LECTURE 2

PERFORMANCE METRICS AND VECTOR ARCHITECTURES

Miquel Pericàs
EDA284/DIT361 - 2019/2020 SP3

TODAY'S LECTURE

- **Metrics**
  - Performance and Efficiency
  - Review: Amdahl’s Law and Gustafson’s Law
- **Vector / SIMD**
  - Architecture
  - Memory Organization
  - Example: ARM SVE
- **Discussion of EDA284 project**

PERFORMANCE METRICS
How to claim that machine X is faster than Y?

- **METRIC #P1: Time to complete a task (T_{exe}): execution time, response time, latency**
  - “X is N times fast than Y” means Texe(Y)/Texe(X) = N (="Speed-up")
  - The major metric used in this course

- **METRIC #P2: number of tasks per day, hour, sec, ns**
  - The throughput for X Is N times higher than Y if THROUGHPUT(X)/THROUGHPUT(Y) = N
  - Not the same as Latency (E.g.: Multiprocessors, Pipelining)

Fundamental performance equations for CPUs:

Texe = IC x CPI x Tc

- **IC** ("Instruction Count"): depends on program, compiler and ISA
- **CPI** ("Cycles per Instruction"): depends on instruction mix, ISA, and implementation
- **Tc** (Clock Cycle): depends on implementation complexity and technology

CPI (CLOCK PER INSTRUCTION) IS OFTEN USED INSTEAD OF EXECUTION TIME

- **When processor executes more than one instruction per clock it is common to use IPC (instructions per clock) instead:**

Texe = (IC X Tc)/IPC

EFFICIENCY METRICS

- **METRIC #E1: Performance per Watt**
  - Application performance measure: FLOPS, MIPS, benchmark score, ...
  - Usually: average power while running the benchmark. Alternative: peak power, idle power.
    - Eg: Green500 ranking: Linpack FLOPS / average Watt during core phase
      - see: Power Measurement Methodology

- **METRIC #E2: Energy (J) of a computation**
  - Determines battery life
  - Related to metric #E1.

- **METRIC #E3: Energy-Delay Product**
  - E x D = (P x D) x D = P x D^2
  - Why/when is this useful?

- Other metrics?

WHICH PROGRAM TO CHOOSE?

- **A single program may execute efficiently on one machine but be inefficient on a different one.**
- **Need benchmark suites to cover different types of computation:**
  - SPEC: standard performance evaluation corporation
    - SPEC Cloud IaaS, SPEC CPU, SPEC ACCEL, SPEC MPI, SPEC OMP…
    - cloud/general purpose/scientific/engineering
    - SPEC CPU:
      - integer and floating point
      - new set every so many years (95,98,2000,2006,2017)
  - TPC benchmarks:
    - for commercial systems (transaction processing)
    - TPC-B, TPC-C, TPC-H, AND TPC-W
  - embedded benchmarks
  - media benchmarks

UNRELIABLE METRICS

EXAMPLES OF UNRELIABLE METRICS:

- MIPS: million of instructions per second
- MFLOPS: million of floating point operations per second
- Why are they unreliable?

EXECUTION TIME OF A PROGRAM IS THE ULTIMATE MEASURE OF PERFORMANCE BENCHMARKING

WHICH PROGRAM TO CHOOSE? OPTIONS

- **REAL PROGRAMS:**
  - porting problem; complexity; not easy to understand the cause of results
- **KERNELS**
  - computationally intensive piece of real program
- **TOY BENCHMARKS (e.g. quicksort, matrix multiply)**
- **SYNTHETIC BENCHMARKS (not real)**

REPORTING PERFORMANCE FOR A SET OF PROGRAMS

Let Ti be the execution time of program i:

1. (Weighted) Arithmetic mean of execution times:

$$\underbrace{\sum_i T_i / N}_{\text{N programs}} \quad \textbf{OR} \quad \sum_i T_i \times \underbrace{W_i}_{\text{weights}}$$

Problem: programs with longest execution times dominate the result!

2. Dealing with speed-ups

- Speedup measures the advantage of a machine over a reference machine 'R' for a program 'i'

$$S_i = \frac{T_{R,i}}{T_i}$$

- Reporting speed-ups for multiple programs
  - Arithmetic, Geometric or Harmonic mean?

REPORTING PERFORMANCE FOR A SET OF PROGRAMS

- **Geometric means of Speedups** $\bar{S} = \sqrt[N]{\prod_{i=1}^{N} S_i}$ Property $\mathrm{GM}\left(\frac{X_i}{Y_i}\right) = \frac{\mathrm{GM}(X_i)}{\mathrm{GM}(Y_i)}$
  - Geometric Mean Speedup comparisons between two machines are independent of the reference machine. Applies also to normalized mean execution times
  - Used to report SPEC numbers for integer and floating point

Mean of Execution Times

	Program A	Program B	Arithmetic Mean	Speedup (ref 1)	Speedup (ref 2)
Machine 1	10 sec	100 sec	55 sec	91.8	10
Machine 2	**1 sec**	200 sec	100.5 sec	50.2	5.5
Reference 1	100 sec	10000 sec	5050 sec		
Reference 2	100 sec	1000 sec	550 sec		

Performance of Program B (longer) dominates the results!

Means of SpeedUps

		Program A	Program B	Arithmetic	Harmonic	Geometric
Wrt Ref 1	Machine 1	10	100	55	18.2	31.6
	Machine 2	100	50	75	66.7	70.7
Wrt Ref 2	Machine 1	10	10	10	10	10
	Machine 2	100	5	52.5	9.5	22.4

conclusion: independent of reference machine!

Discussion on usage of GeoMean

- GM gives consistent results independent of reference[1], but does it mean it is correct?

- Multiplying execution times has no physical meaning[2], in contrast to adding times as in the arithmetic mean. Hard to understand intuitively.

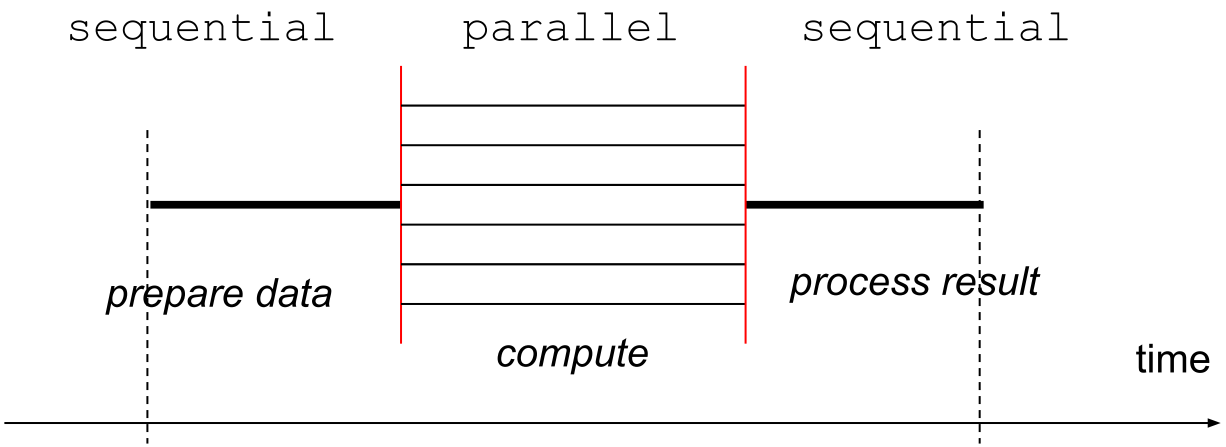
- Alternative: assign weights to each of the programs, calculate the average weighted execution time (using the arithmetic mean), and then normalize that result to one of the computers.

(1) Philip J. Fleming and John J. Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. Commun. ACM 29, 3 (March 1986), 218–221. DOI: https://doi.org/10.1145/5666.5673
(2) J. E. Smith. 1988. Characterizing computer performance with a single number. Commun. ACM 31, 10 (October 1988), 1202–1206. DOI: https://doi.org/10.1145/63039.63043

Amdahl's law and Gustafson's law (review)

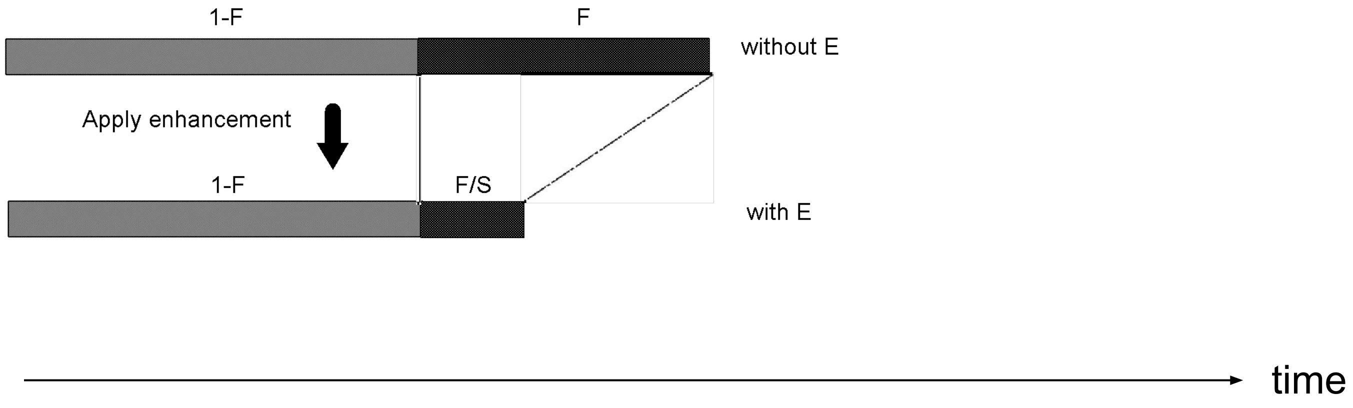
Assessing the limits of parallel performance

- <u>Example</u>: prepare a data set (sequential), perform a computation (parallel), and process the result (sequential)



- **What is the maximum performance improvement that can be achieved by improving the speed of the parallel computation?**

AMDAHL'S LAW (by Gene Amdahl, 1967)



- Enhancement ***E*** accelerates a fraction ***F*** of the task by a factor ***S***

$$T_{exe}(withE) = T_{exe}(withoutE) X \left[(1-F) + \frac{F}{S}\right]$$

$$Speedup(E) = \frac{T_{exe}(withoutE)}{T_{exe}(withE)} = \frac{1}{(1-F) + \frac{F}{S}}$$

- Basic assumption: total work to be done is constant

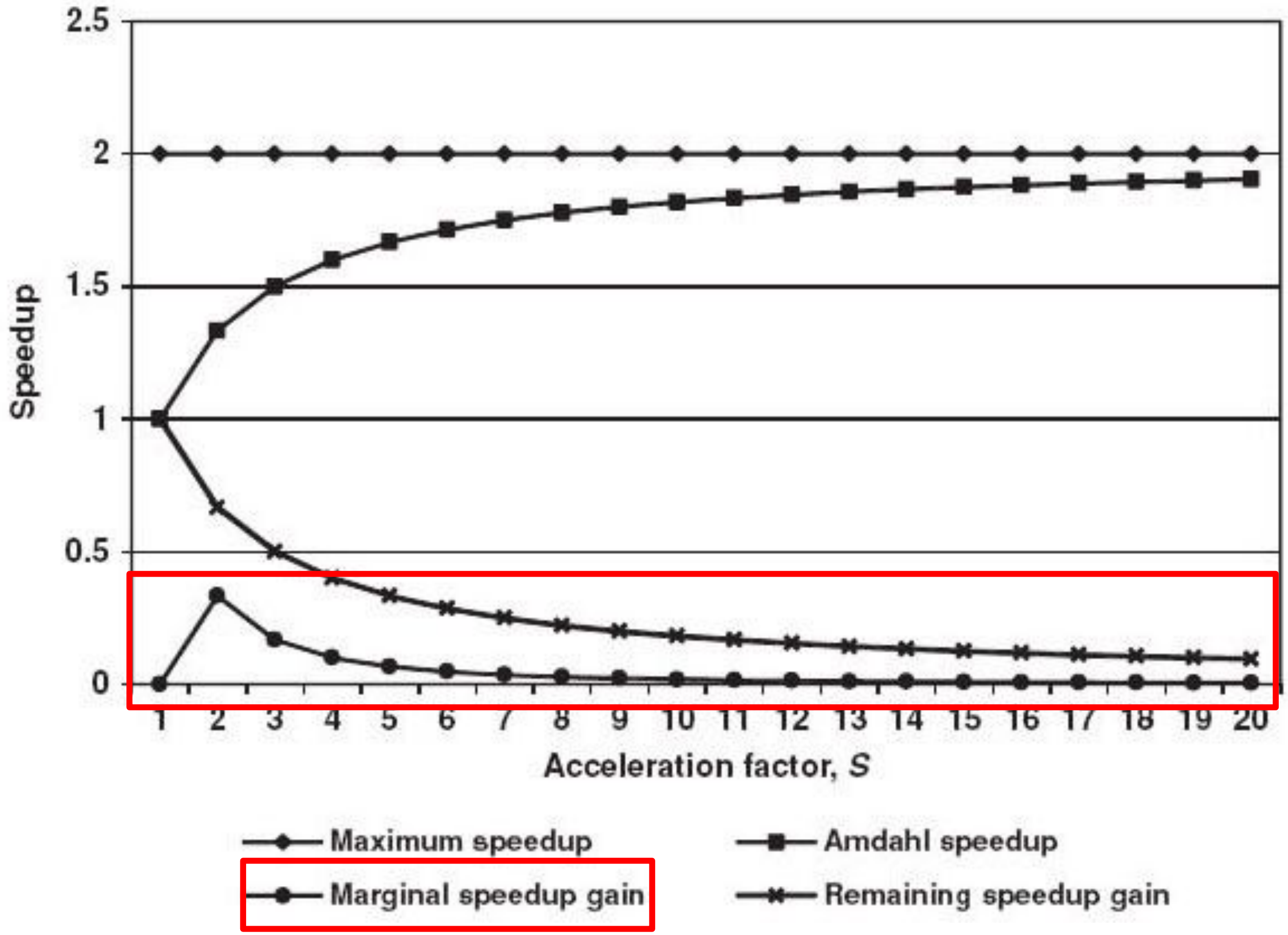
LESSONS FROM AMDAHL'S LAW

- Improvement is limited by the fraction of the execution time that cannot be enhanced. Assuming $S \rightarrow \infty$

- **Optimize the common case (e.g., with dedicated hardware)**
  - Execute the rare case in software (e.g., via exceptions)

- A moderate enhancement *S* already achieves most of the potential speed-up: **Law of diminishing returns**

LAW OF DIMINISHING RETURNS (F=0.5)



Further improvements to the same enhancement are unlikely to provide large benefits!

PARALLEL SPEEDUP

In a parallel system, the number of processors *P* becomes the enhancement factor *S*:

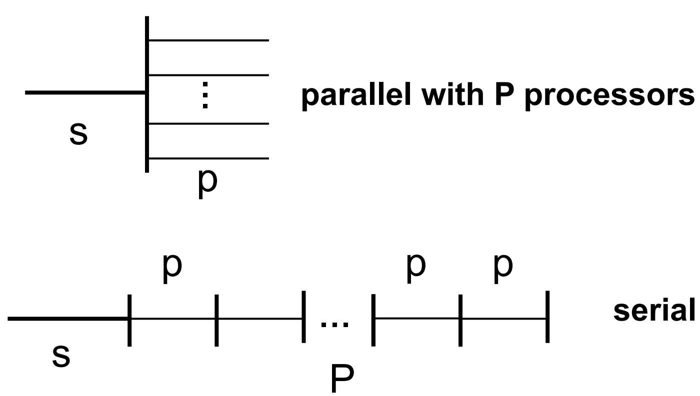
- Case S_p = P is called *linear* or *ideal* speed-up

OVERALL NOT VERY HOPEFUL:

Even with F=99% and 1M processors cannot achieve SpeedUp > 100

GUSTAFSON'S LAW (by John Gustafson, 1988)

- **Redefine Speedup**
  - Rationale: <u>Larger machines → larger challenges.</u> Assume now that fixed compute time, not fixed problem size
  - **T_p = s+p**; *s* is the time taken by the serial code and *p* is the time taken by the parallel code
  - **T_1 = s+pP**; exec time on one processor
  - Let **F=p/(s+p)**,
    - then **S_P = (s+pP)/(s+p) = 1-F+FP = 1+F(P-1)**

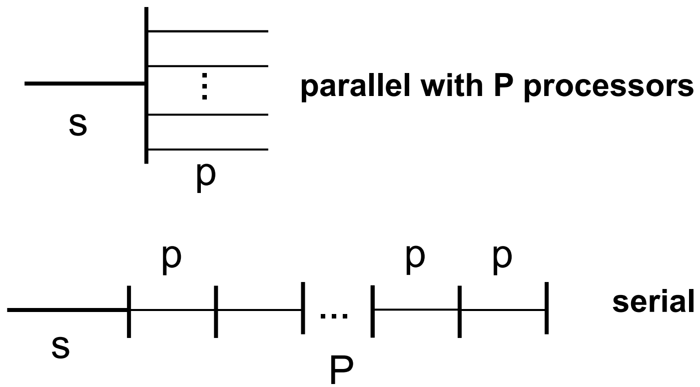


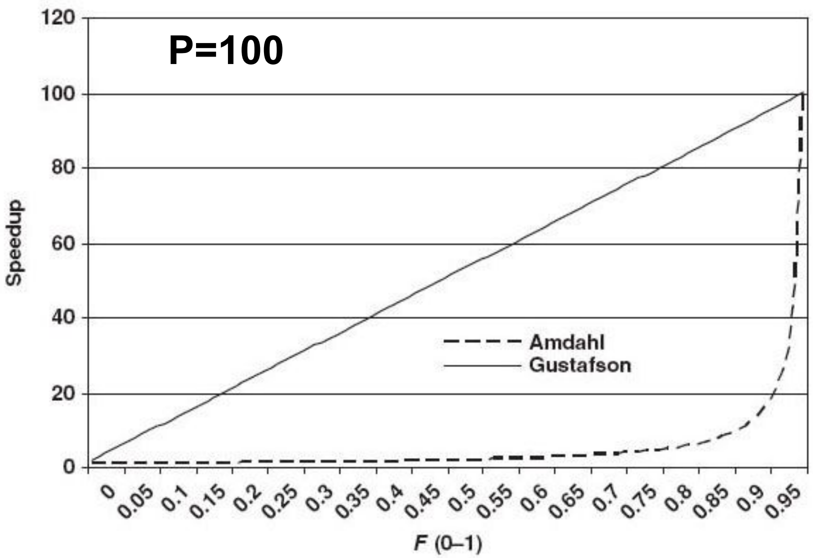
$T_p = s + p$

$T_1 = s + pP$

GUSTAFSON'S LAW (by John Gustafson, 1988)

- In other words, take T_p <u>constant time</u> → what is the largest problem size we can solve with P processors?
- Serial part must also be constant (independent of problem size)



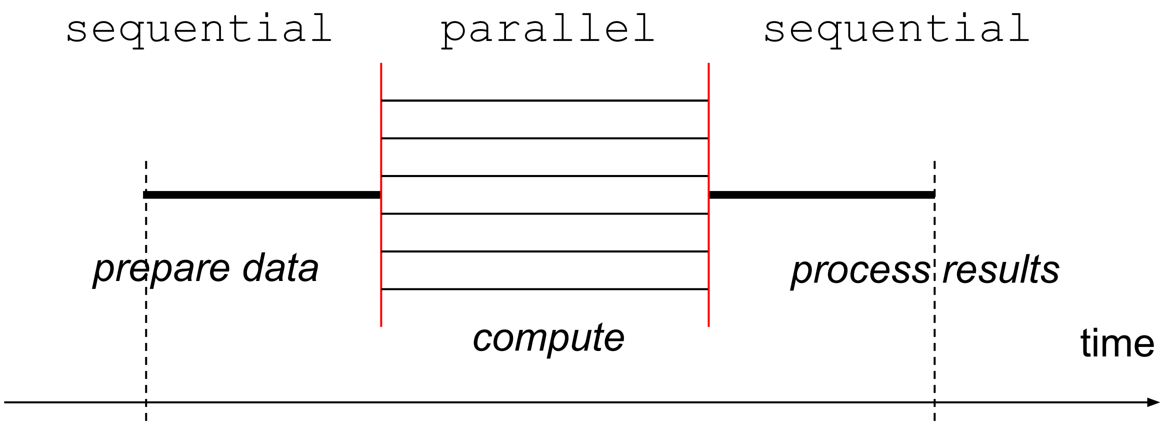


F (parallel fraction)

MORE OPTIMISTIC OUTLOOK ON THE USEFULNESS OF PARALLEL COMPUTERS

Exercise

- Let T_{seq}=10s be the time for the sequential part (prepare data + process results)
- Let T_{comp}=90s be the total time of the parallelizable computation (assume linear speed-up for the parallel part)



- What is the maximum speed-up according to Amdahl's law? What is the max speed-up with 9 processors?
- For the current problem size, we are satisfied with a total time of 20s. What is, according to Gustafson's law, the speed-up (wrt to the single processor system) for a 10x larger problem?

Amdahl’s and Gustafson’s law

- Amdahl's law and Gustafson's law assess the maximum speed-up in the context of fixed size problem (Amdahl) and constant time problem (Gustafson)
  - Provide expected impact of an architectural improvement
  - Provide guidance on where to focus the effort to improve the performance of a computer system

PARALLEL ARCHITECTURES

- **Types of parallel architectures**
  - Vector Processors: pipelining, array processing (Lecture 2, Today)
  - Scalar Processors: pipelining, superscalar, OoO (Lecture 3)
  - (Chip) Multiprocessors: thread parallelism (Lectures 4-5, 8)
- **Trade-offs**
  - Functional, performance and cost

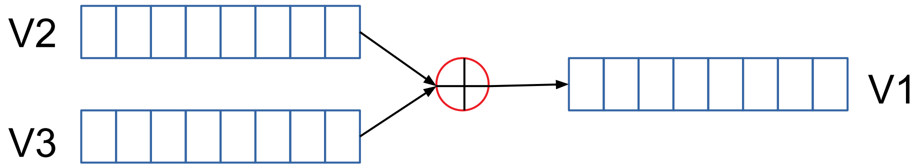
VECTOR PROCESSORS

- **Vector processors are able to execute instructions on entire vectors and not just on scalars**

- **Instructions are of the type**

  ADDV V1, V2, V3

- V1, V2, AND V3 are vectors of scalars of **same type** and **length**
- They are specified by a base address, a vector length and a stride (memory operand), or they can be vector registers
  - Executes V1[i] = V2[i] + V3[i], for all i's
- Vector length and stride may be held in special control registers



- **Need advanced compilers to automatically vectorize loops into vectors**
- <u>Alternative</u>: use compiler intrinsics or (inline) assembly

VECTOR PROCESSORS: HISTORY

- Vector Machines existed well before Superscalar processors
- Originally built for supercomputers for engineering/scientific computation
- Hugely popular during 1975-1990s, then overtaken by massively parallel computers ("killer micros" https://en.wikipedia.org/wiki/Killer_micro)
- Because of media (Streaming) application they are now also common in commodity markets (DSP) and also general purpose (e.g. SSE, AVX)
- Provide high efficiency: now getting popular again in the HPC domain

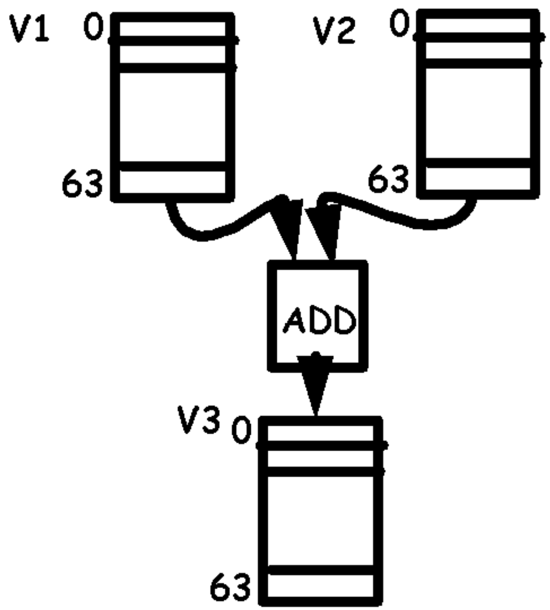
CRAY-1 (1975)

1974 CDC STAR-100

160 MFLOPS

Photograph courtesy of Charles Babbage Institute, University of Minnesota, Minneapolis

(PIPELINED) VECTOR ARCHITECTURE



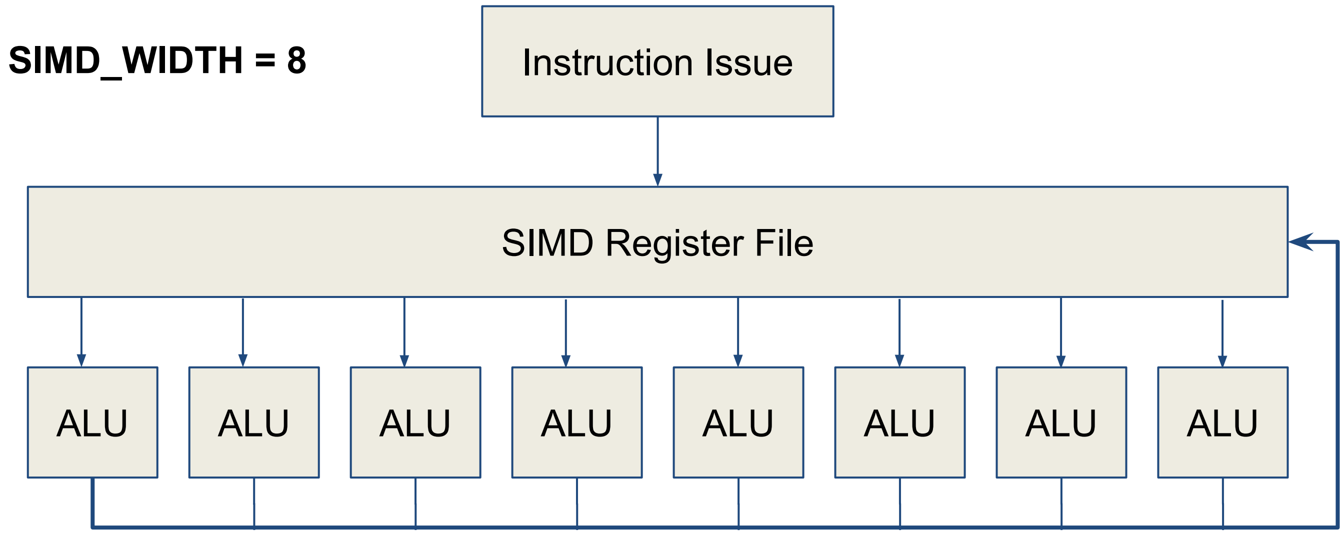
Assume ADD has 10 stages
The total execution time is:
$T_{ex} = 10 + 63 = 9 + 64$
In general: $T_{ex} = T_{start} + N$

- After vector startup time (the time to get the first result), results are computed one per clock

Tvector = VECTOR_LENGTH + STARTUP TIME

SIMD Architecture

- Modern SIMD computers process all elements of the vector simultaneously
- SIMD_WIDTH is usually smaller than VECTOR_LENGTH, e.g. 512-bit in AVX-512
- Latency equal to equivalent scalar operation



- It is also possible to combine the pipeline approach with the parallel (SIMD) approach (similar to superscalar execution), either via SW (loop) or HW (pipeline)

Tvector = VECTOR_LENGTH/SIMD_WIDTH + STARTUP TIME

VECTOR - MEMORY SYSTEM

- **LOAD/STORE instructions are of the form**

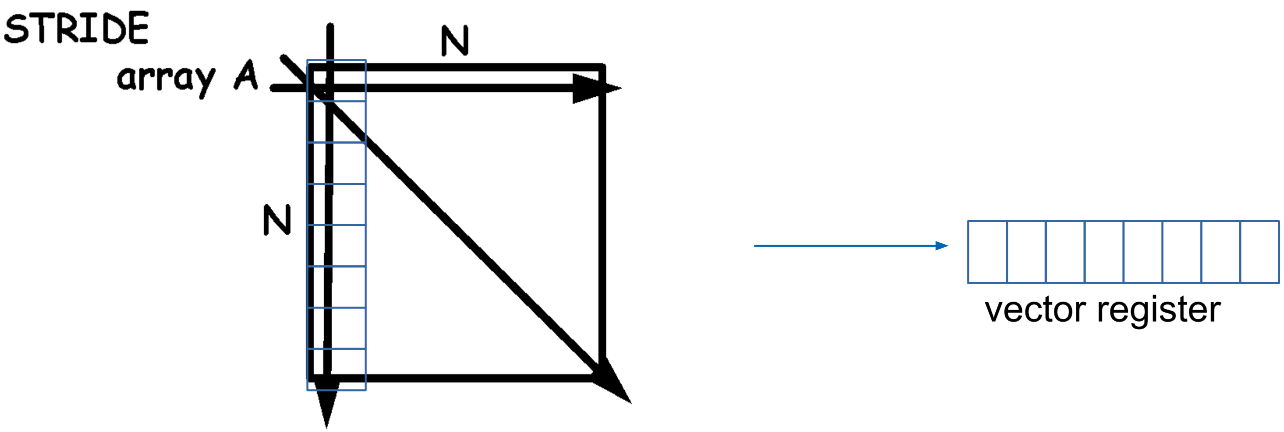
  L.V V1, R1, R2 // load V1 from Mem[R1+(0,R2,2xR2,…)] R2: STRIDE

STRIDE := number of locations in memory between beginnings of successive array elements, measured in bytes or in units of the size of the array's elements

- **Access pattern to memory is known at decode time for the entire vectors**
  - all accesses involved in a memory vector operation can be efficiently scheduled right after instruction decode.
  - memory is interleaved. No need for caches, particularly for long vectors (why?)
  - vector load/store units from memory to registers

VECTOR - MEMORY SYSTEM

- **Load store units can be seen as pipelines**
  - the startup time is the time to get the first component
  - startup time is much longer than for functional units
  - vectors accessed with a stride are stored in consecutive locations of vector register
- **Efficient accesses to matrices are important**

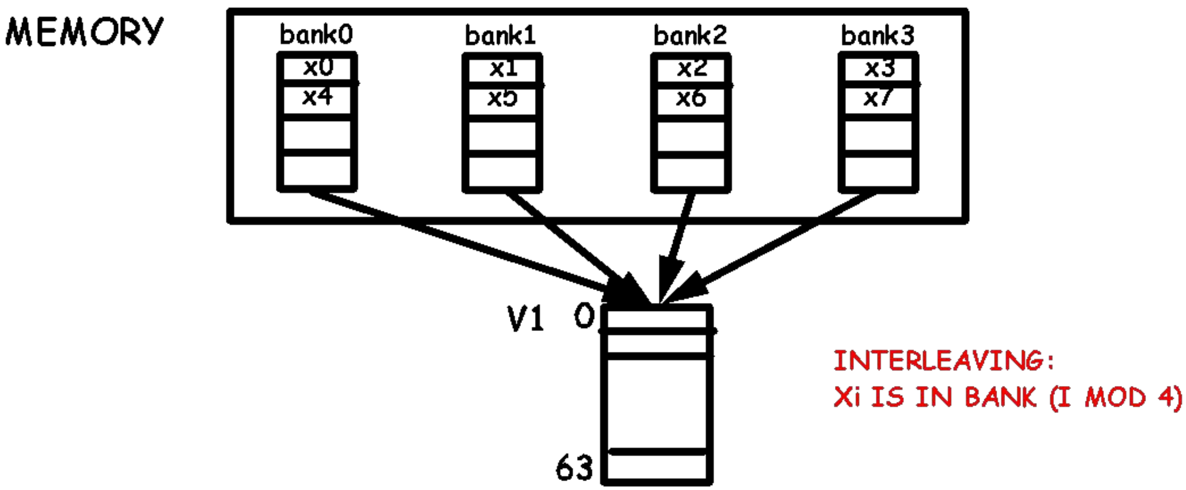


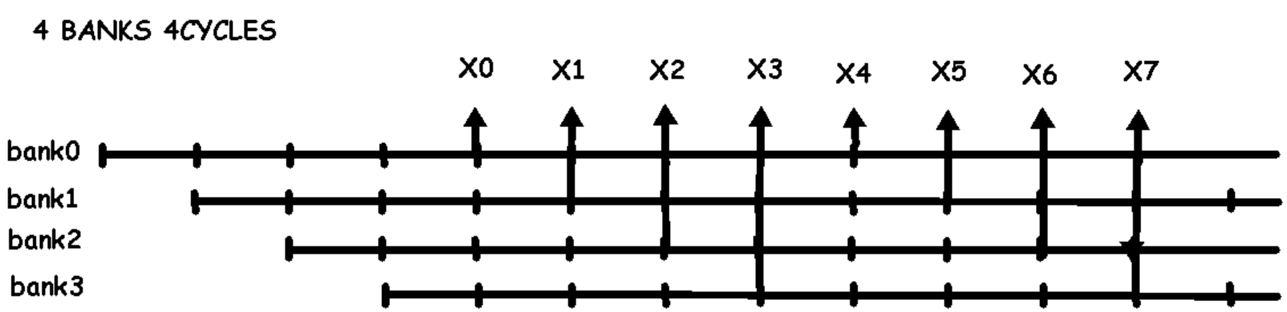
STRIDEs:

- Rows: stride 1, Columns: stride N
- Forward diagonal: stride N+1, Backward diagonal: stride N-1

VECTOR - MEMORY ORGANIZATION

- **Heavily interleaved (potentially hundreds of memory modules)**
- Banks are started one after the other
- if the number of banks is greater that the memory cycle time, we have no conflicts, and results come out one per clock





T_{load} **= VECTOR_LENGTH + STARTUP TIME (TIMETOGETX0 -1)**

MEMORY SCATTER GATHER

- **SCATTER/GATHER**
  - many scientific computations use sparse matrices
    - Most components are 0
    - But the pattern is not regular
    - Compress a sparse matrix into vectors of non-zero elements A[1:1000] ==> K[1:9], A*[1:9]
    - A*: Nonzero elements of A; K: Indices of nonzero elements of A
  - GATHER operation implements following function:

```
for(i=0;i<N,i++)
        A[i] = B[INDEX[i]];
```

B[ ]= $\begin{pmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{pmatrix}$

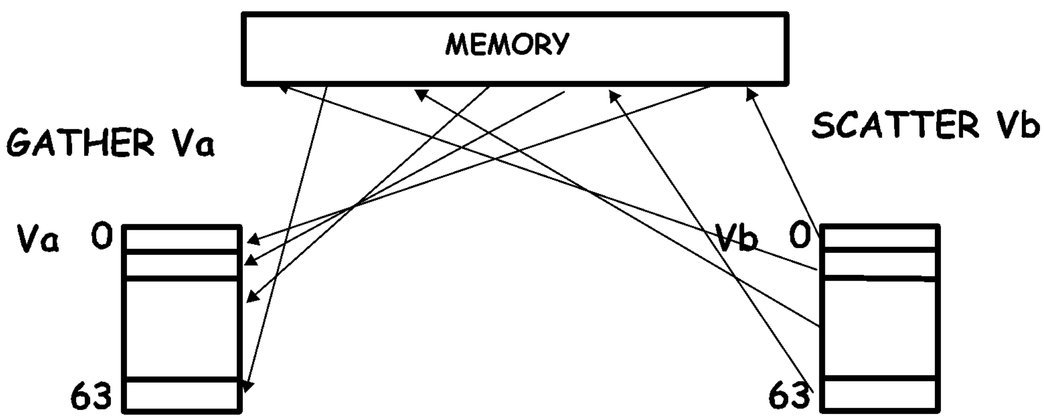
A_INDEX	Value (A)
0	10
1	20
7	30
9	40
14	50
15	60
16	70
23	80

- SCATTER operation implements corresponding functionality for stores

MEMORY SCATTER/GATHER

- **Memory locations of vector components are spread out in memory**
- **Use scatter and gather instructions**
  - **Gather** is a load instruction loading the components at indexed addresses into consecutive v-register locations
  - **Scatter** is a store instruction from v-register to indexed addresses

```
L.V    Vk,0(R1),R6          //load vector K in REGISTER Vk
LI.V   Va,Vk,0(R2)          //load indexed from A(K(i)): gather
           <work on Va, put result in Vb>
SI.V   0(R2), Vk, Vb        //store indexed to A(K(i)): scatter
```



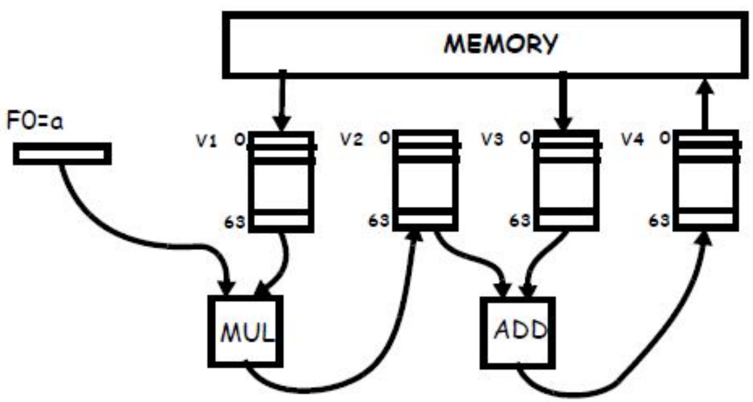
CHAINING AND PARALLEL EXECUTION OF INDEPENDENT VECTOR INSTRUCTIONS

- **CONSIDER THE CODE:**

  Y = a * X + Y   // “AXPY” in BLAS library

- **EXECUTION TIME (ONE OP AT A TIME):** startup(load) + vector_length + startup(multv) + vector_length + startup(load) + vector length+ startup(addv) + vector_length + startup(store) + vector_length = (startup + vector_length)x5

- **EXECUTION TIME (CHAINING+PARALLEL):** startup(load) + startup(multv) + startup(addv) + startup(store) + vector_length (the two loads execute in parallel)




```
L.D      F0,0(R4),R6
L.V      V1,0(R1),R6
MUL.V    V2, V1, F0
L.V      V3,0(R2),R6
ADD.V    V4, V2, V3
S.V      0(R2),V4,R6
```

SUMMARY ON VECTOR PROCESSORS

- **Design complexity?**
  - explicit parallelism enables high performance with simple hardware
- **Compiler?**
  - need advanced technology to find vectorizable loops
- **Memory subsystem?**
  - no need for caches if long vectors
- **Applications?**
  - need long vectors for high performance!

- **Vector architectures are highly efficient. Currently experiencing a rebirth of vector technology for high performance computing:**
  - INTEL AVX-512: Short/medium, fixed-length vectors (512 bits, SIMD)
  - ARM SVE: Long, variable-length vectors (128-2048 bits)
  - NEC SX-10+: Long, fixed-length vectors (16384 bits)

ARM-SVE: an example of a modern vector architecture

ARM SVE - Scalable Vector Extension

The following slides are taken from the ARM tutorial: "Vector Architecture Exploration with gem5" given at ICS'18 in Beijing, June 12th 2018, by Alex Rico, Jose Joao and Giacomo Gabrielli

Scalable Vector Extension – SVE

Significantly extends vector processing capabilities of AArch64

Enables implementation choices of vector lengths – 128 to 2048 bits

- *Vector Length Agnostic* (VLA) programming adjusts dynamically to the available VL
- No need to recompile, or to rewrite hand-coded SVE assembler or C intrinsics

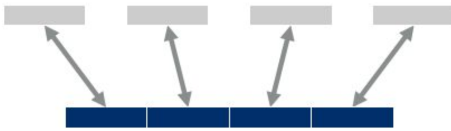
Focus is HPC scientific workloads and machine learning, not media/image processing

Will enable advanced vectorizing compilers to extract more fine-grain parallelism from existing code and so reduce software deployment effort

arm Research

Introducing the Scalable Vector Extension (SVE)

A vector extension to the ARMv8-A architecture with some major new features:



Gather-load and scatter-store

Loads a single register from several non-contiguous memory locations.

	1	2	3	4
+	5	5	5	5
pred	1	0	1	0
=	6	2	8	4

Per-lane predication

Operations work on individual lanes under control of a predicate register.

```
for (i = 0; i < n; ++i)
```

INDEX i	n-2	n-1	n	n+1
CMPLT n	1	1	0	0

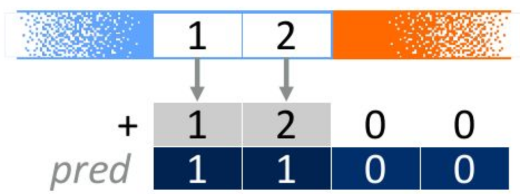
Predicate-driven loop control and management

Eliminate scalar loop heads and tails by processing partial vectors.

arm Research

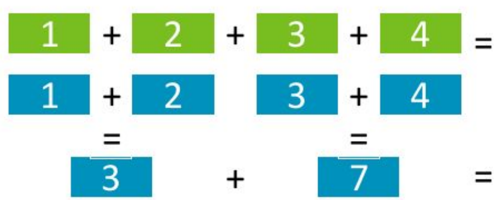
Introducing the Scalable Vector Extension (SVE)

A vector extension to the ARMv8-A architecture with some major new features:



Vector partitioning and software-managed speculation

First Faulting Load instructions allow memory accesses to cross into invalid pages.



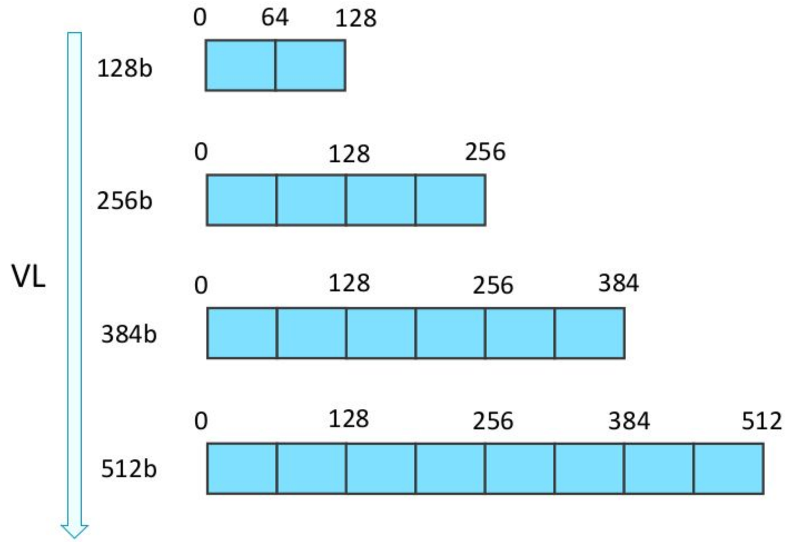
Extended floating-point horizontal reductions

In-order and tree-based reductions trade-off performance and repeatability.

What's the vector length?

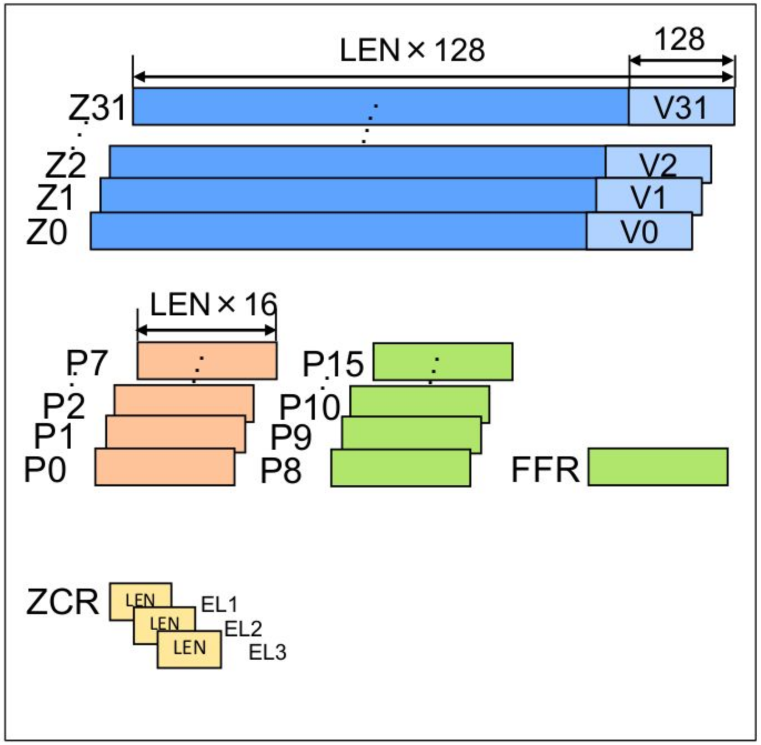
There is **no** preferred vector length

- Vector Length (VL) is the CPU implementor's choice, from 128 to 2048 bits, in increments of 128
- Adopting a Vector Length Agnostic (VLA) code generation style makes code portable across all possible vector lengths
- VLA is made possible by the per-lane predication, predicate-driven loop control, vector partitioning and software-managed speculation features of SVE
- No need to recompile, or to rewrite hand-coded SVE assembler or C intrinsics



SVE – architectural state

- Scalable vector registers
  - Z0-Z31 extending NEON's V0-V31
    - DP & SP floating-point
    - 64, 32, 16 & 8-bit integer
- Scalable predicate registers
  - P0-P7 lane masks for ld/st/arith
  - P8-P15 for predicate manipulation
  - FFR *first fault register*
- Scalable vector control registers
  - ZCR_ELx vector length (LEN=1..16)
  - Exception / privilege level EL1 to EL3



arm Research

daxpy (scalar)

```c
void daxpy(double *x, double *y, double a, int n)
{
   for (int i = 0; i < n; i++) {
      y[i] = a * x[i] + y[i];
   }
}
```

```
// x0 = &x[0]
// x1 = &y[0]
// x2 = &a
// x3 = &n
daxpy_:
        ldrsw       x3, [x3]
        mov         x4, #0
        ldr         d0, [x2]
        b           .latch
.loop:
        ldr         d1, [x0, x4, lsl #3]
        ldr         d2, [x1, x4, lsl #3]
        fmadd       d2, d1, d0, d2
        str         d2, [x1, x4, lsl #3]
        add         x4, x4, #1
.latch:
        cmp         x4, x3
        b.lt        .loop
        ret
```


arm Research

DAXPY

```
void daxpy(double *x, double *y, double a, int n)
{
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

daxpy (SVE)

```
daxpy_:
        ldrsw   x3, [x3]
        mov             x4, #0
        whilelt p0.d, x4, x3
        ld1rd   z0.d, p0/z, [x2]
.loop:
        ld1d            z1.d, p0/z, [x0, x4, lsl #3]
        ld1d            z2.d, p0/z, [x1, x4, lsl #3]
        fmla            z2.d, p0/m, z1.d, z0.d
        st1d            z2.d, p0, [x1, x4, lsl #3]
        incd    x4
.latch:
        whilelt p0.d, x4, x3
        b.first .loop
        ret
```

daxpy (scalar)

```
daxpy_:
        ldrsw           x3, [x3]
        mov             x4, #0
        ldr             d0, [x2]
        b               .latch
.loop:
        ldr             d1, [x0, x4, lsl #3]
        ldr             d2, [x1, x4, lsl #3]
        fmadd           d2, d1, d0, d2
        str             d2, [x1, x4, lsl #3]
        add             x4, x4, #1
.latch:
        cmp             x4, x3
        b.lt            .loop
        ret
```

- `whilelt p0.s, x4, x3` fills predicate register p0 by setting each lane as `p0.s[idx] := (x3 + idx) < x4` (x3 and x4 hold i and N respectively), for each of the indexes `idx` corresponding to 32-bit lanes of a vector register
- `incd`: increment the index `x4` by as many words as a vector can store with `incd`
- `b.first`: The first active element is true.

arm Research

SUMMARY

- **Metrics**
  - Performance and Efficiency
  - Review: Amdahl's Law and Gustafson's Law
- **Vector / SIMD**
  - Architecture
  - Memory Organization
  - Examples: ARM SVE
- **Discussion of EDA284 project**

Next Lecture

- Next Lecture is tomorrow (Jan 28th)
- Topic: Out-of-Order Execution + Multilevel Cache Hierarchy (mostly review of DAT105)
- Note: There will be no lecture this Friday.
- Final topic for today: EDA284 project