Disclaimer

This summary is part of the lecture "Systems-on-chip for Data Analytics and Machine Learning" (227-0150-00L) by Prof. Dr. Luca Benini (FS20).

Please report errors to huettern@student.ethz.ch such that others can benefit as well.

The upstream repository can be found at https://github.com/noah95/formulasheets

Contents

ETH Systems-on-chip for Data Analytics and Machine Learning 2020

Noah Huetter

May 5, 2020

1 Architecture review, MCUs

1.1 Datacenters

Cloud Service Models

- Software as a Service (Saas)
  - Provider licenses applications to users as a service
  - Avoid costs of installation, maintenance, patches, ...
- Platform as a Service (Paas)
  - Provider offers software platform for building applications
  - e.g. Google's App-Engine
  - Avoid worrying about scalability of platform
- Infrastructure as a Service (Iaas)
  - Povider offers raw computing, storage and network
  - e.g. Amazon AWS
  - Avoid buying servers and estimating resource needs

Building blocks of moden data centers

- Network Switch
- Rack
- Server Racks
- Cluster Switch

Rack of servers (so called commodity servers) consist of a modular design with top-of-rack switch, power, network and storage. The ToR is the link aggregate to the next level.

A data center is not just a collectino of servers, it's a "small internet". It is administered as a single domain that does not have to be compatible with the "outside world". No need for international standards.

There exists specialized machine learning cloud hardware.

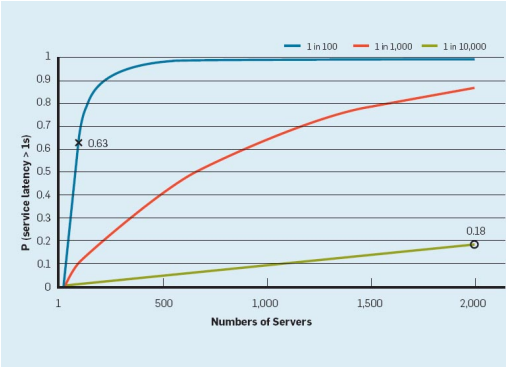
Performance Metrics

- Throughput
  - Requests per second
  - Concurrent users
  - Gbytes/sec processed
- Latency
  - Execution time
  - Per request latency

Tail Latency

Output depends on all servers finishing. Probability of a slow server P_{slow}. Job on N servers finishes when the last server is done. Even if P_{slow} is very small, P_{jobFsat} is small if N is large.

$$P_{\text{jobFast}} = (1-P_{\text{slow}})^N$$



TCO: Total Cost of Ownership

TCO = capital (CapEx) + operational (OpEx) expenses. CapEx: building, generators, HW. OpEx: Elec., repairs, insurance. Users perspective: CapEx: cost of long term leases on HW and services. OpEx: Pay per use cost on HW and services

Reliability

- Failure in time (FIT)
  - Failures per billion hours of operation
- Mean time to failure (MTTF)
  - Time to produce first incorrect output
- Mean time to repair (MTTR)
  - Time to detect and repair a failure

Steady state availability = MTTF/(MTTF + MTTR).

Multicore

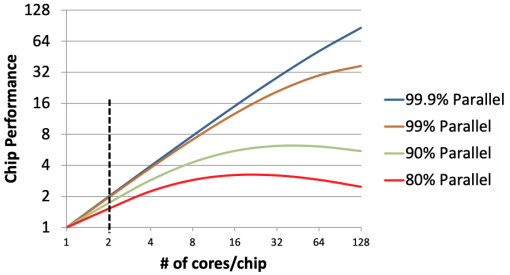
	Single	Dual	Quad
Core area	A	$\approx A/2$	$\approx A/4$
Core power	W	$\approx W/2$	$\approx W/4$
Chip power	$W+O$	$W+O'$	$W+O''$
Core performance	P	$0.9P$	$0.8P$
Chip performance	P	$1.8P$	$3.2P$

Amdahl's Law

Not all operations can run in parallel, hence speedup is limited. f fraction that can run in parallel.

$$\text{Speedup} = \frac{1}{(1-f)+\frac{f}{n}} \quad \lim_{n\to\infty} \frac{1}{1-f+\frac{f}{n}} = \frac{1}{1-f}$$

Amdahl's Law ignores power cost of n cores. More cores resupts in each core being slower. Parallel speedup $< n$. Seiral portion takes longer, also, inerconnect and scaling overhead. Fixed power budget forces slow cores, serial code quickly dominates.



1.2 Computing Systems Performance

Instruction Count and CPI

CPI: Cycles per instruction. CPU time = instruction count x CPI x clock cycle time. CPI is determined by program, ISA and compiler. Different instructions have diferent CPI.

Performance

IC: Instruction count. Depends on

- Algorithm: affects IC, possibly CPI
- Programming language, affects IC, CPI
- Compiler: affects IC, CPI
- Instruction set architecture: affects IC, CPI, Tc

Power

$$\text{Power} = \text{Capactivice load} \times \text{Voltage}^2 \times \text{Frequency}$$

Multiprocessors

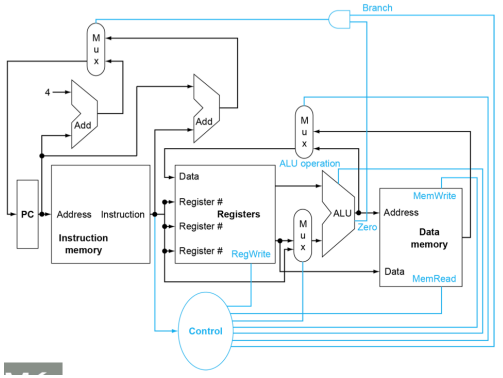
More than one processor per chip. Requires explicitly parallel programming. Hard to rogramm for performance, balance load and optimize synchronization.

1.3 Processor Architecture

CPU performance factors are determined by instruction count, CPI and cycle time.

Instruction Execution

Programm counter (PC) points to instruction memory to fetch instructions. Register numbers point to the register file to read registers. Depending on instruction class, use the ALU for arithmetic results, memory address calculation or branch compare. Access data memory for load/store and increment PC by target address of 4 (4 byte instruction words).



R-format instructions

Read two register from register file, perform arithmetic/logical operation and write back to register.

Load/Store Instructions

Read register operands, calculate address using offset, load memory to register or vice versa.

Branch Instructions

Read register, compare operands, calculate target address.

Pipelining

RISC-V is designed for pipelining: All instructions 32-bit, few regular instruction format, load/store addressing.

Hazards

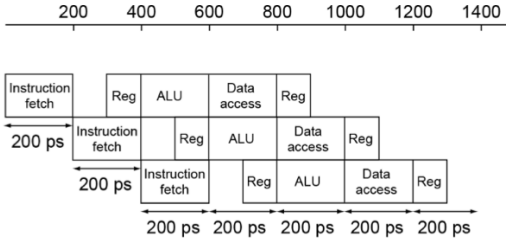
Situations that precent starting the next instruction in the next cycle.

- Structure hazard
  - A required resource is busy
  - Fix: requires separate instruction/data memories
- Data hazard
  - Need to wait for precious instruction to complete its data read/write
  - Fix: Forward data to next op without storing in register
  - Fix: Code scheduling to avoid stalls
- Control hazard

- Deciding on control action depends on precious instruction
- Fix: Branch prediction

Branch Prediction

Predict outcome of branch and only stall if prediction is wrong. RISC-V can predict branches not taken. Static prediction based on typical branch behaviour. Dynamic measures actual branch behaviour, e.g. record recent history of each branch. Assume future behav. will continue the trend, when wrong, stall while re-fetching. 2-Bit predictor changes prediction only on two successive mispredictions.



1.4 Exploiting Memory Hiearchy

Programs access a small portion of their address space at any time. Temporal locality aims at items that are accessed recently to be likely accessed again soon (e.g. loop). Spatial locality towards items near recently accessed (e.g. sequential instructions, array data).

Locality

Copy recently accessed (and nearby) items from disk to smaller DRAM (main memory) and from there to smaller SRAM (cache attahched to CPU). If accessed data is present, **hit**, else **miss**. Hit ratio: hits/accesses.

Cache Memory

- Direct Mapped Cache
  - Location determined by address
  - Direct mapped: only one choice
  - Store tags (high order bits of source data) and valid flag if there is data at that location
- Larger block size
  - Store tag, index and offset
  - Larger blocks should reduce miss rate but larger miss penalty
- Associative Cache
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
- n-way set associative
  - Each set contains n entries
  - Blcok number determines which set
  - Search all entries in a given set at once

AMAT: Average memory access time

AMAT = Hit time + miss rate x miss penalty

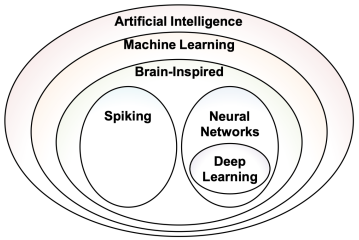
Replacement Policy

Direct mapped: no choice. Set associative: Prefer non-valid entry, if there is one. Otherwise, choose among entries in the set. Least recently used (LRU): Choose the one unused for the longest time. Random: Gives approx. same performance as LRU.

2 (Deep) NN Workloads

2.1 Artificial Intelligence

John McCarthy: "The science and engineering of creating intelligen machines"



2.2 Linear Classifier

If we want to classify an 32×32 image into 10 classes we can use the linear classifier:

$$f(x, W) = Wx + b$$

	Dimension	Description
W	10×3072	parameters or weights
b	10×1	bias or offset
x	3072×1	Input, e.g. 32x32x3 image pixels

Every column in W defines a plane. One side of this plane corresponds to images belonging to this class, the other side meaning it does not belong to this class.

To evaluate a linear classifier, a loss function L_i is defined. Given a dataset of examples $\{(x_i, y_i)\}_{i=1}^N$ the loss over the dataset is a sum of losses:

$$L = \frac{1}{N} \sum_i L_i \left(f(x_i, W), y_i \right)$$

x_i image
y_i is (integer) label

Multiclass SVM loss

The SVM (support vector machine) loss function with the shorthand $s = f(x_i, W)$:

$$\begin{aligned} L_i &= \sum_{j \neq y_i} \begin{cases} 0 & s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases} \\ &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \end{aligned}$$

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9	0	12.9

Other cost functions

Softmax	$-\log \left(\frac{e^{sy_i}}{\sum_j e^{sy_j}} \right)$
SVM	$\sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$
Full loss	$\frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$

Regularization

Model should be "simple", so it works on test data.

$$L(W) = L + \lambda R(W)$$

2.3 Training Linear Classifier

How do we find the best W using a dataset, a score function ($s = f(x, W)$) and a loss function? Enter **gradiend-based optimization**: The slope in any direction is the dot product of the direction with the gradient.

```
while True:
  weights_grad = eval_gradient(
    ↪ loss_fun, data, weights)
  weights += -step_size*weights_grad
```

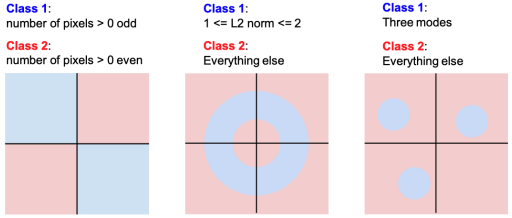
Stochastic Gradient Descent (SGD)

The full sum in the cost function is expensive. Aprroximate it using a **minibatch** of randomly selected data (32/64/128 common).

```
while True:
  data_batch = sample_train_dat(data,
    ↪  64)
  weights_grad = eval_gradient(
    ↪ loss_fun, data_batch, weights)
  weights += -step_size*weights_grad
```

Hard cases

Most data are not linearly seperable.



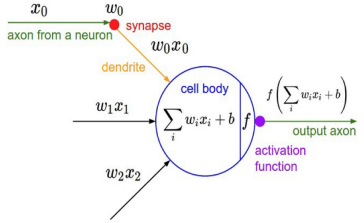
2.4 From linear to non-linear classifiers

Instead of feeding the classifier with raw data, perform a non-linear transformation beforehand so that the data becoms linearly seperable. This is called the **Kernel Trick**.

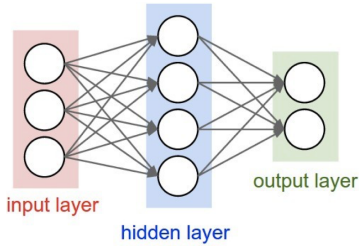
Later we will see that convolutional netowrks can work on raw data so that no effort has to be made on pre-processing.

2.5 Neural Networks

Consists of interconnected cells. Each cells has multiple inputs that are weighted and added (like a linear classifier), but then put through a non-linear activation function.



Multiple cells are connected in several layers.

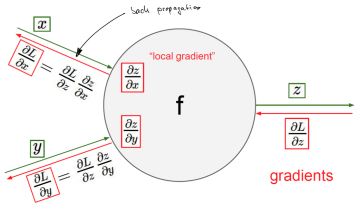


2.6 Deep Learning

A deep neural network is a neural network with a lot of hidden layers.

2.7 Training Neural Networks

Neural nets are trained based on gradient descent using backpropagation. To compute the gradient, input values are propagated from front to back through the network and then the partial derivatives calculated during backpropagation.



add gate	gradient distributor
max gate	gradient router
mul gate	gradient switcher

Mini-batch SGD

1. **Sample** a batch of data
2. **Forward** propagate it through the graph, get loss
3. **Backward** propagate to calculate the gradients
4. **Update** the parameters using the gradient

Non-linearity

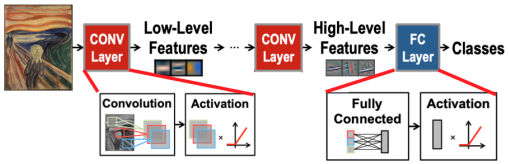
The simplest is the so called **ReLU** (rectified linear unit).

$$f(x) = \max(0, x)$$

- Does not saturate
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice
- Actually more biologically plausible than sigmoid

2.8 Convolutional Neural Networks

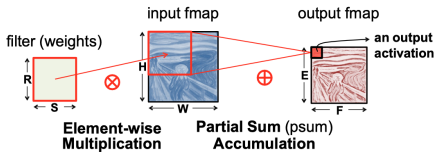
Convolutional Neural Networks are neural networks with feed forward and sparsely-connected with weight sharing. They are trainedusing supervised learning (training set has inputs and outputs, i.e., labeled). The main two layers are **convolutional layers** and **fully connected** layers.



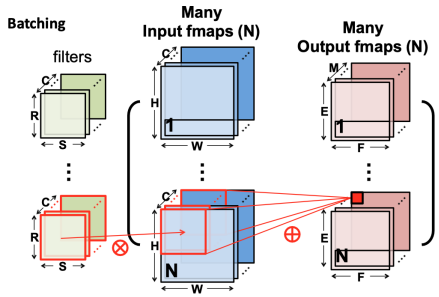
The convolution layer can further be dissected into convolution, non-linearity, norm and pooling.

Convolution

The input fmap (feature map) is element-wise multiplied with filter coefficient and partially summed to get one entry of the output fmap. The filter is then shifted one entry to get the next output value and so on.



There can be many input channels C, filters M and output channels M.



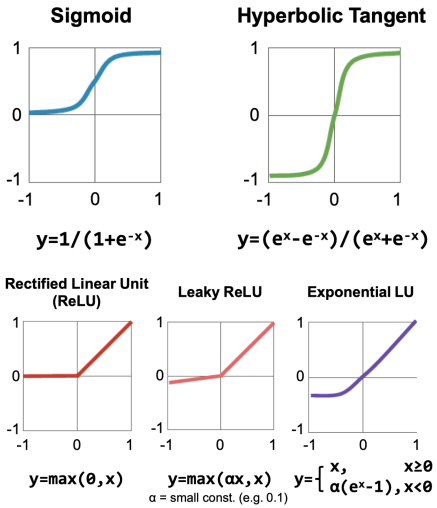
$$O[n][m][x][y] = \text{Activation}(\boldsymbol{B}[m]) + \sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{k=0}^{C-1} \boldsymbol{I}[n][k][Ux+i][Uy+j] \times \boldsymbol{W}[m][k][i][j]$$

$$0 \le N, - \le m < M, 0 \le y < E, 0 \le x < F$$
$$E = (H - R + U)/U, F = (W - S + U)/U$$

Sym	Description
$\boldsymbol{B}$	Biases
$\boldsymbol{I}$	Input fmaps
$\boldsymbol{W}$	Filter weights
U	Convolution stride

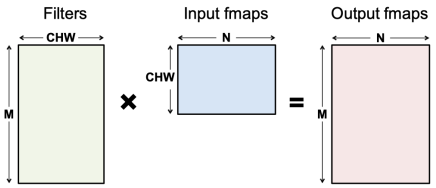
Sym	Name	Description
N	batch size	number of in/out fmaps
C	channels	number of 2D in maps/filters
H	activations	Height of input fmap
W	activations	Width of input fmap
R	weights	Height of 2D filter
S	weights	Width of 2D filter
M	channels	Number of 2D output fmaps
E	activations	Height of output fmap
F	activations	Width of output fmap

Non-linearity



Fully connected (FC) layer

Height and width of output fmaps are 1 ($E = F = 1$), filters as large as input fmaps ($R = H, S = W$). Implementation using matrix multiplication.

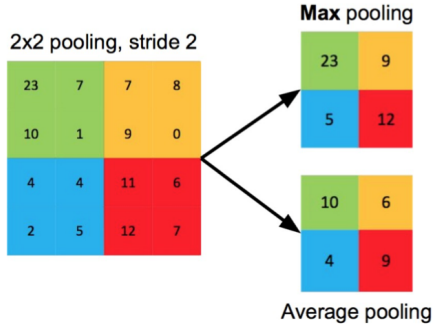


Norm

Batch normalization (BN) normalizes activations towards mean=0, sigma=1 based on statistics of training dataset. Put in between CONV and activation. Believed to be key to getting high accuracy and faster training on very deep neural nets.

Pooling

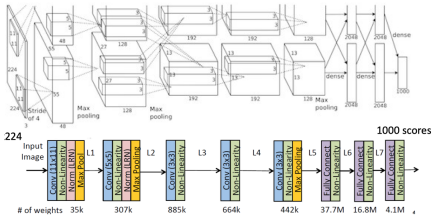
Reduce resolution of each channel independently. Overlapping or non-overlapping, depending on stride.



2.9 Example nets

AlexNet

The AlexNet has 5 conv layers, 3 FC, 61M weights, 724M MACs and uses ReLU as non-lin.

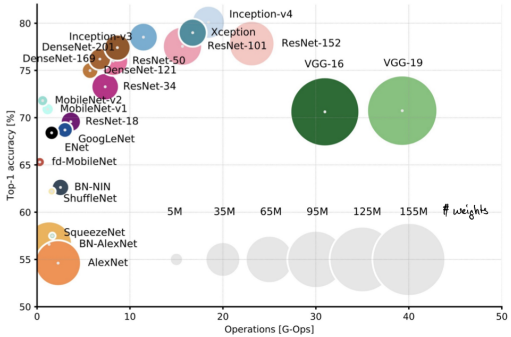


L#	Filter	#Filters	#Chan	Stride
1	11×11	96	3	4
2	5×5	256	48	1
3	3×3	384	256	1
4	3×3	384	192	1
5	3×3	256	192	1

Altough layers 1 and 3 have approx. same number of MACs (120M), L3 is significantly more complex to compute because it has 26x more parameters (memory usage).

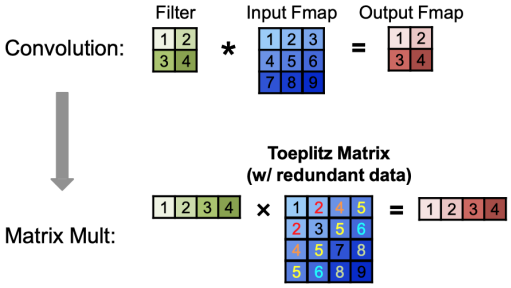
Summary

Metrics	LeNet-5	AlexNet	VGG-16	GoogLeNet (v1)	ResNet-50
Top-5 error	n/a	16.4	7.4	6.7	5.3
Input Size	28x28	227x227	224x224	224x224	224x224
# of CONV Layers	**2**	**5**	**16**	**21 (depth)**	**49**
Filter Sizes	5	3, 5,11	3	1, 3 , 5, 7	1, 3, 7
# of Channels	1, 6	3 - 256	3 - 512	3 - 1024	3 - 2048
# of Filters	6, 16	96 - 384	64 - 512	64 - 384	64 - 2048
Stride	1	1, 4	1	1, 2	1, 2
# of Weights	2.6k	2.3M	14.7M	6.0M	23.5M
# of MACs	283k	666M	15.3G	1.43G	3.86G
# of FC layers	**2**	**3**	**3**	**1**	**1**
# of Weights	58k	58.6M	124M	1M	2M
# of MACs	58k	58.6M	124M	1M	2M
Total Weights	**60k**	**61M**	**138M**	**7M**	**25.5M**
Total MACs	**341k**	**724M**	**15.5G**	**1.43G**	**3.9G**



2.10 Kernel Computations

Convolutions can be reshaped to form standard matrix multiplications. GPUs are very good in matrix multiplications. Problem is that data is repeated.



Likewise linear classifiers can be reshaped to form matrix multiplications.

2.11 Computational Transforms

Goal: Bitwise same result, but reduce number of operatinos. Focuses mostly on compute.

Strassen

Uses partial products to reduce the number of multiplications. Reduces matrix multiplication complexity from $\mathcal{O}(N^3)$ to $\mathcal{O}(N^{2.807})$. Comes at the price of reduced numerical stability and requires significantly more memory. Performance gain only for large matrices.

Winograd

1D Winograd targets convolutions instead of matrix multiply. Reduces number of multiplications from, e.g., 36 to 16 for 3 by 3 filter and 4 by 4 input fmap. Winograd works on small regions of output at a time, and therefore uses inputs repeatedly.

- Optmized computation for convolutions
- Can significantly reduce multiplies
- Each filter size is a different computation

FFT

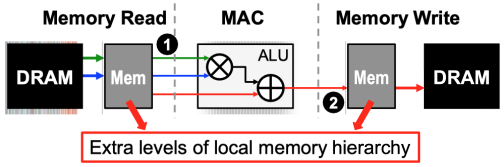
Convolution in time domain becomes multiplication in frequency domain. Reduces convolution complexity from $\mathcal{O}(N_{O_2} N_{f_2})$ to $\mathcal{O}(N_{O_2} \log_2 N_O)$. Comes at the cost of more memory space and bandwidth.

More compute reduction approaches

- Reduce size of operands
  - Floating point to fixed point
  - Bit-width reduction
  - Non-linear quantization
- Reduce number of operations
  - Exploit activation statistics
  - Network pruning
  - Compact network architectures

2.12 HW-centric View

Most operations are based on multiply-accumulate (MAC).



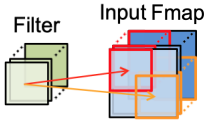
Opportunities

- Data reuse
- Partial sum accumulation does not have to access RAM

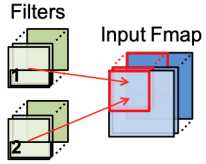
Types of data reuse

Data reuse can reduce DRAM reads of filter/fmap by up to 500x (for AlexNet)

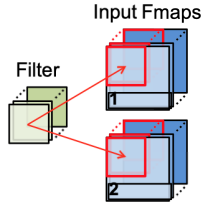
Convolutional Reuse CONV layers only: reuse activations and filter weights.



Fmap Reuse For CONV and FC layers: can reuse activations.



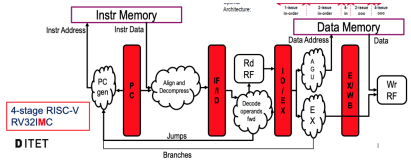
Filter Reuse CONV and FC layers with bacht size > 1: Reuse filter weights.



Tuning Processors for CNNs

Single issue, in order is the most energy efficient.

- Memory is always critical: 1 full stage for fetch, 2 full stages for load+writeback
- Single stage decode
- Single stage execute+WB



ISA extensions

- HW loops and post modified LD/ST
- Bit manipulations
- Packed-SIMD ALU operations with dot product
- Rounding and Normalization
- Shuffle operations for vectors

RISC-V3 implements all these features. **HW loops** mitigate counter increments and branch overheads by specifying the number of iterations in the loop instruction.

Bit manipulations E.g. extract N bits starting from M from a word and extend with sign or find first bit set, count bumbers of 1, rotate, ...

Packed-SIMD NN inference does not need 32 bit precission. Back e.g. 4 8 bit values into one 32 bit register and run operation on all simultaneous (SIMD=single instruction, multiple data). Can run 4 8-bit MAC in one cycle.

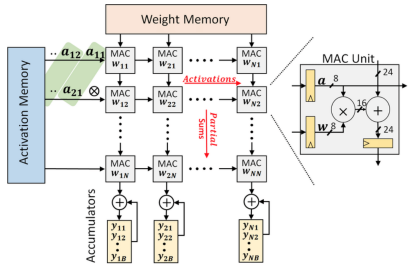
MMUL on serial, parallel, systolic

MMUL = matric multiplication.

Serial calculation is done by a standard CPU. Load activations and fmaps from memory, perform single MAC, store back in memory. Very slow.

Parallel is easily implemented in GPUs. Large number of small processing units have their own small memory. Each performs serial computation. Better than serial but still slow.

Systolic Arrays Load weights and fmaps once and propagate through the array to compute MMUL. This is how modern NN accelerators work (e.g. Google TPU). Con: if array is too small, partial sums have to be stored and the array behaves like a standard CPU.



MMUL in memory Single memory cell used to store data (binary or multi-level voltage). By word line activation, bits get added or multiplied and result can be measured.

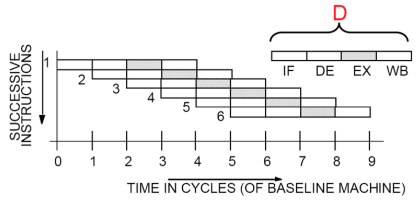
3 Advanced Processors

3.1 Terms

- Instruction parallelism
  - Number of instructions being worked on
- Operation Latency
  - Time (in cycles) until the result of an instruction is available for use as an operand in a subsequent instruction
- Peak IPC
  - maximum sustainable number of instructions executed per clock cycle

3.2 Instruction-Level Parallelism

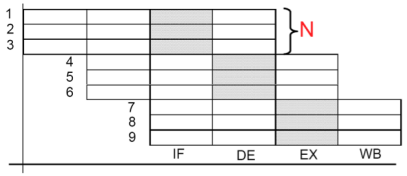
IF	Instruction Fetch
DE	Decode
EX	Execute
WB	Write Back



Superscalar Machine

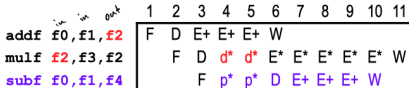
A **superscalar machine** is able to execute multiple operations during a single clock cycle.

Instrucion parallelism	$D \times N$
Operation latency	1
Peak IPC	N



Pipeline problems

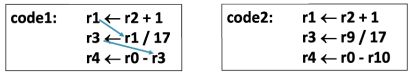
The pipeline can **stall** due to a raw hazard (because data dependence) or due to pipeline hazard (because of stuck pipeline). Here, `subf` can't proceed into D because `mulf` is there.



An **in-order pipeline**, often written as F, D, X, W, is vulnerable to such harards. **Out-of-order pipeline** implements "passing" funcionality by removing structural hazards.

Instrucion-Level Parallelism (ILM)

ILM is a measure of the amount of interdependencies between instructions. Average ILP = number of instructions / number of cycles required. Code 1 has ILP = 1, code 2 has ILP = 3. All three instructions could be executed simultaneously.



Takeaway: Out-of-order execution exposes more ILP.

3.3 Out of Order Execution

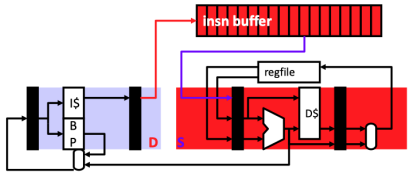
Or dynamic scheduling, uses a window of instructions, reorders them at runtime to exploit maximum pipeline usage.

- Dynamic scheduling
  - Totally in hardware
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past
  - Flush pipeline on branch misprediction
- Rename to avoid false dependencies (WAW and WAR)
- Execute instructions as soon as possible
  - Register dependencites are known
  - Handing memory dependencies more tricky
- Commit instructions in order
  - Any strange happens before commig, just flush pipeline
- Current machines: 100+ instructino scheduling window

Motivation: Execute instructinos in non-sequential order but make it appear like sequential execution. Reduce RAW stalls, increase pipeline and funcional unit utilization, epose more opporunities for parallel issue.

Big picture

Fill instruction buffer with ready to execute instructions. **Dispatch**: first part of decode. Allocate slow in insn buffer, stall back-propagates to youger insns. **Issue**: second part of decode. Sends insns from ins buffer to execution units. Out-of-order: wait doesn't back-propagate to younger insns.



Issue: If multiple insns are ready, which one to choose? Most project use random.

Data dependencies

Definition: If insn 1's result is needed for insn 1000, there is a dependency. It is only a hazard, if the hardware has to deal with it.

True data dependency: **RAW** (read after write) prevents reordering. False dependency: **WAW** (write after write) and **WAR** (write after read) use same CPU register, but reordering is not possible because they use same reg. This can be made reorderable by **renaming**.

Register Renaming

Concept: The register names are arbitraty and only nneds to be consistent between writes.

Approach: Every time an architected reg is written, we assign it to a physical register. Until the arch reg is written again, we continue to translate it to the physical reg. Leaves RAW dependencies intact. Free ROB at commit. Two key data structures: `map_tbl` maps from arch register to physical reg, free list keeps track of allocated and free registers, implemented as a queue.

```
i.phys_i1 ← map_tbl[i.arch_i1]
i.phys_i2 ← map_tbl[i.arch_i2]
i.old_phys_o ← map_tbl[i.arch_o]

new_r ← new_phys_reg()
map_tbl[i.arch_o] ← new_r

i.phys_o ← new_r
```

At commit, once all older ins have committed, free register.

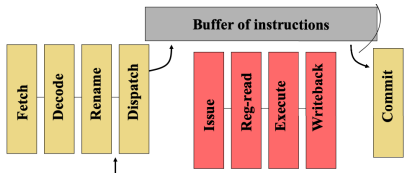
```
free_phys_reg(i.old_phys_o)
```

Example:

```
xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1
```

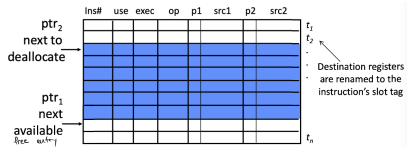
```
xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9
```

Pipeline:



Reorder Buffer

Architectural registers and memory may only be changed if all previous operations are also written. Arch regs & mem writes are always executed **in order**. This leads to the reorder buffer ROB. Any instructions in ROB whose RAW hazards have been satisfied can be dispatched.

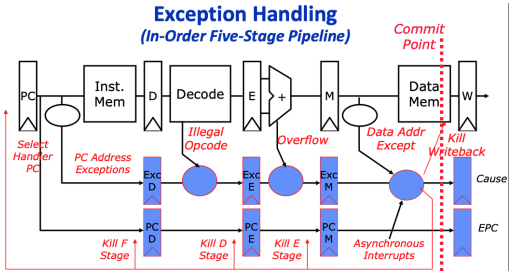


- ROB managed circularly
  - exec bit set when ins begins execution
  - When an ins completes, its use bit is marked free
  - ptr2 is inc only if the use bit is marked free
- Ins slot is candidate for execution when:
  - It holds a valid ins (use bit set)
  - It has not already started exec (exec bit clear)
  - Both operands are acailable (p1 and p2 set)

Buffer size: Big enough to host average age of instruction. Approximately execution time of instruction. In practice 10..100.

Interrupts and Exceptions

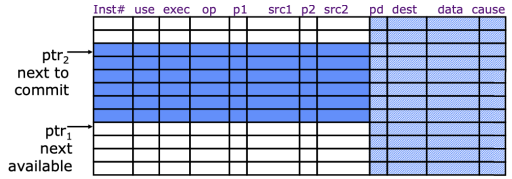
- Hold exection flags in pipeline unti commit point
- Exceptions in earlier pipe stages override later exceptions
- Inject external interrupts at commit point
- If exception at commit, update cause and EPC reg, kill all stages, inkect halder PC into fetch stage



Commit

Because of exceptions, temporary storage is needed to hold results before commit (shadow registers and store buffers).

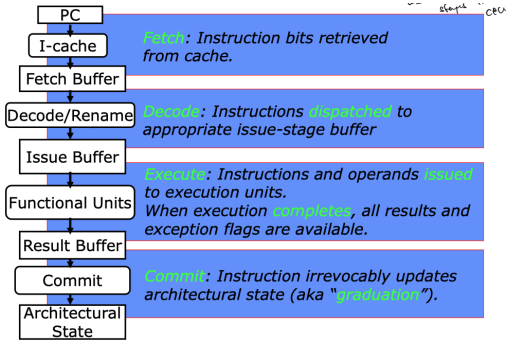
- Add pd, dest, data, cause fields in ins template
- Commit ins to reg file and memory in program order, buffers can be maintained circularly
- On exception, clear reorder buffer by resetting ptr1=ptr2 (stores must wait for commit before updating memory)



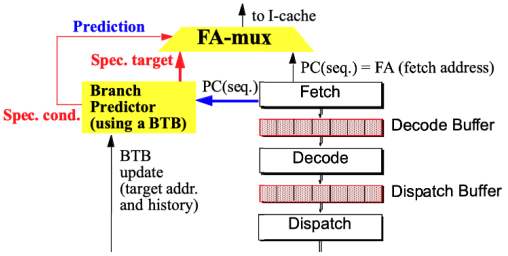
Register file does not contain renaming tags anymore, how does the decode stage find the tag of a source reg? Search the "dest" field in the ROB.

Branching and Prediction

With that many pipeline stages, branching becoms a challenge. Next operation is result of branch operation, which is a heavy dependency. If high speed has to be achieved, multiple branches need to be speculated correctly.



Branch prediction can be divided into:
- Target address generation (Target speculation)
  - Access reg: PC, general purpose reg, link reg
  - Perform calculation: +/- offset, autoincrement
- Condition reolution (Condition speculation)
  - Acces reg: Condition code reg, general purpose reg
  - Perform calculation: Comparison of data reg(s)



The branch target buffer (BTB) keeps track of taken branches. Branch predictor logic takes usually more area than FPU.

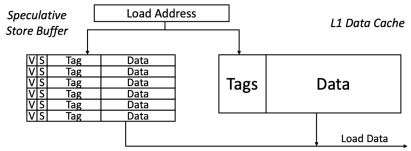
In case of branch miss, ROB is rolled back.
- ROB entry holds all info for recovery/commit
  - All ins & in order
  - Arch reg names, physical reg names, ins type
  - Not removed until very last thing (commit)
- Operation
  - Fetch: insert at tail (if full, stall)
  - Commit: Remove from head (if not yet done, stall)
- Tracking for in-order commit
  - Maintain appearance of in-order execution
  - Used also to support misprediction recovery

Trace Cache

Based on branch predictions, the trace cache stores blocks of instructions of streaming instructions considering also jumps and function calls to allow high-bandwidth fetch.

Loads and Stores

As already mentioned, no instruction is allowed to modify memory out of order. Nothing may change until ready to commit. A speculative store buffer is a structure introduced to hold speculative store data.
- During decode, store buffer slot allocated in program order
- Soters plit into "store adr" and "store data" micro-ops
- "Store adr" execute writes tag
- "Store data" execute writes data
- Store commits when oldest ins and both adr and data avail
  - Clear speculative bit and eventually move data to cache
- On store abort
  - Clear valid bit



On load: check if adr is in store buf and/or cache. Do store buffered and cache request at same time for performance. If ¿1 entry in buf with matching adr, use the youngest entry.

```
sd x1, (x2)
ld x3, (x4)
```

Another problem with loads and stores is the one above. If the address is not known, any load coult conflict with any previous store. We therefore can't load if a store is outstanding.
- Execute all loads and stores in program order
  - Load and store cannot leave ROB for ecevution until all previous loads and stores have completed execution
- Can still execute loads and stores speculatively, and out-of-order with respect to other insns
- Need a structure to handle memory ordering
- Conservative OoO load execution
  - Can execute load before store, if adr known and not equal (x4!=x2)
  - Each load adr compared with adr of all prev. uncommitted stores
  - Don't exec load if any prev store adr not known
- Address speculation
  - Guess that x4 != x2
  - Execute load before store adr known
  - Need to hold all completed but uncommitted ld/sd adr in program order
  - If subsequently find x4 == x2, squash ls nd all following instructions
  - Large penalty for inaccurate adr speculation
- Memory Dependence Prediction
  - Guess that x4 != x2
  - Execute load before store adr
  - If later find x4 == x2, squash ld and all following ins, but mark ld ins as store-wait
  - Subsequent exec of same ld ins will wait for all prev sd to complete
  - Preiodically clear store-wait bits

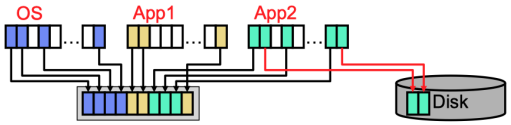
Summary
- Dynamic scheduling
  - Totally in hardware
  - Also called out-of-order execution
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past branches
  - Flush pipeline on branch misprediction
- Rename registers to avoid false dependencies
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies is harder
- commit instructions in order
  - Anything strange happens before commit, flush pipeline

3.4 Virtual and Advanced Memory

Virtualizing

Problem when running multiple apps and the OS: How do they share memory? **Goal**: Each application thinks it has infinite memory.
- Treat memory as a "cache" and store overflowed blocks in "swap" space on disk
- Add a level of indirection (address translation)

Virutual Memory (VM):
- Level of indirection
- App generated adresses are virtual addr. (VA)
  - Each proc can act as if has its own 2^N B adr space
- Mem accessed usin physical adr (PA)
- VA translated to PA at some coarse granularity (page)
- OS contorls VA to PA mapping for itself and all other proc
- Logically: translate performed before ecery insn fetch, ld, sd
- Physicallu: HW acceleration remoces translation overhead

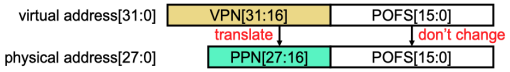


- programs use virtual adr (VA)
  - VA size (V) aka V-bit ISA (e.g., 64-bit x86)
- Memory uses physical adr (PA)
  - 2^M is most phy mem machine supports
- VA → PA at page granulariy (VP→PP)
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap) or nowhere (if not yet touched)

Uses:
Isolation: Each app thinks it has its own memory. Apps can't address another program's memory.
Protection: Each page with a r/w/ex permission set by OS. Enforced by hardware.
Inter-process com: Map sampe phys pages into multiple virtual adr spaces. Or share files via UNIX `mmap()`

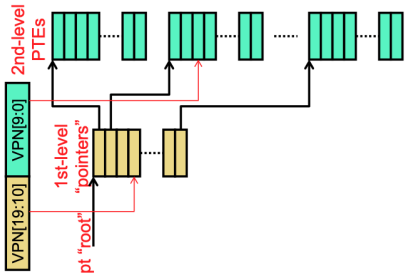
Address Translation

VA→PA mapping called address translation. Split VA into virtual page number (VPN) and page offset (POFS). Translace VPN into physical page number (PPN). POFS is not translated. Translation is done in software (for now) with HW acceleration. OS has a page table (PT) that maps VPs to PPs ot to disk (swap) adr. Translation is table lookup.



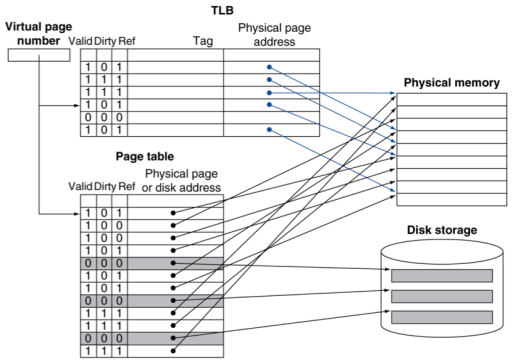
Replacements and writes:
- To reduce page fault rate, prefer least recently used (LRU) replacement
  - Reference bit (aka use bit) in PT set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference but = 0 has not been used recently
- Disk writes take millions of cycles
  - Block at once, not individual locations
  - Write through is impractical
  - Use write-back
  - Dirty bit in PT set when page is written

Problem: Page tables can get big. One solution is multi-level page tables. A trie of pages, lowest level tables hold PTEs. VPN is split into N levels.

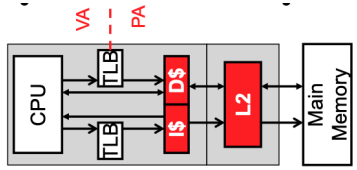


Fast Translation

Problem with virtual memory is that a ld/sd op would first have to access PT then the actual memory access. But this access has good locality. Therefore use a fast cache of PTEs within the CPU called **Translation Look-aside Buffer (TLB)** Typically 16-512 PTEs , 0.5-1 cycle for hit, 10-100 for miss, 0.01-1% miss rate. Misses can be handled by HW or SW.



TLB misses load PTE from memory and retry. If page not in memory, OS handles fetching and update of table. TLB miss indicates page present, but PTE not in TLB or page not present. Must recognize TLB miss before destination register overwritten →raise exception.



Summary

TLB should "cover" the size of on-chip caches.

Abbr.	Text
VM	Virtual Memory
PA	Physical address
VP	Virtual page
PP	Physical page
VPN	Virtual page number
POFS	Page offset
PPN	Physical page number
PT(E)	Page table (entry)
LRU	Least recently used
TLB	Translation Look-aside Buffer

3.5 Cache

Direct mapped

In a direct-mapped cache structure, the cache is organized into multiple sets with a single cache line per set. Based on the address of the memory block, it can only occupy a single cache line.

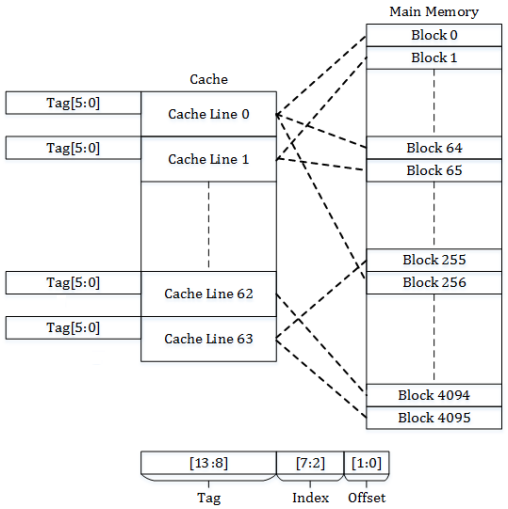


Figure 1: Direct mapped $

- Place block in cache
  - Set determined by idx bit derived from mem adr
  - Place mem block and set tag
  - If cache line previousy occupied, new data replaces
- Search word in cache
  - Set is identified by idx bit of adr
  - If tag match hit, else miss
- Advantages
  - Placement is power efficient as it avoids search
  - Place and replace is simple
  - Cheap HW as only one tag needs to be checked at a time
- Disadvantages
  - Lower cache hit rate

Fully associative

n a fully associative cache, the cache is organized into a single cache set with multiple cache lines. A memory block can occupy any of the cache lines.

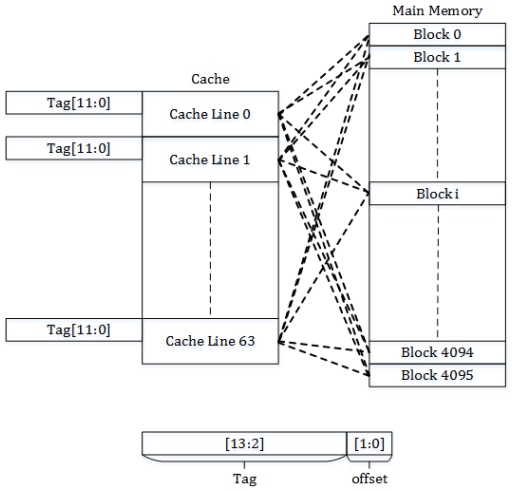


Figure 2: Fully associative $

- Place block in cache
  - Mem block can be placed if valid bit = 0
  - If cache is completely occupied, a block is evicted and mem is placed in that line
- Search word in cache
  - Tag field of mem adr is compared with tag in all cache lines. If match, block is present in cache and is a hit.
  - Based on offset, byte is selected and returned
- Advantages
  - Any mem block can be placed in any cache line, full cache utilization
  - better hit rate
- Disadvantages
  - placement is slow and power inefficient
  - High cost due to comparison HW

Set associative

Set-associative cache is a trade-off between direct-mapped cache and fully associative cache. A set-associative cache can be imagined as a $(n \times m)$ matrix. The cache is divided into n sets and each set contains m cache lines. A memory block is first mapped onto a set and then placed into any cache line of the set. The range of caches from direct-mapped to fully associative is a continuum of levels of set associativity. (A direct-mapped cache is one-way set-associative and a fully associative cache with m cache lines is m-way set-associative.)

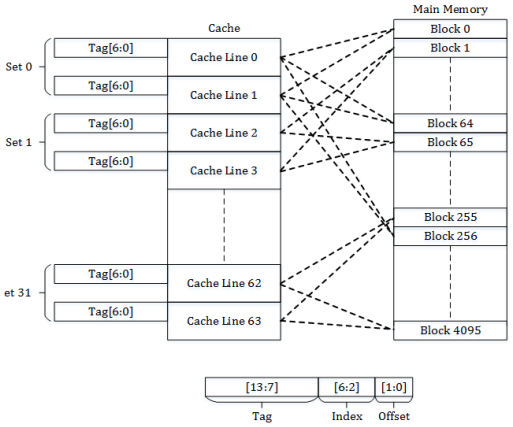


Figure 3: Set associative $

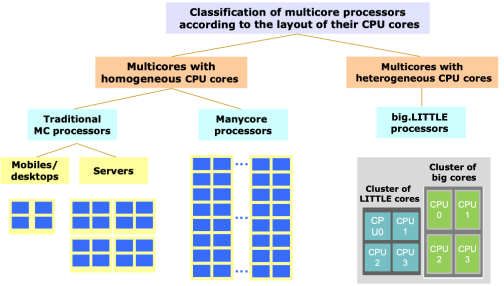
- Place block in cache
  - Set is determined by index bits derived from memory adr
  - Place memory block and store tag with associated set
  - If cache line occupied, new data replaces old
- Search word in cache
  - Set is determined by index bit from memory adr
  - Compare tag bits with all tags of selected set. If match then hit, else miss
- Advantages
  - Placement is trade-off between direct map and fully associative
  - Offers flexibility of replacement algos if a cache miss occurs
- Disadvantages
  - Placement will not effectively use all availabe cache lines

4 Multicore Processors

4.1 Today's Chips

- Multiple Cores per chip yields higher performance
- Multiple memory hierarchy
  - L1 private to processor
  - L2 private to processor
  - L3 distributed and shared across procesor
- Up to 200W TPU

4.2 Classification



- On energy constrained devices with varying task load big LITTLE architecture is best
- Task scheduling is a big deal for heterogenous architectures in orde to decide on which hardware a task runs

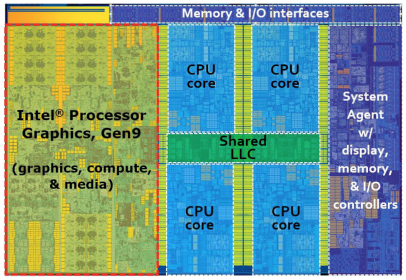


Figure 4: Skylake Intel 14nm, high performance, GPU, IO

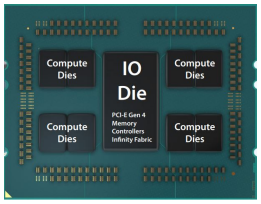
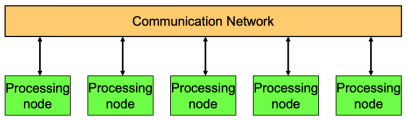
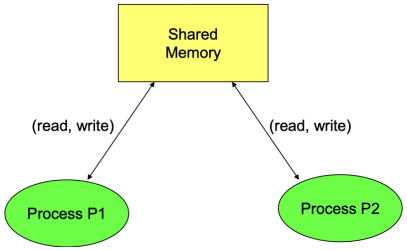


Figure 5: AMD chiplet, cores on different dies, one main die for IO, L3$

4.3 Parallel Architectures



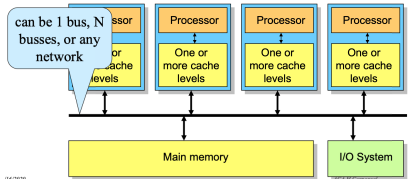
Shared Memory



- To commuicate from one proc to the other, write to specific memory
- Key problems
  - Cherence problem: copy to all processors local cache
  - Memory consistency issue: modify all copies
  - Synchronization problem: Only read if data is ready
- Two variants
  - Physical shared (SMP)
  - Distributed shared (DSM)

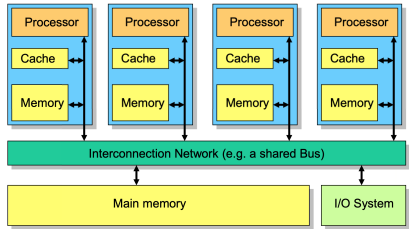
Symmetric Multi-Processors SMP

Memory: Centralized with uniform access time **UMA** and bus interconnect, IO



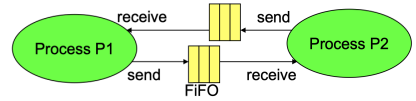
Distributed Shared Memory DSM

Nonuniform acces time **NUMA** and scalable interconnect (distributed memory)

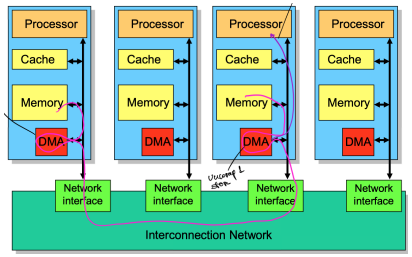


Message Passing

Is a communication model using communication primitives, e.g, send, receive library calls. Message passing can be built on top of shared memory and vice versa.

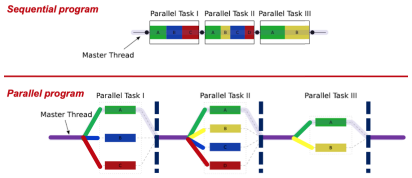


- DMA reads from memory, wraps message, sends on network
- DMA listens on network parses message, writes to memory and inform processor
- There is no linear speedup due to sequential program and communication overhead
- Latency hiding: do something while waiting
- Easier scalable



Shared memory programming

- OpenMP compiles code to multiple independant threads
- Process consists of multiple parallel threads
- Two threads can have same variable name, but allocated in two different private space
- If allocated in shared mem, all procs can read-/write
- OS decides which thread to run
- Fork/join parallelism



Pragma

Use pragmas to tell compiler infos about data and parallelism.

```
int main() {
  #pragma omp parallel {
    printf("Hello␣World\n");
  }
}
```

```
int main() {
  omp_parallel_start(&parfun, ...);
  parfun();
  omp_parallel_end();
}
int parfun(...)
{
  printf("Hello world\n");
}
```

- Use `shared()` and `private()` to indicate variables

- For loop can be executed in parallel using `#pragma omp for`

```
#pragma omp for schedule(static)
```

Static: Usefule for simple, regular loops with equal duration iterations.

```
#pragma omp for schedule(dynamic)
```

Dynamic: A thread is generated for a single iteration. Thds are moved to work queue, work is fetched from queue dynamically.

- Fine-grain parallelism
  - Good for load balancing
  - Small amounts of computational work between parallelism computaiton stages
  - Low computaiton to parallelization ratio, high overhead
- Coarse-grain parallelism
  - Harder to load balance efficiently, but
  - Large amounts of computational work between parallelism computation stages
  - High computation to parallelization ratio, low overhead

```
#pragma omp for schedule(dynamic, 2)
```

Reduces overhead by making bigger chunks (always queue 2 iterations on same core).

```
#pragma omp critical
```

Critical section: a portion of code that only one thread at a time may execute. Stops race concidions.

```
#pragma omp reduction(+:area)
```

Instructs the compiler to create private copies of `area` for every thread. At the end of loop, partial sums are combined on the shared variable.

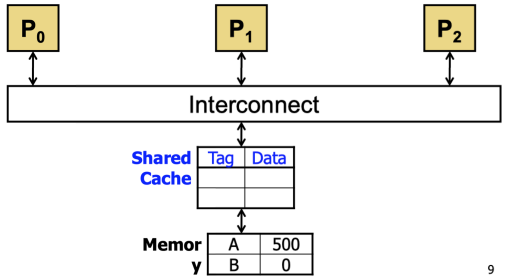
4.4 Threading & Shared Memory Programming Model

- Software thread
  - Per thread state: PC & registers
  - Shared state: global variables, heap
  - Different registers in memory
- OS manages threads
  - Thread scheduling and time multiplexing

Shared Memory

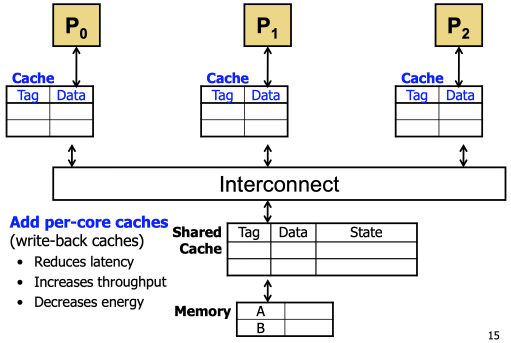
- Programmer explicitly creates multiple threads
- All ld, sd to a single share memory space
- A thread switch can occur at any time
  - Pre-emptive multithreading by OS
- Issues
  - Cache coherency
  - Synchronization
  - Memory consistency

Shared Cache



No cache coherency problems because only one cache. But shared cache becomes bottleneck.

Private Cache



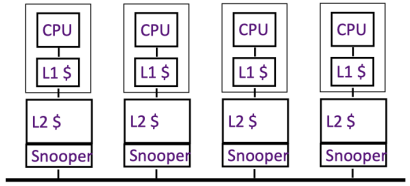
Faster access, lower power. Coherency problem: Snooping solves this.

Snooping

- Send all requests for data to all procs
- Procs snoop to see if they have a copy and respond accordingly
- Requires broadcast, since caching information is at processors
- Works well with bus
- Dominates for small scale machines
- Write invalidates all caches and main memory

Optimized Snoop with L2$

Snooping on L2 does not affect CPU-L1 bandwidth. Only small amount of invalidations is forwarded to L1: Cache lines included in L1 that need to be invalidated. Most is filtered at L2 level. Inclusion property: We have to make sure, that if an entry is in L1, it must bee in L2.



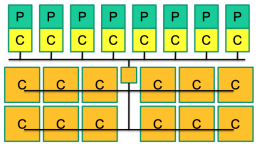
- Improves performance and complexity
- New interaction between L1 and L2 (Inclusion property)
- Reduces traffic of invalidates to L1

Write-back cache: Modified chache line is only written back to memory, if cache line is removed.

Write-through cache: Modified cache line is immediately written back to main memory.

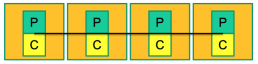
Multi-Core Shared Cache

Private L1$, shared L2$, bus between L1s and single L2 controller, snooping-based coherence between L1s. Todays CPUs have shared L3.



Multi-Core Cache

Private L1, shared but physically distributed L2. Bus connecting the four L1s and four L2 bankds. Snooping-based coherence between L1s.



False Sharing

A cache line contains more than one word. Cache-coherence is done at the line-level and not word-level. Suppose M_1 writes w_1 and M_2 writes w_2 and both have same line address, what can happen?

Performance of Symmetric Multiprocessors SMP

Cache performance is combination of

- Uniprocessor cache miss traffic
- Traffic caused by communication
  - Results in invalidations and subsequent cache misses
- Coherence misses
  - Also called communication miss
  - 4th C of cache misses along with Compulsory, Capacity & Conflict

Coherency miss

Also called communication miss

- True sharing misses areise from the communication of data through the cache coherence mechanism
- False shring misses when a line is invalidated because some word in the line, other than the one being read, is written into

Example: x1, x2 in same cache line. P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	True miss; x2 not writeable
5	Read x2		True miss; invalidate x2 in P1

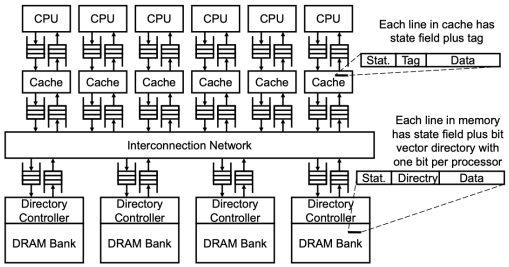
Directory Cache Protocol

Every memory line has associated directory information that

- Keeps track of copies of cached lines and their states
- On a miss, find directory entry, look it up and communicate only with the nodes that have copies if necessary
- In scalable networks, communicate with directory and copies is through network transactions

Many alternatives for organizing directory information.

Assumptions: Reliable network, Fifo message delivery between any given source-destination pair.



Direcotry bit is set to 1, if the selected CPU has line in its own cache. 1 bit per CPU. On write, if multiple bits are set, send invalidates **only** to CPUs that have the line in cache. Broadcast becomes multicast.

Each cache line has 4 possible states:
- C-invalid (Nothing): The accessed data is not resident in the cache
- C-shared (Sh): Accessed data is resident in the cache, and possibly also cached at other sites. Data in memory is valid
- C-modified (Ex): Data exclusiverly resident in this cache, and has been modified. Memory does not have most up-to-date data
- C-transient (Pending): Data is in a transient state (e.g., site has just issued a protocol request, but has not received the corresponding reply)

Each cache line has 4 possible directory states:
- R(dir): Mem line is shared by the sites specified in dir, data in mem is valid in this state. If dir is empty, memory line is not cached by any site
- W(id): Mem line is exlusively cached at site id, and has been modified at that site. Mem does not have the most up-to-date data
- TR(dir): The mem line is in a transient state, waiting for the ack to the invalidation req that the home site has issued
- TW(id): The mem line is in a transient state waiting for a line exclusively cached at site id (i.e., in C-modified state) to make mem line at home site up-to-date

Alternatives to cache coherence

Coherence keeps cahces "coherent", Load returns the most recent stored value by any processor and thus keeps caches transparent to software.

Alternatives:
- No caching of shared data (slow)
- Requiring software to explicitly "flush" data (hard to use), use some new instrucitons
- Message passing (programming witout shared memory), used in clusters of machines for high-performance computing

Directory-based coherence protocol scales well, perhaps to 1000s of cores.

4.5 Shared memory issue: Synchronization

How to regulate access to shared data? How to implement "locks"?

Problem

Thread level parallelism on shared data must be synchronized.

Synchronization

Regulate access to shared data (mutual exclusion). Low-level primitive: lock
- `acquire` and `release`
- Region between `acquire` and `release` is a critical section
- Must interleave `acquire` and `release`
- Interfering `acquire` will block

Strawman Lock (incorrect)

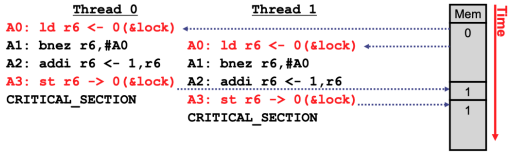
Spin lock: software lock implementation

```
acquire(lock): while (lock != 0) {}
    ↪ lock = 1;
```

In assembly:

```
A0: ld r6 <- 0(&lock)
A1: bnez r6,A0
A2: addi r6 <- 1,r6
A3: st r6 -> 0(&lock)
```

Problem: Can and will go wrong:



Correct Spin Lock

Compare and Swap: Single atomic operation, e.g. `cas r3 <- r1,r2,0(&lock)` performs atomically:

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

New acquire sequence:

```
A0: cas r3 <- 1,0,0(&lock)
A1: bnez r3,A0
```

Ensures that lock is held by at most one thread.

RICS implementation

CAS: Load+branch+store in one instruction is not very "RISC". Broken up into micro-ops: load-link and store-conditional pairs.

```
label:
  load-link r1 <- 0(&lock)
  // potentially other insns
  store-conditional r2 -> 0(&lock)
  vranch0not0zero label // check for
    ↪ failure
A1: bnez r3,A0
```

On `load-link`, processor remembers address, looks for writes by other processors, if write is detected, next `store-conditional` will fail, sets failure condition.

Used by ARM, PowerPC, MIPS, Inanium.

Fine-grain locks

Multiple locks, one per record. Fast: critical sections to different records, can proceed in parallel. But easy to make mistakes.

Deadlock: If two threads need mutliple locks, they acquire one lock and the other thread acquires the other lock, both threads are stuck in a deadlock.

```
Thread 0                          Thread 1

id_from = 241;                    id_from = 37;
id_to = 37;                       id_to = 241;

acquire(accts[241].lock);         acquire(accts[37].lock);
// wait to acquire lock 37        // wait to acquire lock 241
// waiting…                       // waiting…
// still waiting…                 // …
```

Coffman Conditions for Deadlock

4 necessary conditions: Break any one of these conditions to get deadlock freedom.
- Mutual exclusion
- Hold and wait
- No preemption
- Circular waiting

Lock order

Always acquire multiple locks in same order, yet another thing to keep in mind when programming.

```
Thread 0                          Thread 1

id_from = 241;                    id_from = 37;
id_to = 37;                       id_to = 241;
id_first = min(241,37)=37;        id_first = min(37,241)=37;
id_second = max(37,241)=241;      id_second = max(37,241)=241;

acquire(accts[37].lock);          // wait to acquire lock 37
acquire(accts[241].lock);         // waiting…
// do stuff                       // …
release(accts[241].lock);         // …
release(accts[37].lock);          // …
                                  acquire(accts[37].lock);
```

More problems..

What if:
- Some actions require 1 or 2 locks and others require all of them?
- There are locks for global variables? Then should operations grab this lock?

Research: Transactional memory (TM). Goals:
- Programming simplicity of coarse-grain locks
- Higher concurrency (parallelism) of fine-grain locks
- Lower overhead than lock acquisition

Idea: No locks, just shared data, optimistic (speculative) concurrency. Execute critical section speculatively, abort on conflicts. Detect conflicts via coherence protocol. "Better to ask for forgiveness than permission".

4.6 Memory consistency models

How to keep programmer sane while letting hardware optimize? How to reconcile shared memory with compiler optimiztions, store buffers, and out-of-order exec?

Process P1	Process P2
`A = 0;`	`B = 0;`
`...`	`...`
`A = 1;`	`B = 1;`
`L1: if (B==0) ...`	`L2: if (A==0) ...`

Observation: If write are immediately, it is impossible that both if-statements evaluate to true. But what if write invalidate is delayed?

Sequential Consistency (SC)

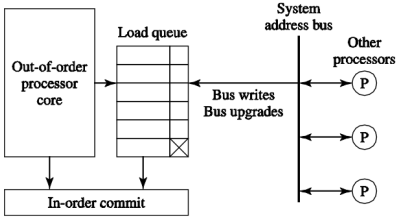
A multiprocessor is sequentially consistent, if the result of any execution is the same as if the (mem) ops of all procs were executed in seq. order. All procs see all ld and sd happening in same order.

Implementation:
- Delay completino of any mem access until all invalidations caused by that access are completed
- Delay next mem access until precious one is completed

Enforcing SC can be quite expensive. Solutions: Exploit latency-hiding techniques, employ relaxed consistency.

Hih-Performance SC

Load queue records all speculative loads. Bus write/upgrades are checked againts LQ. Any mathicn load gets marked for replay. At commit, loads are checked and replayed if necesary.



Relaxed concistency models

Key insight: only synchronizatino references need to be ordered. Hence, relax memory for all other references. Require programmer to label synchronization references. Ofte: Fence ops cause pipeline drain in moden OOO machine.

5 Vector Processors, Multithreading, Virtualizing

Performance beyond single thred instruction level parallelism: Data level parallelism: Perform identical operations on data, and lots of data.

5.1 Multithreading (MT)

Performing multiple threads of execution in parallel: Replicate registeres, PC, etc. Fast switching between threads. **Fine-Grain MT** switch threads after each cycle, interleave instruction execution, if one thread stalls, others are executed. **Coarsegran MT** only switch on long stall (eg. L2$-miss). Simplifies hardware, but doesn't hide short stalls.

Simultaneous MT (SMT)

In multiple-issue dynamically scheduled processor.

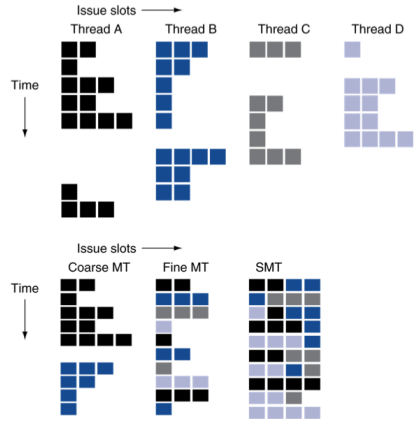
- Schedule insns from multiple thds
- Insn from independent thds execute when function units are available
- Within thds, dependencies handler by scheduling and reg nenaming

Example: Intel P4 HT. Two thds, duplicated registers, shared function units and caches.

Example MT

CMT: Switches between threads on each instruction causing the exec to be interleaved. Usually tone in a round-robin fashion. Pro: can hide short and long stalls. Con: Slows down exec of individual thds, since a thd rdy to exec wo stalls will be delayed by insns from other thds.

FMT: Switches thds only on costly stalls. Pro: Relieves need to have very fast thd switching. Con: Hard to overcome throughput losses from shorter stals, due to pipeline startup costs



SMT Design Challenges

SMT makes sense only with fine-grained implementation. Larger resgister file needed to hold multiple contexts. Ensure that cache and tLB conflicts generated by SMT do not degrade performance.

Future of Multithreading

Multiple simple cores might share resources more effectively.

5.2 Vector Processing

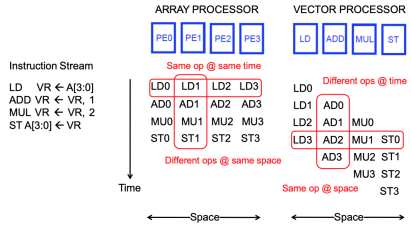
An alternative classification:

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

5.3 SIMD Processing

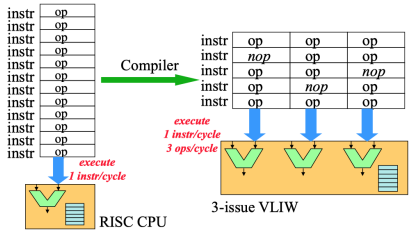
Single instruction operates on multiple data elements, in time or in space.

- Array processor: Insn operates on multiple data elements at the same time using different spaces
- Vector processor: Insn operates on multiple data elements in consecutive time steps using the same space



VLIW: vector HW without vector ISA

Multiple independent operations packed together by the compiler.



Problem: Register file needs $3n$ ports, 2 read and 1 write per issue slot.

Vector Processors

A vector is a 1D array of numbers. Many scientific/commervial programs use vectors. A vector processor is one whose insns operate on vectors rather than scalar values.

Basic requirements

- load/store vectors → vector registers
- Operate on vectors of different lengths → vector len reg `VLEN`
- Elements of a vector might be stored apart from each other in memory → vector stride register `VSTR`

Stride: distance in memory between two elements of a vector.

A vector insn performs an operation on each element in sonecutive cycles. Vectir functional units are pipelined, each pipeline stage operates on a different data element.

Vector instructions allow deeper pipelines

- Non intra-vector dependencies
- No control flow within a vector
- Known stride allows easy address calculation for all vector elements

Vector Processors Advantages

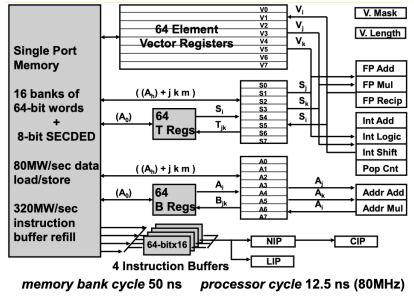
No dependencies within a vector, each insn generates a lot of work, highly regular memory access pattern, no need to explicitly code loops.

Vector Processors Disadvantages

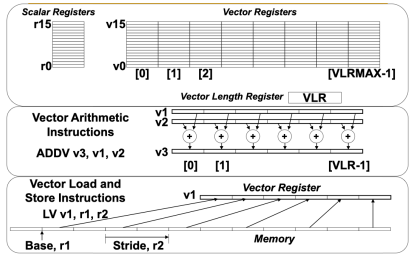
Works (only) if parallelism is regular (data/SIMD parallelism).

Cray-1

The first vector processor.



Vector programming model



Vector instrucion set advantages

- Compact
- Expressive, tells HW that theses N operations are
  - Independent
  - Use the same functional unit
  - ...
- Scalable

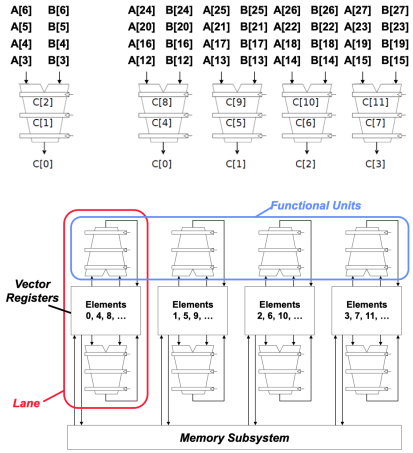
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

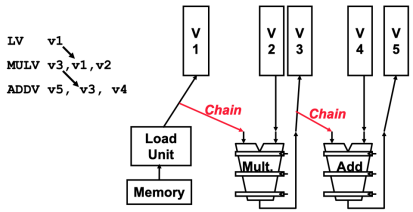
Vector Instruction Execution

By one heavily pipelined ALU or multple.



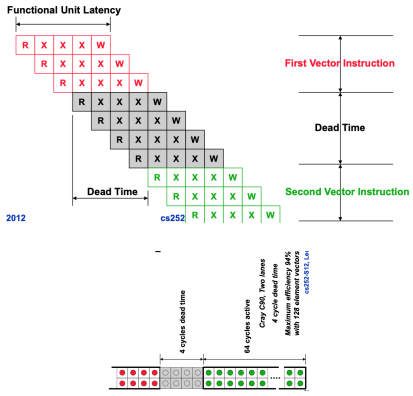
Vector Chaining

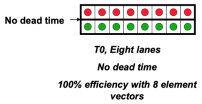
Vector version of register bypassing. Introduced with Cray-1. With chaining, can start dependent insn as soon as first result appears.



Vector Startup

Two components of vector startup penalty. Functional unit latency (time through pipeline), dead time or recovery time (time before another vector insn can start down pipeline).





Vector Stripmining

If large vectors don't fit in `VLEN`, do N operations with 64 elements and the remainder with x elements.

Automatic Code Vectorization

Vectorization is a massive compile-time reordering of operation sequencing and requires extensive loop dependence analysis.

Memory operations

Load/Store operations move groups of data between registers and memory. Three types of addressing:

- Unit stride
  - congiuous block of information in mem
  - Fastest: always possible to optimize
- Non-unit (constant) stride
  - Harder to optimize mem system for all possible strides
  - Prime number of data banks makes it easier to support different strides at full BW
- Indexed (gather-scatter)
  - Vector equivalent of register indirect
  - Good for sparse arrays of data
  - Increases number of programs that vectorize

Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
  a[i] = b[i] + c[d[i]]
```

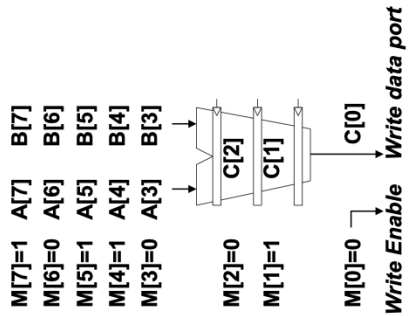
```
ld vD, rD # ld indices in D vect
lvi vC, rC, vD # ld indirect from rC
    ↪ base
lv vB, rB #ld B vect
ADDV.D vA, vB, vC # do add
sv vA, rA # store result
```

Vector Conditional Execution

Want ro vectorize loops with conditional code:

```
for (i=0; i<N; i++)
  if (a[i] > 0)
    a[i] = b[i];
```

Solution: Add vector mask (or flag) registers and maskable vector instructions.



Vector Reductions

Operations that reduce a vector to a scalar (e.g. sum of all elements).

```
sum = 0;
for (i=0; i<N; i++)
  sum += a[i];
```

Solution: Re-associate operations if possible, use binary tree to perform reduction.

```
sum[0:VL-1] = 0;
for (i=0; i<N; i+=VL)
  sum[0:VL-1] += a[i:i+VL-1];
// now have VL partial sums in 1 vec
    ↪ reg
do {
  VL = VL/2;
  sum[0:VL-1] += sum[VL:2*VL-1];
} while (VL>1);
```

Novel Matric Multiply Solution

Consider a matrix multiplication: 3 loops. Do need to do a bunch of reductions? No. Calculate multiple indep. sums within one vector register. Vectorize the second loop to perform 32 dot-products at the same time.


```
// multiply a[m][k] * b[k][n] to get
    ↪ c[m][n]
for (i=1; i<m; i++) {
  for (j=1; j<n; j+=32) {
    sum[0:31] = 0; // init vector
    for (t=1; t<k; t++) {
      a_scal = a[i][t]; // get scalar
      b_vec[0:31] = b[t][j:j+31]; //
    ↪ get vec
      // do vect-scal mul
      prod[0:31] = b_vec[0:31]*a_scal
    ↪ ;
      // Vector-vector add into res
      sum[0:31] += prod[0:31];
    }
    // unit stride store of vec of
    ↪ res
    c[i][j:j+31] = sum[0:31];
  }
}
```

5.4 Vector Processing in Modern ISAs

The rise of SIMD

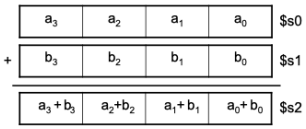
SIMD is good for applying identical computations across many data elements. It is energy efficient and tend to be bandwidth-efficient and latency-tolerant.

Multimedia (SIMD) Extensions

MMX follows a packed-SIMD execution model. A la array processing, yet much more limited. Other examples include ARM NEON, Intel MMX/SSE/AVX, RISC-V P (DSP) Extension.

- Single insn acs on multiple pieces of data at once

- Common application: graphics
- Perform short arithmetic operations (also packed arithmetic)



Packed-SIMD is no Vector-SIMD. Limited instruction set, limited vector register length, vector length is set in stone. Very short vectors, e.g., ARM NEON has a VLEN of 128 bits (Cray-1 in 1976 had 1024 bits). VLEN is encoded in the instruction itself. Widest extension to date: `Intel AVX-512`.

Renaissance of Vector Processing

Cray-like vector processing is currently expierencing a renaissance during the last few years. Main ISAs now include a vector processing extension: ARM SVE (scalabe vector extension) or RISC-V vector extension.

ARM SVE

No preferred vector length: VL is a hardware choice, 128-2048b in 128b increments. Vector length agnos-

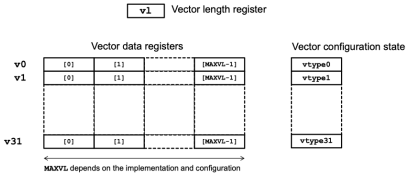
tic (VLA) programming model: Write once, compile once, vectorize more loops.

- Aarch64 (scalar)
  - Ten iterations over an 8-byte register
- NEON (128-bit vector engine)
  - Fout it. over a 16-byte reg + two it of a drain loop over a 8-byte reg
- SVE (VLA vector engine)
  - Three it over a 32-byte VLA register with an adjustable predicate

5.5 RISC-V Vector Extension

Being added as a standard extension to the RISC-V ISA. An updated form of Cray-style vectors for moden microporessors. Still a WIP.

- 32 vector registers, `v0-v31`
- Each vector reg has an associated type, including
  - Number of bits in each element
  - Representation (signed, unsigned, floating)
  - Shape (scalar, 1D)
- Number of vec regs and their type are configured with special insns before using the vector unit
- Vect insns are *polymorphic*, operation depends on the opcode and on the vector register operand types
- Vec len reg `v1` controls the num of elements executed by each insn
- Instructions can be *predicated* by a mask



Vector configuration

Can be done writing a number of control regs. Requested regs are zeroed. `MAXVL` depends on the number of vector regs and type. It should be possible to writea assembly code without knowing `MAXVL`.

```
setvl xdst, xsrc
```

`vl` is set to `min(MAXVL, xrc)` also copied to `xdst`.

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
```

Register `a0` holds `N`
Register `a1` holds `@A[0]`
Register `a2` holds `@B[0]`
Register `a3` holds `@C[0]`

```
    vcfg 3*32bINT # Set up v0, v1, v2 to be 32-bit ints
stripmined_loop:
    setvl t0, a0   # t0 holds amount done
    vld v0, 0(a1)  # Load strip of vector A
    vld v1, 0(a2)  # Load strip of vector B
    vadd v2,v0,v1  # Add vectors
    vst v2, 0(a3)  # Store strip of vector C
    slli t1,t0,2   # Multiply t0 by 4 to get bytes
    add a1,a1,t1   # Bump pointers
    add a2,a2,t1
    add a3,a3,t1
    sub a0,a0,t0   # Subtract amount done
    bnez a0, stripmined_loop
```

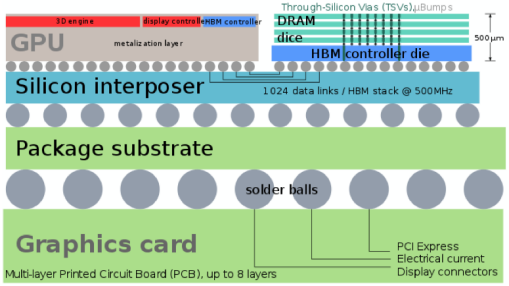
Important property: Minimal changes in data types, operation and length are also only minor changes in assembler code.

6 GP-GPUs

6.1 GPU Memory

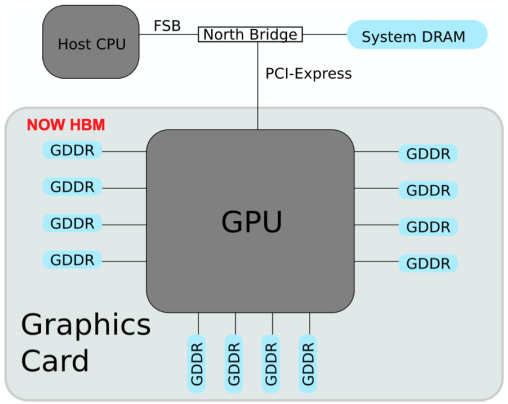
HBM2 GPU Memory

High bandwidth memory on same chip as GPU substrate. Mutliple DRAM dice on top of each other with vertical connections.



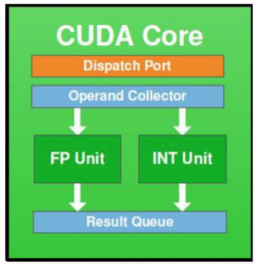
Access to system memory

If GPU wants to access more memory, it can do so using PCIe to system DRAM.



6.2 GPU Processing Cores

Basic element of GPU is a simple scalar core, consisting of floating point and integer ALU (onwards also tensor core). Receive data from very large register file.

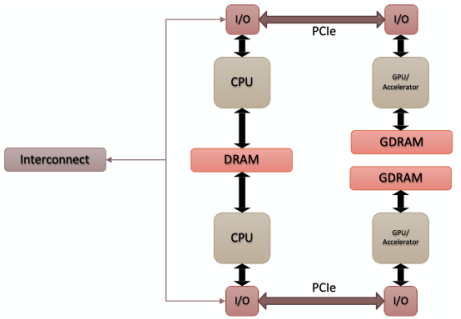


SM streaming multiprocessor houses a number of CUDA cores. (Tesla V100 has 64 FP32 cores / SM). Register file holds insns for all cores inside SM (shared among cores). Also share a dispatch and scheduler unit.

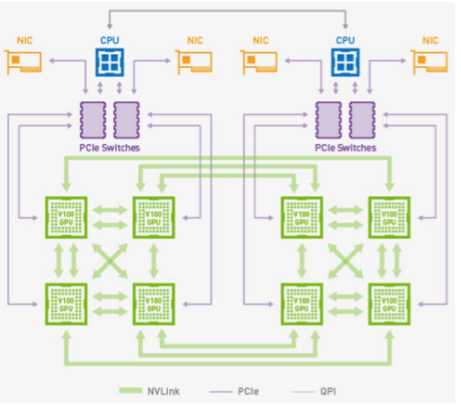
The tensor core performs $D = A \times B + C$ on 4×4 matrices. A, B are FP16, C, D FP16 or FP32.

6.3 Using the GPU

CPUs and Accelerators are used together. GPUs cannot be used instead of CPUs. GPUs perform compute heavy parts. Communication is via PCIe bus, up to 8GB/s throughput. Can also have multiple CPUs and GPUs in same share memory node.

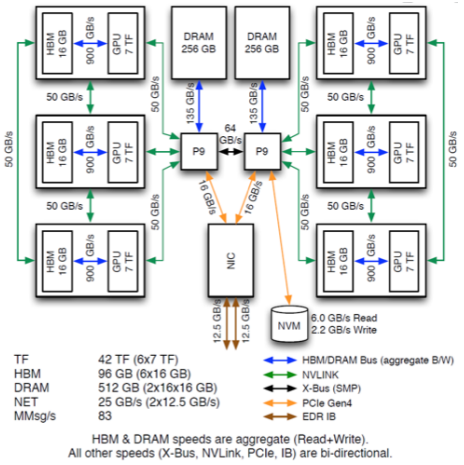


New GPU allos inter-GPU-connections.



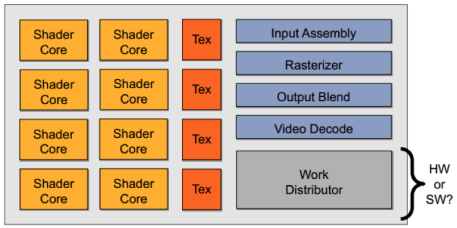
Summit supercomputer

Fully connected quad, 150Gb/s per GPU bidirectional for peer traffic. 50 GB/s per GPU bidirectional to CPU. Direct ls/sd access to CPU memory. High speed copy engines for bulk data movement.



6.4 Why GPU?

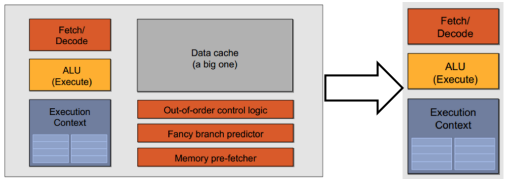
Graphics workloads are embarrassingly parallel (linear transformations are tensor operations). Shared cores are what cuda cores are today.



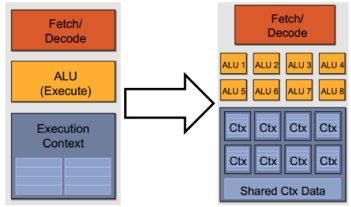
A single shader program is executed thousend times in parallel for different input data.

Slimming down shaders

First, remove components that help a single instruction stream run fast.

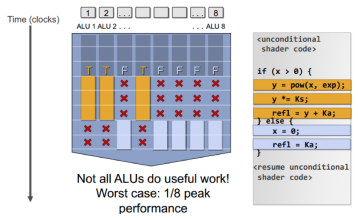


Second, amortize cost/complexity of managing an instrucion stream across many ALUs. SIMD processing, loose some flexibility.



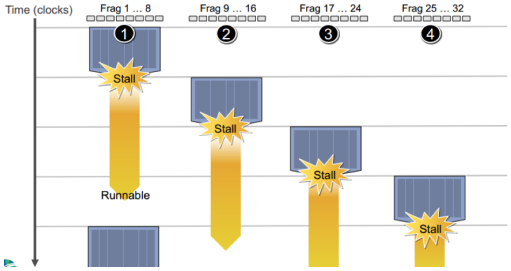
Branching in shaders

Since all ALUs share register file ld/sd, branching can cause stalls. Clarification: **SIMD processing does not imply SIMD instructions**. Option 1: Explicit vector insns (x86 SSe, AVX, Intel Larrabee). Option 2: Scalar insns, implicit HW vectorization: HW determines insn stream sharing across ALUs, NVIDIA GeForce (SIMT warps), ATI Radeon wavefronts.

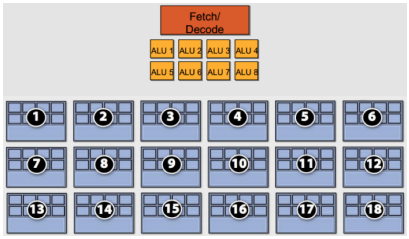


Memory stalls

Data are stored in textures. Access latency is 100-1000 cycles. We removed the cache that avoids stalls in 6.4. Idea: Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.



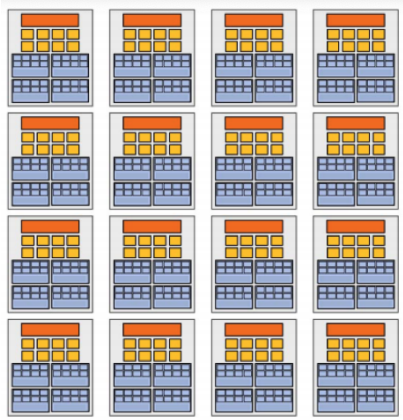
Latency can be hidden using latency hiding. More small contexts resul in maximal latency hiding, few large context in low latency hiding ability.



Clarification: Interleaving between contexts can be managed by HW or SW (or both). GPUs: HW schedules/manages all contexts (lots of them). Special on-chip storage holds fragment state. Intel Larrabee: HW manages four x86 (big) contexts at fine granularity, SW scheduling interleaves many groups of fragments on each HW context, L1-L2 cache holds fragment state.

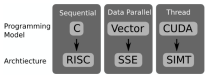
Example Chip

16 cores, 8 mul-add ALUs per core, 16 simultaneous insn streams, 64 concurrent (but interleaved) insn streams, 512 concurrent fragments: **256 FLOPs at 1GHz**.



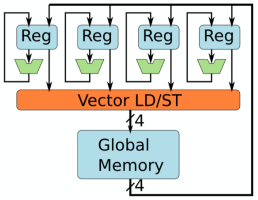
6.5 GP-GPU more in depth

Different programming models for different architectures.



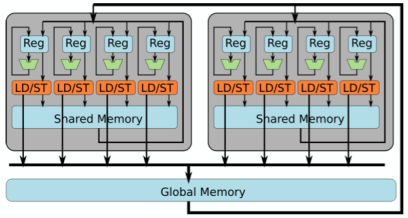
C and RISC

Most CPUs have vector SIMD units. Programmer's view of a vector SIMD:



GPU

CUDA programmer's view of GPUs: A GPU contains multiple SIMD units. All of them can access global memory. Important difference: GPUs use threads instead of vectors, GPUs have the shared memory spaces.

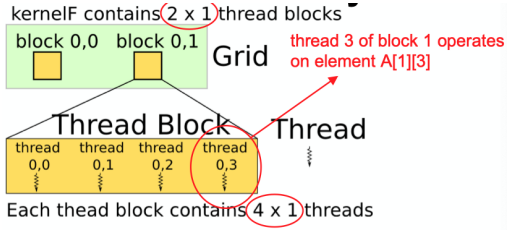


Thread hierarchy in CUDA: Grid contains thread blocks. Thread block contains threads.

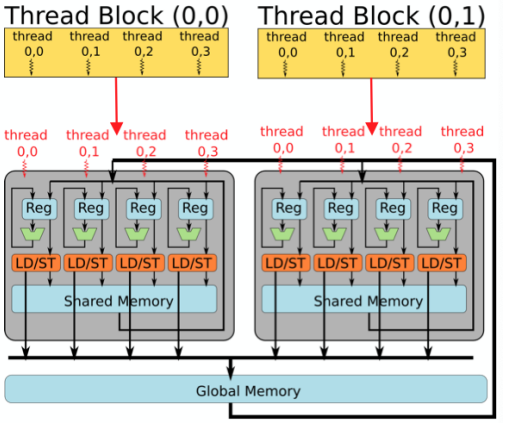
```
int A[2][4];
for(i=0;i<2;i++){
  for(j=0;j<4;j++){
    A[i][j]++;
  }
}
```

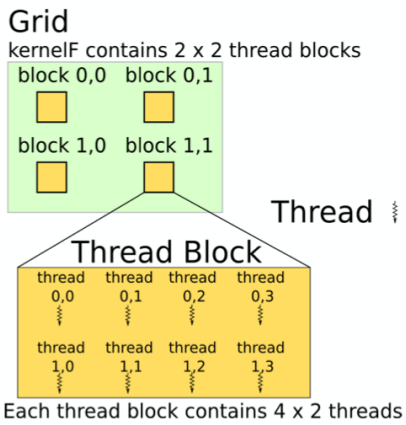
Converted to cuda:

```
int A[2][4];
// define threads
kernelF<<<(2,1),(4,1)>>>(A);
// all thds run same kernel
__device__ kernelF(A) {
  // each thd block has its id
  i = blockIdx.x;
  // each thd has its id
  j = threadIdx.x;
  // each thd has a different i,j
  A[i][j]++;
}
```



Each thread block will be scheduled on a different srteaming multiprocessor (SM).





GPU memory access

Example: convolution (3x3) win over (16x16) array. With a naiive implementation, each thread loads 9 elements from global memory. Utilizing shared memory: First load 16x16 to shared memory before computing filter in local memory. Each thread will load one element but erquires the elements that the neighboring threads have loaded. Need barrier before computation.

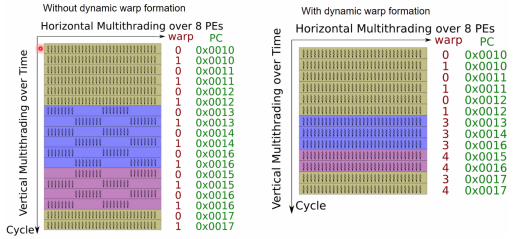
```
kernelF<<<(1,1),(16,16)>>>(A);
__device__ kernelF(A) {
  __shared__ smem[16][16];
  i = threadId.x;
  j = threadIdx.x;
  smem[i][j] = A[i][j];
  __sync(); // barrier
  A[i][j] = (smem[i-1][j-1] + ...) /
    ↪ 9;
}
```

Review

We write scalar code that is SIMT (single instruction multiple threads).

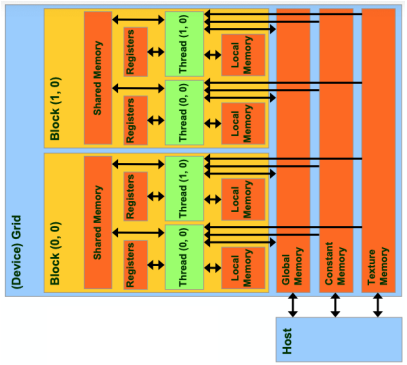
6.6 Warps and Branching

Programmer converts data level parallelism (DLP) into thread level parallelism (TLP). Hardware groupd threads into wraps, each warps holds so many threads the HW can execute in parallel = number of processing elements (PE). Switching warps on each cycle, altough not needed. Conditional part (blue, purple) gets interleaved. Create warp 3(4) from warp 0(0) and warp 1(1).



6.7 Memory Hierarchy

Cuda specific: RW per-thread: registers and local memory. RW per-block: shared memory. RE per-grid: global memory. RO per-grid constant memory and texture memory. The host can RW global, constant and texture memory.

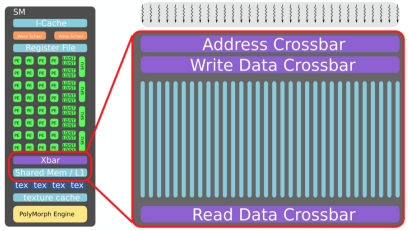


Single cycle: register and shared memory. DRAM, no cache: local and global memory (slow). DRAM, cached: constant and texture memory (1..100s of cycles)

Important Make sure to organize data structure, such that in a warp, accessed are made to data that is continuous in memory and properly aligned to get full bus bandwidth when accessing global memory. If not properly aligned, scattered transactions are required.

Shared Memory

On-chip on each SM. Order of magnitude lower latency and 10x higher bandwidth than global memory. Organized in multiple banks, one bank per thread. Each thread can access each bank. Can result in bank conflicts.



6.8 Atomics

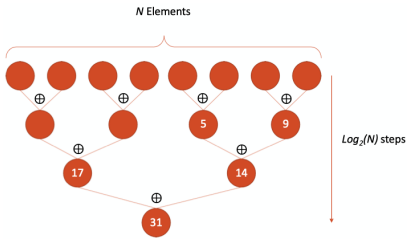
Race conditions can occur if multiple threads want to modify the same memory location at once. Atomics are used to ensure correctness when concurrently reading and writing to a memory location (global or shared).

```
__global__ void max_kernel(int *a)
{
  __shared__ int max;
  int my_local = a[threadIdx.x +
    ↪ blockIdx.x*blockDim.x];
  if (my_local > max)
    max = atomicMax(&max, my_local);
    ↪ // built-in
}
```

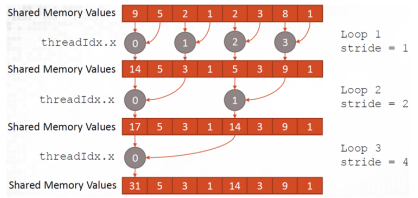
For the atomic instrucion access to the shared variable is serialised. Different ops are supported such as `atomicCAS` (compare and swap), `atomicInc`, `atomicAdd`, etc. .

6.9 Reductions

A reductino is where all elements of a set have a common binary associative operator (e.g. add) applied to them to "reduce" the set to a single value. Order in which ops is performed on set does not matter.

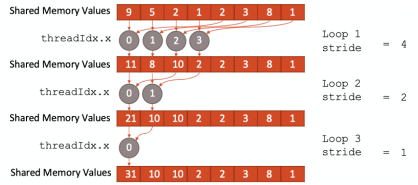


Each row could be a launch of several kernels but this is inefficient in memory access and launch overhead. A better way is **Block Level Recursive Reduction**:




```
for (int stride = 1; stride <
    ↪ blockDim.x; stride *=2) {
  unsigned int strided_i = threadIdx.
    ↪ x*2*stride;
  if(strided_i < blockDim.x) {
    sdata[strided_i] += sdata[
    ↪ strided_i+stride];
  }
  __syncthreads();
}
```

Even better if sequential addressing is used so memory banking can be exploited.



7 CPU + GPU computing

Basic flow: copy data to GPU, run kernel, copy back. Bandwidth between CPU and GPU is the bottleneck.

7.1 PCI as Memory Mapped IO

PCI device registers are mapped into the CPUs physical addrss space. Addresses are assigned to the PCI devices at boot time.

Links and Lanes

Each lane is 1-bit wide (4 wires, each 2-wire pair can transmit 2.5Gb/s in one direction). Each link can combine 1, 2, 4, .., 16 lanes: x1, x2, etc. Data is 8b/10b encoded: net data rate is 2 Gb/s per lane each way.

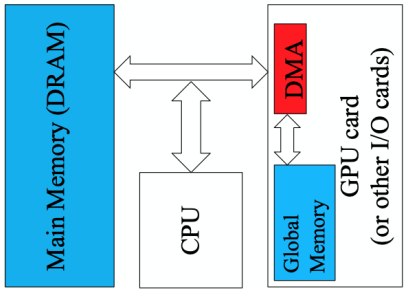
PCIe 3

8 Giga tranfers per second in each direction. No 8/10 but polynomials. 985MB/s per lane.

PCIe Data Transfer using DMA

DMA (Direct Memory Access) is used to fuly utilize the bandwidth of an IO bus. Needs pinned memory in DRAM: DMA uses physical addresses, the

OS could accidentally page out the data. Pinned memory cannot be paged out to swap.



NVLink

16 lamce PCIe3.0 has 16GB/s which is hardly adequate vs. 250GB/s of memory bandwidth available within a single card. NVLink is faster. Proprietary by NVIDIA.

7.2 Speeding Things Up

Simultaneously run a host-to-device transfer, a kernel and a device-to-host transfer. Requires buffers, concurrent kernels, two copy engines. `cudaMemcpyAsync` can do this.

Unified Virtual Adressing/Memory

"Smart pinned memory". Driver is allowed to cache memory on host or any GPU. Developer sees only one memory. OS keeps a page at fixed location (pinning). Directly access physical memory on host from GPU.

If memory is mapped to the GPU, migration can be triggered by access counters. with access counters migration only hot pages will be moved to the GPU.

8 Heterogeneous SoCs

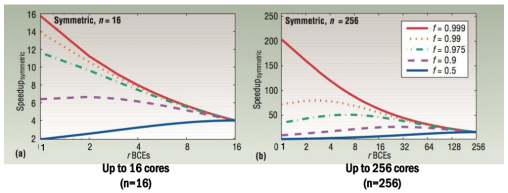
Amdahl's law:

$$\text{Speedup} = \frac{1}{(1-f)+\frac{f}{n}}$$

In terms of resource limits

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

f fraction of program that is parallelizable, n total processing resources, t resources dedicated to each processing core, each of the n/r cores has sequential performance. Assume $perf(1) = 1$.



Most "real world" applicatinos have complex workload characteristcs. Idea: the most efficient processor is a heterogeneous mixture of resources.

9 big.LITTLE

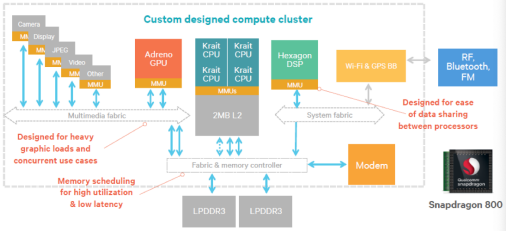
Aims at power reduction. Idea is taht power consumption of the chip could be reduced when high intensit tasks would run on suitable powerfull but power hungry cores, whereas low intensity tasks on less powerful and less power hungry cores.

10 CAPI

Coherent Accelerator Processor Interface. Provides a high-performance interface for the implementation of software-specific, computation-heavy algorithms based on FPGAs.

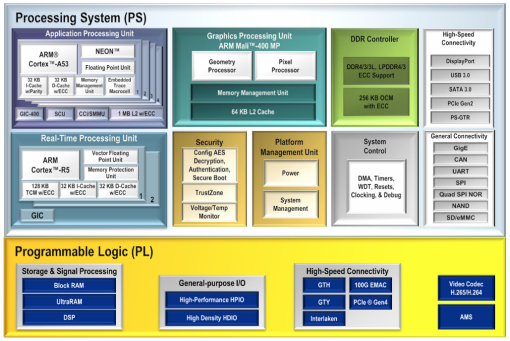
11 AMBA

AMBA is the interface used by ARM. Physical memory is shared amongs all cores.



12 CPU and FPGA

Zynq-7000 as example for a chip with CPU and FPGA. IO coherency with ACP and ACE: Whenever the accelerator does writes, these writes are captured by the cache logic and are invalidated. Asymmetric: Invalidates come from FPGA to CPU, not otherwise.



13 Sandbox

$$U = RI$$

•

	Single	Dual	Quad