates all others. Get serialized.

optimized: two cache levels, snooping only on L2 required.

coherency miss = invalidation miss

MULTICAST

directories: better for scaling. Every memory has dir. keeps copy of cached lines & their states. On miss, comm only with the nodes that have a copy

Synchronization

Need to regulate access to shared data. → use of "locks"

lock: low level semaphore: high

region between acquire(lock) and release(lock) is a critical section

An interfering acquire() will block

A barrier is where all threads wait for all others to complete

spin lock acq(x): while(x!=0) {}; x=1;

↯ doesn't work ↯ acq is multi-op.

Solution: ISA provides atomic compare-and-swap CAS

```
cas r3 <- r1, r2, 0(&lock)
atomically executes
    ld r3 <- 0(&lock)
    if r3 == r2:
        st r1 -> 0(&lock)
```

now acquire: A0: cas r3, <-1, 0 0(&l)
A1: bnez r3, A0

- test-and-test-and-set spin repetitively
st → heavy interconnect use. better load to local cache and st if lock is free
coarse lock: easy but slow
fine lock: difficult but faster

Always acquire multiple locks in same order

omp others

reduction, master, single, ...

Memory Consistency

All processors "see" all ld/st in the same order

???

OpenMP

Directives & libraries → generates calls to threadly libraries s.a. pthread

omp parallel

code executed in parallel

id = omp_get_thread_num();

shared(a) declare a as shared mem

private(id) id is priv. to each thd

firstprivate copy in, private storage

lastprivate copy out, ——"——

omp for

splits/unrolls a loop

schedule(static) for simple regular loops with equal duration

schedule(dynamic) iteration space to work queue

↯ fine/coarse, parallelization overhead ↯

omp critical

section executed by only 1 thd at a time

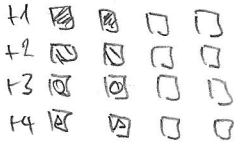
L5: SMT, Vector, SIMD

Categories ▪ t1 ▪ t2 ▪ t3 ▪ t4 ...

Superscalar

- single thread
- multi ops per cycle possible
- OoO

Fine-Grained

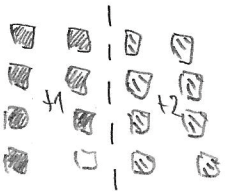


- fast context switches btw threads

Coarse-Grained

- slow switches

Multiprocessing



- two superscalars
- each working on a thread

SMT Simultaneous multi threading

- multi threads at single cycle

Simultaneous Multithreading: - performing multiple threads of execution in parallel
- replicate PC, RF but keep EX the same

Fine/Coarse grained: - can hide large/short stalls but not both

Example Adding SMT in Power5: - Incr. L1 associativity, - Add per thd ld/st queue
- add separate ins prefetch & buffering per thd - virtual regs 152 to 240 → A increase of 24%

Instruction and Data Streams

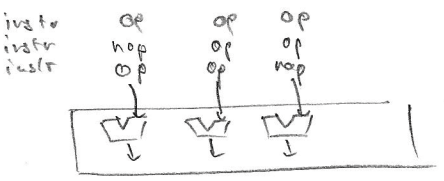
		data stream: single	data stream: multi
Instr Stream	single	SISD	SIMD
	multi	MISD	MIMD

Vector - Highly pipelined func unit - Simplify data-parallel pgm
- good with regular access patterns
- more general than ad-hoc media extensions

SIMD - Operate elementwise on vectors of data
- simplifies synchronization

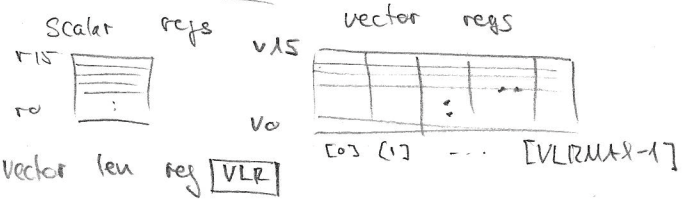
Multimedia Extensions
- Very short vectors added to existing ISAs for micros
- Limited instr set: no vector len control, no stride, ...

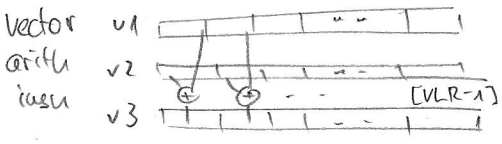
VLIW / **Vector** very long instr word HW without vector ISA

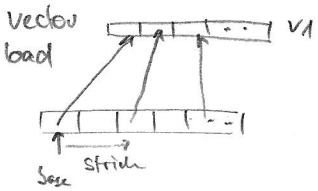


Vector Processor

Programming model







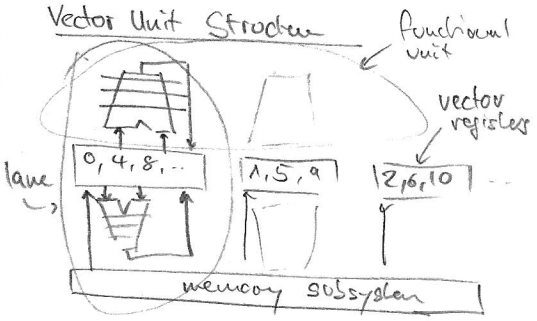
Advantages
- 1 instr encodes N operations
- expressive: o indep o same func unit o ...
- scalable

Mem layout
- perfect for strides 1,2
- Shitty for strides 4,8, ...

- mod 4 = 0 0 4
- mod 4 = 1 1 5
- mod 4 = 2 2 6
- mod 4 = 3 3 7

Conditional exec for (...) if a[i] ... then ... ✓ { maskable vector instructions

Vector Unit Structure



Ara is a vector processor for Ariane

reductions are slow. use trick to do vector MAC instead

Stripmining

Vector regs have finite len.
→ Break loops into fitting pieces and a remainder

Mem Op
- Unit stride: cont stack in mem
- Strided: prime number of data banks best supports different strides
- Indexed: scatter-gather sparse

GPU

Cores

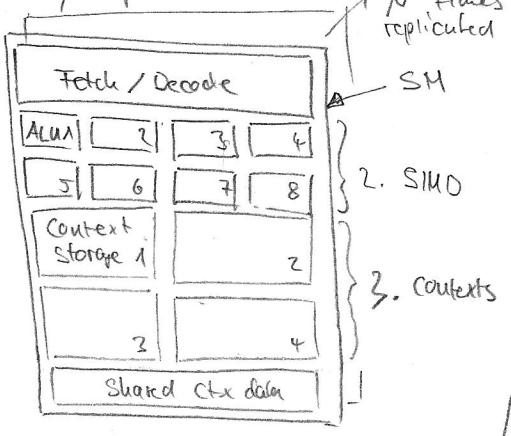
Tak CPU core and do 3 things

1. Stripping down

Remove components that help a single instr stream run fast

2. SIMD Amortize cost/complexity of managing an instr stream across many ALUs

3. Interleave processing of many fragments to avoid stalls due to high latency operations



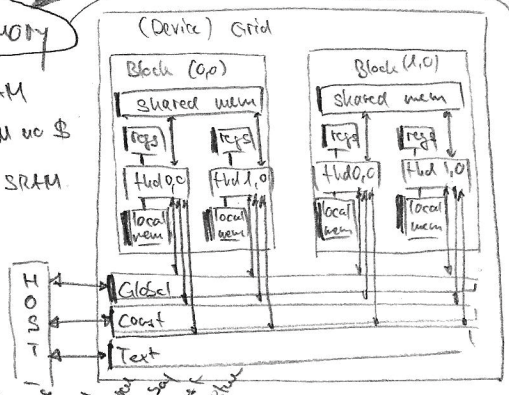
N: simultaneous instr streams
4·N concurrent interleaved instr streams
8·4·N concurrent fragments

Warps

HW groups thds that exec same instr into warps. A thd blk in HW is composed of warps s.t. all thds in a warp exec the same instr.

Memory

- DRAM
- DRAM w/o $
- Fast SRAM



	Reg	Local	Shared	Global	Const	Texture
thd	RW	RW	RW	RW	R	R
host	-	-	-	RW	RW	RW

global: main means to host/dev comm
cst/const: initialized by host

access times

1 cyc { reg, shared } DRAM w/o $ { local, global } DRAM w/ $ 1..100 cyc { const, texture }

Compared

	Programming Model	Architecture
Sequential	C →	RISC
Data Parallel	Vector →	SSE
Thread	CUDA →	SIMT

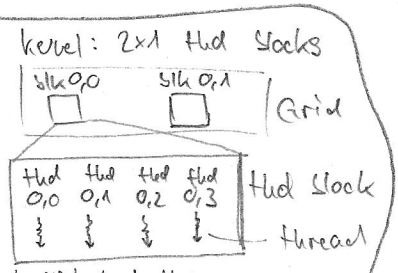
SIMT: Single instr multiple threads

CUDA Thread Hierarchy

Grid → Thd Blocks → Threads

```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A);
__device__ kernelF(A) {
  i = blockIdx.x;
  j = threadIdx.x;
  A[i][j]++;
}
```

kernel: 2×1 thd blocks



each blk: 4×1 thds

Blocks are dynamically scheduled

Threads can use shared memory
→ use sync barriers

Streaming multiprocessor

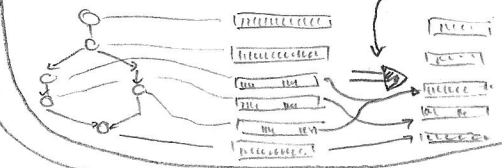
Vector SIMD vs SIMT

Vector	SIMT
- Programmer needs to know the # of PE	- # PE in HW is transparent to programmer
- Each inst. is executed by all PEs in lock step	- Exec ordering is handled by HW

SIMT: Programmer converts data level parallelism into thread level parallelism

Hazards

Thds within a warp are free to branch. If thds within a warp take same path, other path is not executed. If diverge both are executed and masks filter out the correct.



dynamic warp formation
merge 2 divergent warps into new warp if possible

Shared memory

- organized in banks, 1 bank per thd
- conflict if 2 thds in 1 warp access same bank
- data is programmer managed

Atomics & Reduction

- thds serialized in critical sec.
- CUDA provides atomic functions, also for reduction
- binary tree for reductions

Memcopy to/from host

- pinned memory allocs mem on host s.t. PCIe transfer is faster
- unified memory aims to enable virtual style memory
  ↳ only host pages moved to GPU

L8: Heterogeneous Architectures

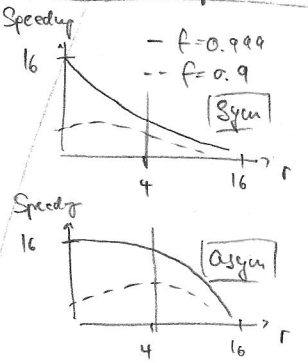
Amdahl's law (again)

$$\text{speedup}(f,n,r) = \frac{1}{\frac{1-f}{\text{perf}(r)} + \frac{f}{\text{perf}(r)\cdot\frac{n}{r}}}$$

f: parallelizable fraction
n: total resources (e.g. transistor count)
r: resources dedicated to each core
perf(n): a core's sequential performance $\approx \sqrt{r}$

Symmetric | asymmetric

Heterogeneous Architectures

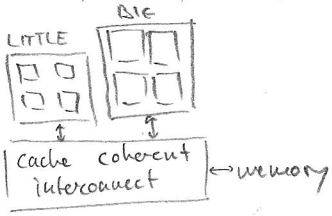


Observation: most real-world loads are complex
- parallelizable & non-parallelizable loads
- amenable to SIMD & divergent
- predictable data access & random

Idea: the most efficient processor is a heterogeneous mixture of resources.

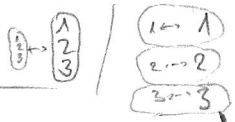
big.LITTLE

aims at power reduction. Smartphones have dynamically changing usage patterns.



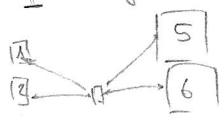
Exclusive Cluster Switching

OS runs workload on cluster that is powerful enough but least power hungry. If demand grows, cluster is switched. Can be done on cluster or core base. Unused turned off.



Inclusive core migration

OS scheduler has all cores at its disposal. Can also activate all at same time. Heterogeneous Multiprocessing (HMP).



Accelerators

On- and off-chip dedicated accelerators. CAPI: Coherent Accelerator Processor Interface.

Example: Mobile phone incl. modem, GPU, DSP, neural accelerator

CPU + FPGA

Zynq: Dual ARM + FPGA.

UltraScale+: 4x A53, 2x R5, Mali GPU, FPGA

Important: PS → PL interconnects

ASIP

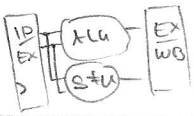
Application Specific Instr. set Processor

Motivation

GP processor too slow for some applications. ASIP designed for specific application. Compromise btw performance & flexibility adds custom insn to CPU commercial tools for fast devel.

SFU

Special function unit



can be:
- combinational like ALU
- with internal state
- with special register file (SPR)

implementation
- possible in embedded FPGA

E1: Single-Core RV MCU

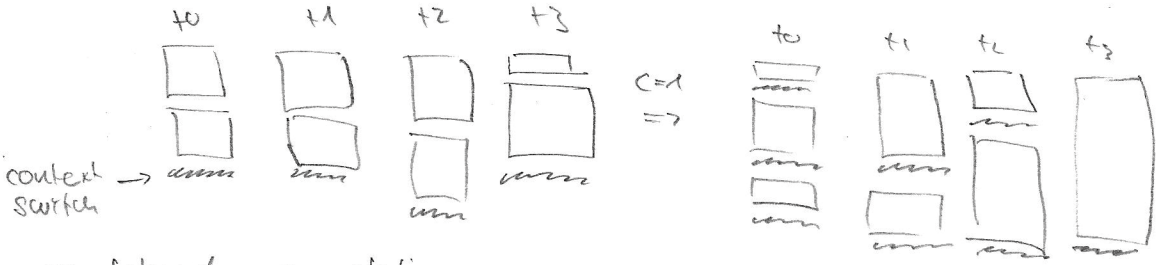
RISCY: RV32IMFCXpulp

I: Integer M: Mult/Div F: float C: compressed Xpulp: - Post incr ld/st
- HW loop
- packet SIMD
- Fixed point
- Bit Manipulation

Loop unrolling exploits instr-level parallelism

E2: OpenMP Programming

omp parallel for num_threads (N) schedule (dynamic, C)
↳ chunk size



- ~~for~~ balanced, use static
-

Fine granularity can increase total performance but overhead grows as well.

SPMD: single prog. multiple data → omp parallel for

Shared memory programming

MPMD: multi prog. multiple data → omp section /tasks
execute diff streams of instrs over possibly different data

omp nowait: removes the implicit sync barrier after a parallel/for/section

E4: Cuda

1. Data Need to malloc on device and copy data to it

2. Func `__global__ void add (int*a, ..)` - parameters must be pointers

call with: `add <<< N, M >>> (..)` N: number of threads blocks to create, can be 2D
M: number of threads per thread block, up to 3D

use blockIdx.x to access block index, blockDim.x for block size (# threads inside block)
use threadIdx.x to access thread index

threads can efficiently communicate & synchronize.

E5: Heterogeneous SoC

Zynq - Contains PS & PL and interconnects

- PL IP
- PL Datapath
- PS Driver
- PS Application

SDSoC: Software Defined System on Chip: From C to: an all-in-one tool.

- Do all in SW, if reqs not met, evaluate HW solution
- Trade-off between data movement cost & acceleration benefits

Zynq Ports: ACP (Accelerator Coherency Port)

- Has access to CPU cache hierarchy
- DDR/OCM access through ARM L1 & L2 cache
- Useful for data sizes that fit in cache L1: 32K L2: 256K

HLS: - Main tasks: schedule & binding
- Schedule: what op to do in which clock cycle
- Binding: map operators to HW blocks, share resources or duplicate

E6: HERO Heterogeneous SoC

HERO: Heterogeneous Research Platform

- single source, single binary cross compile toolchain
- OpenMP 4.5
- shared virtual memory for host & PMCA
- HERO includes the first non-commercial heterogeneous cross-compilation toolchain