Data Level Parallelism

Increasing single-core performance

Introduction

- Yesterday, we saw how to speed up a pipeline
  - Improving the memory stage (caches)
  - Increasing instruction-level parallelism
  - Using software pipelining
  - Using loop unrolling

  } Instruction-centric overview of the performance

- But instructions are just a way to transform code into operations

- And what is code useful for? → Transform data
  - What if we look at the pipeline from a data-centric point of view?

Loops are quite inefficient

Given this sum of two arrays…

```
for(int i=0; i<100; ++i){
    C[i] = A[i] + B[i];
}
```

… The assembly ends up looking like this:

```
a0 ← &A[0]
a1 ← &B[0]
a2 ← &C[0]
mv s1 ← 100-1

loop:
load t0, 0(a0)
load t1, 0(a1)
add t2, t0, t1
store t2, 0(a2)
addi a0, a0, 4
addi a1, a1, 4
addi a2, a2, 4
addi s1, s1, -1
bnez s1, loop
```

Four “useful” instructions (load/compute data)

Five “loop control” instructions

55% (5/9) of the instructions are “loop overhead” !

Let's see it differently

We are applying the same operation to contiguous data elements…

It #1

```
load t0, 0(a0)
load t1, 0(a1)
add t2, t0, t1
store t2, 0(a2)
addi a0, a0, 4
addi a1, a1, 4
addi a2, a2, 4
addi s1, s1, -1
bnez s1, loop
```

A[0], B[0], C[0]

It #2

```
load t0, 0(a0)
load t1, 0(a1)
add t2, t0, t1
store t2, 0(a2)
addi a0, a0, 4
addi a1, a1, 4
addi a2, a2, 4
addi s1, s1, -1
bnez s1, loop
```

A[1], B[1], C[1]

…

Reorder

```
load t0, 0(a0)
addi a0, a0, 4
load t3, 0(a0)
```

Load A[0:1]

```
load t1, 0(a1)
addi a1, a1, 4
load t4, 4(a1)
```

Load B[0:1]

```
add t2, t0, t1
add t5, t2, t4
```

Add [0:1]

```
store t2, 0(a2)
addi a2, a2, 4
store t5, 4(a2)
```

Store C[0:1]

```
addi a0, a0, 4
addi a1, a1, 4
addi a2, a2, 4
addi s1, s1, -2
bnez s1, loop
```

Same instructions, still no benefit

Can we do this better?

- We can use **SIMD**: Single Instruction Multiple Data

```
load t0, 0(a0)
addi a0, a0, 4
load t3, 0(a0)
```
} `simd_load s0, 0(a0)`

```
load t1, 0(a1)
addi a1, a1, 4
load t4, 4(a1)
```
} `simd_load s1, 0(a1)`

```
add t2, t0, t1
add t5, t2, t4
```
} `simd_add s2, s0, s1`

```
store t2, 0(a2)
addi a2, a2, 4
store t5, 4(a2)
```
} `simd_store s2, 0(a2)`

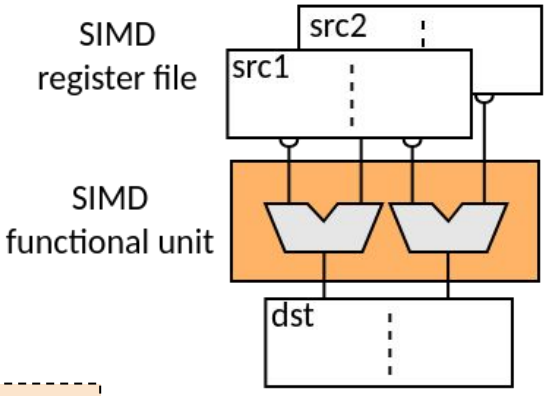
```
addi a0, a0, 4
addi a1, a1, 4
addi a2, a2, 4
addi s1, s1, -2
bnez s1, loop
```

```
addi a0, a0, 8
addi a1, a1, 8
addi a2, a2, 8
addi s1, s1, -2
bnez s1, loop
```

From: **16 instr** / 2 it
to: **9 instr** / 2 it

Remember arm-neon from yesterday?

It requires:
- New instructions in the ISA
- Separate register file
- New functional units



First implementation of SIMD

- This use of SIMD also called an “**Array processor**”.

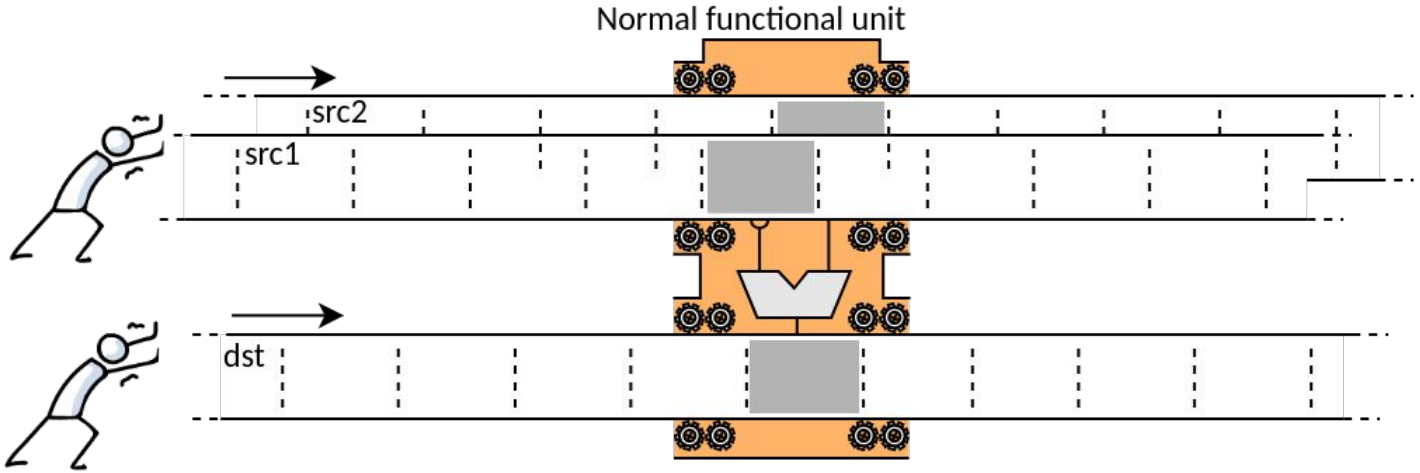
- Benefits:
  - Operate multiple data elements in one cycle
  - Reduce the number of loop iterations
  - Fewer instructions → Less I.Cache pressure, Fetching, Decode, …

- Drawbacks:
  - Can’t put too many Functional Units → Area, Power, Cost, …
  - We still get a lot of “loop control” instructions

- Is there a SIMD alternative to an **“Array processor”**?

Going longer (in another dimension)

- SIMD has another implementation in **Vector Processors**
- Increase the Register File length (like in Array processors)
- But without adding functional Units!
- Instructions are executed / pipelined in a "***Hardware loop***"



Vector Processors

- Example: A machine with vector registers of 64*100 bits
- The length of the vectors is set first, and the loop “disappears”:

```
for(int i=0; i<100; ++i){
    C[i] = A[i] + B[i];
}
```

```
a0 ← &A[0]
a1 ← &B[0]
a2 ← &C[0]

set_vl 100
“loop”:
vector_load v0, 0(a0), 4
vector_load v1, 0(a1), 4
vector_add v2, v0, v1
vector_store v2, 0(a2), 4
```

What if the loop < 100? (e.g. 10)

```
set_vl 10
```

Stride between memory elements

What if the loop > 100?

```
a0 ← &A[0]
a1 ← &B[0]
a2 ← &C[0]
mv s1 ← 1000-1

set_vl 100
vec_loop:
vector_load v0, 0(a0), 4
vector_load v1, 0(a1), 4
vector_add v2, v0, v1
vector_store v2, 0(a2), 4
addi a0, a0, 4*100
addi a1, a1, 4*100
addi a2, a2, 4*100
addi s1, s1, -100
bnez s1, vec_loop
```

We got “loop control” instructions again…

… But they are executed 100x fewer times

And take 100x fewer cycles than vector operations

Performance comparison

Still using this code

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

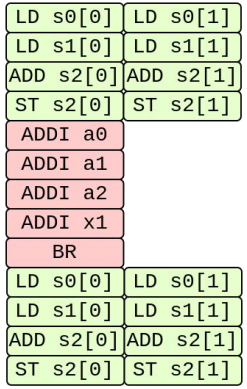
```
a0 ← &A[0]
a1 ← &B[0]
a2 ← &C[0]
mv s1 ← 4-1

loop:
load t0, 0(a0)
load t1, 0(a1)
add t2, t0, t1
store t2, 0(a2)
addi a0, a0, 4
addi a1, a1, 4
addi a2, a2, 4
addi s1, s1, -1
bnez s1, loop
```

Time

Scalar CPU

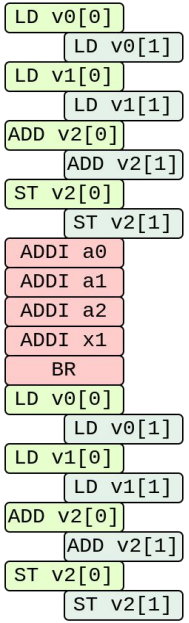
#FU: 1
#R.LEN(e): 1
Instr: 31
Cycles: 31

Array CPU



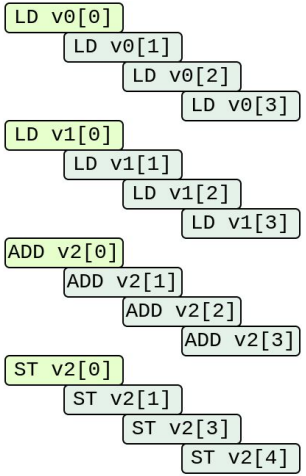
#FU: 2
#R.LEN(e): 2
Instr: 13
Cycles: 13

Vector CPU



#FU: 1
#R.LEN(e): 2
Instr: 13
Cycles: 21

Vector CPU

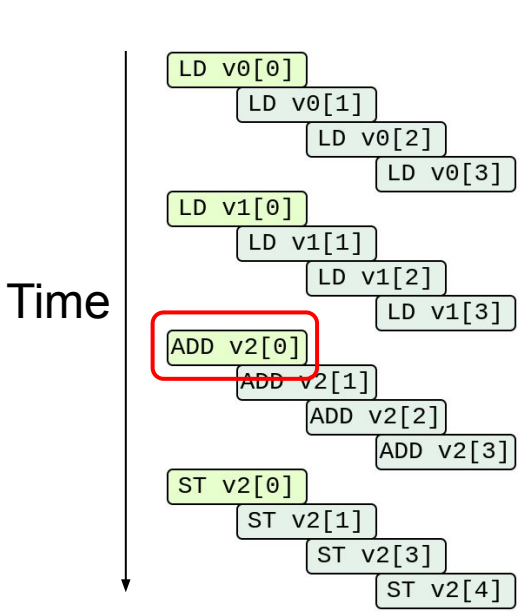


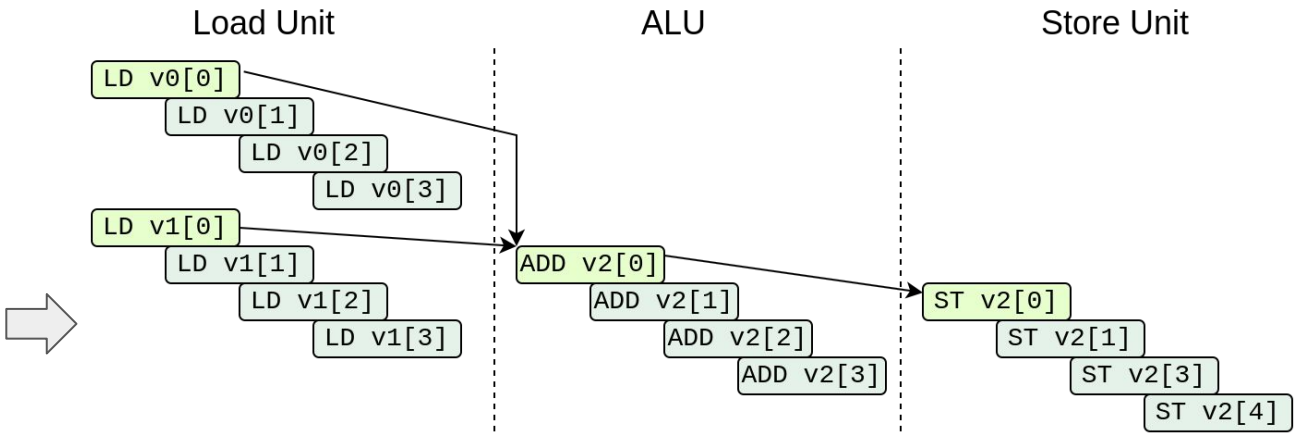
#FU: 1
#R.LEN(e): 4
Instr: 4
Cycles: 16

IPC < 1 !

Speeding Vector Computing

- Can we start the addition (ADD v2[0]) before? (Remember yesterday's lesson!)

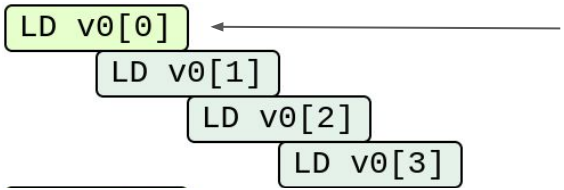




Yes! We can apply **Vector Chaining** using bypasses!
→ from 16 cycles to 10
→ from 0.25 IPC to 0.4

Verdict on Vector Computing

- Benefits:
  - Strongly reduce the number of all loop instructions (both loop control and compute)
  - No need to replicate Functional Units → Cheaper, Simpler, more efficient!
  - Exposing a lot of work at once can helps prefetchers



We know VL=4, we know the three next accesses are coming!

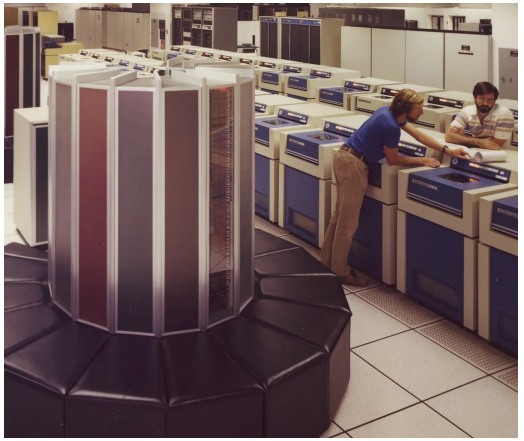
- Drawbacks:
  - Vector instructions take longer to compute (IPC < 1)
    - but can be chained to hide it!
  - Needs a big register file → Cost in Area and Power
  - Code needs to have long loops with regular operations (the same on all data)

First implementations of vector supercomputers

CDC STAR-100:

- Designed by Control Data Corporation in 1974
- 64-bit processor @ 25MHz
- In-memory vectors (i.e. **no vector register file**)
- 65 vector instructions (including masks, permutations, …)
- Up to 65.535 double-precision elements (512 KB!) per vector.
- 100 MFLOPS of peak performance

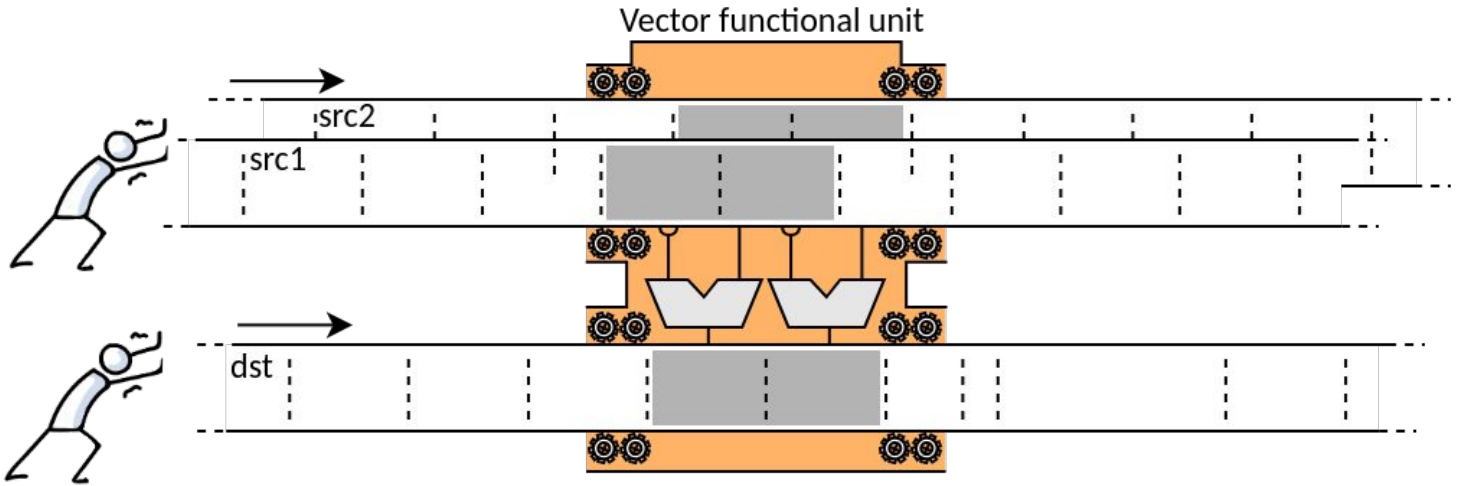
Cray-1



- Designed by Cray-Research in 1975
- 64-bit processor @ 80MHz
- In-register vectors
- 8 registers with 64x64bit elements (512 Bytes)
- 160 MFLOPS of peak performance

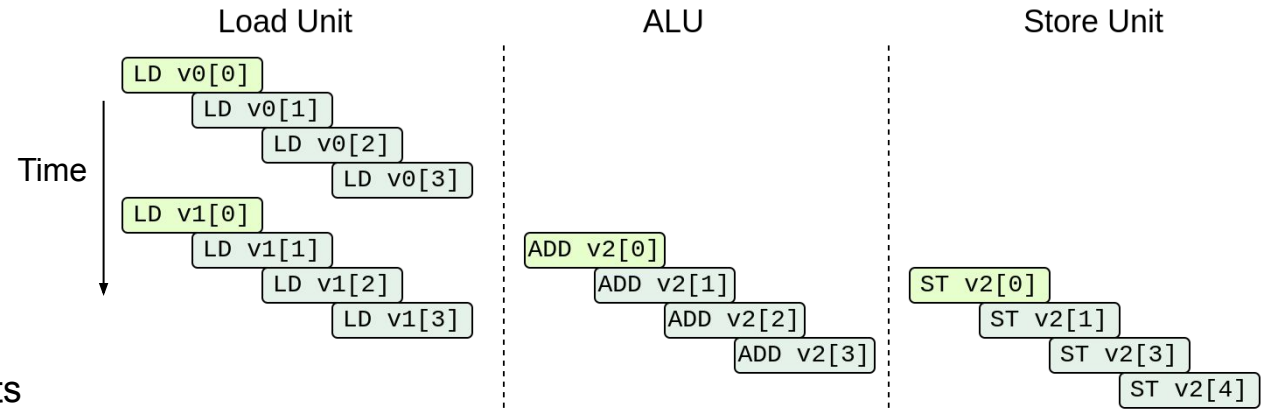
Modern SIMD

- The best of both worlds → Using **Array Computing** + **Vector Computing:**
- Long register file, but with more than one functional unit

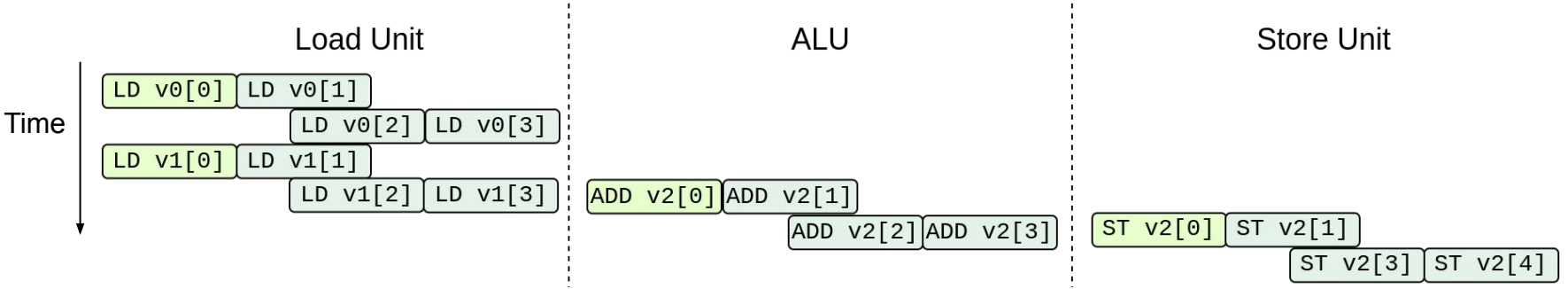


Modern SIMD

- Before:
  - 1 Functional Unit

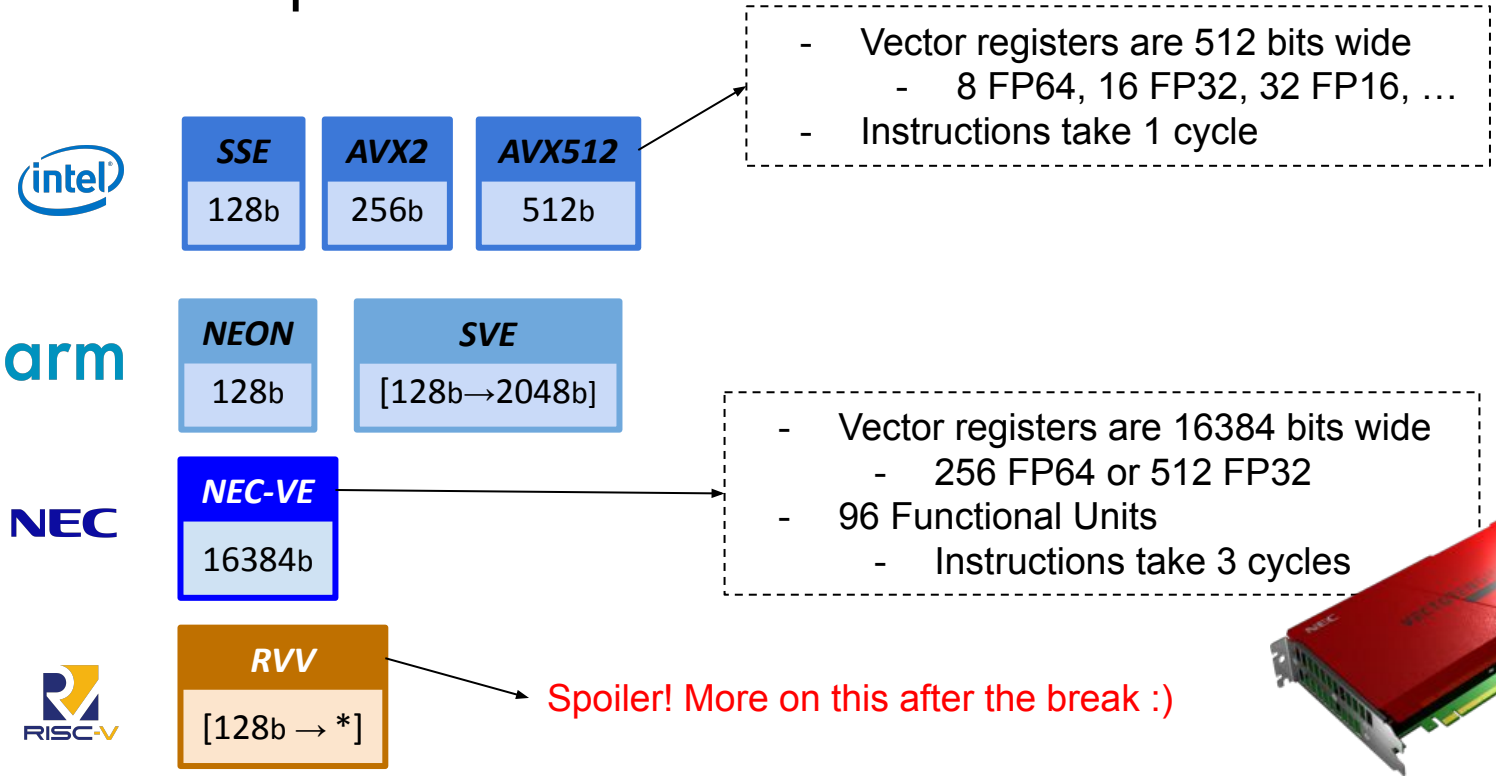


- Now:
  - E.g. 2 Functional Units
  - More expensive than traditional **Vector Computing**, but faster.



From 10 cycles (IPC=0.4) to 6 cycles (IPC=0.66)

Modern examples of SIMD

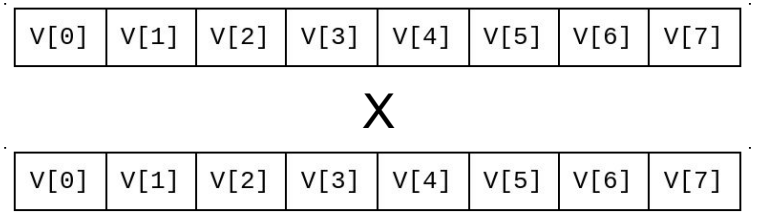


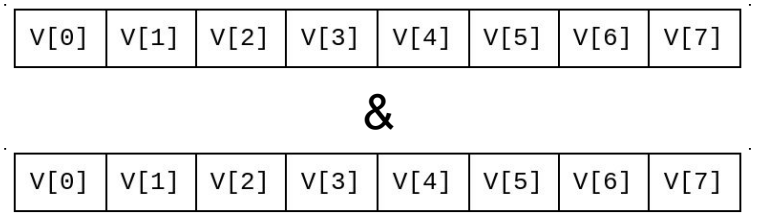
Vector Instructions in Detail

Arithmetic, Memory access modes, and masks

Vector Arithmetic Instructions

- Straight Forward implementation
- Floating Point instructions:
  - VFADD, VFMAD, VFDIV, …
- Integer:
  - VADD, VMUL, …
- Logical:
  - VOR, VAND, VXOR, …





Special case: Reductions

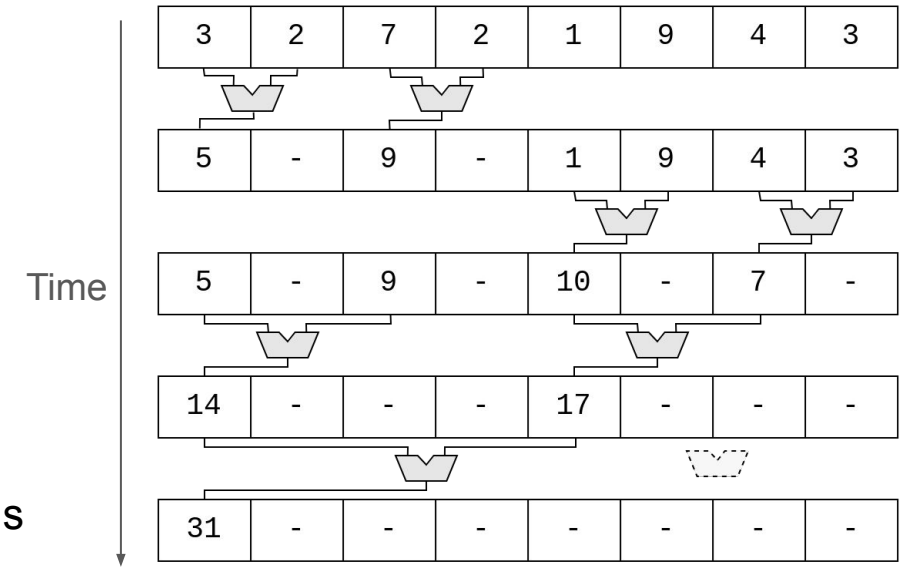
- How do we vectorize this code?

```
int V[8]
int result = 0;
for(int i=0; i<8; ++i){
  result += V[i];
}
```

We are not operating vectors together, but a vector on on itself

We use **Reduction** instructions

e.g. VL=8, FU=2



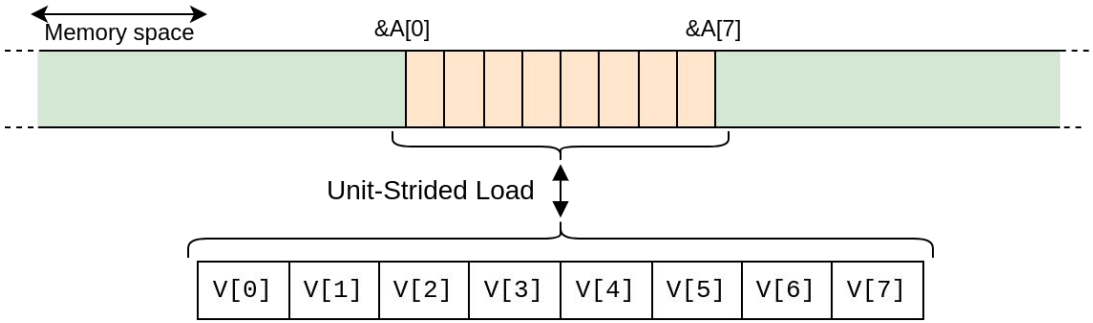
Needs many cycles to complete

Needs to “move” ALUs

Memory access modes

- Until now, we assumed memory accesses where contiguous → **Unit-Strided**

```
for(int i=0; i<8; ++i){
  B[i] = A[i];
}
```

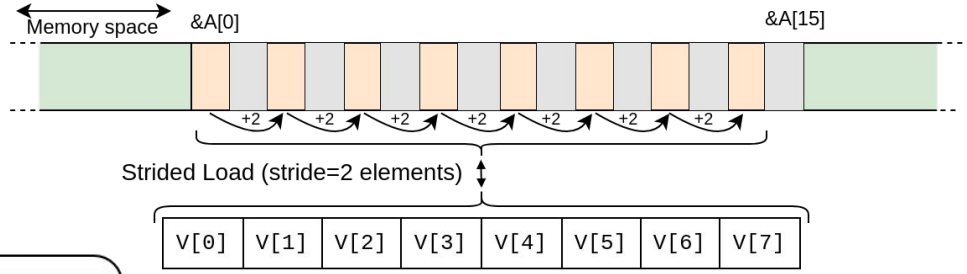


- This takes advantage of the Cache Line (reduces memory accesses)
- Thus, it's a fast access mode

Memory access modes

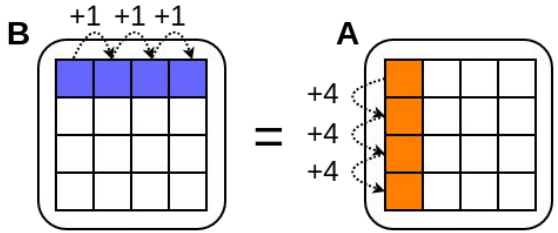
- But it is common to find other patterns in codes, such as **Strided** accesses:

```
for(int i=0; i<8; ++i){
  B[i] = A[2*i];
}
```



- e.g. Accessing a matrix by columns:

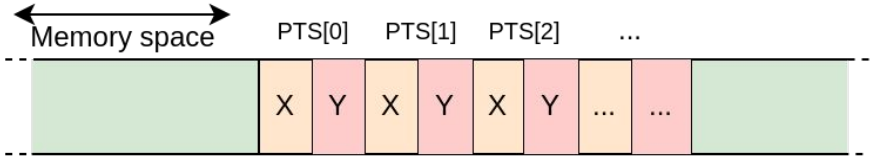
```
for(int i=0; i<4; ++i){
  B[i][j] = A[j][i];
}
```



- e.g. Or an array of structs

```
struct point{
  int X;
  int Y;
}
```

```
point PTS[8];
for(int i=0; i<8; ++i){
  PTS[i].X++;
  PTS[i].Y++;
}
```

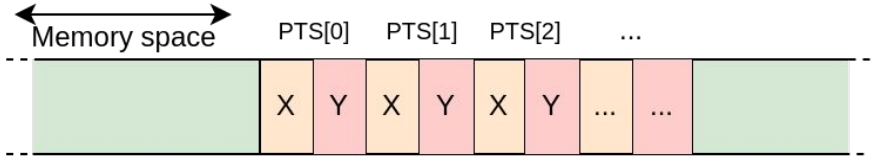


Array of structs vs struct of arrays

- Sometimes you can change your data structures to help vectorization:

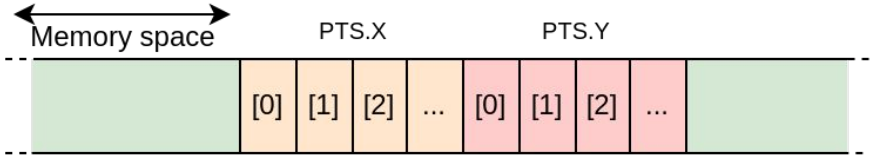
```
struct point{
  int X;
  int Y;
}
```

```
point PTS[8];
for(int i=0; i<8; ++i){
  PTS[i].X++;
  PTS[i].Y++;
}
```




```
struct points{
  int X[8];
  int Y[8];
}
```

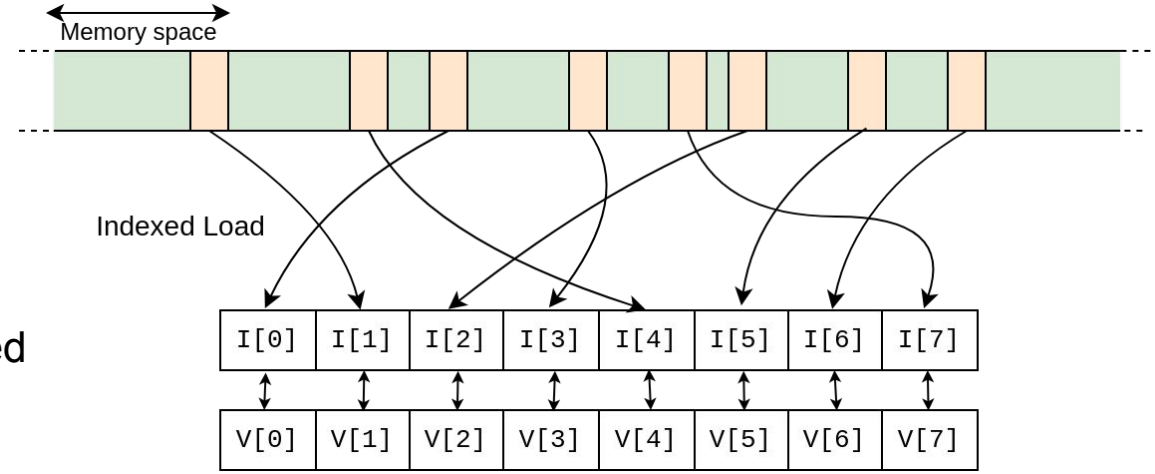
```
points PTS;
for(int i=0; i<8; ++i){
  PTS.X[i]++;
  PTS.Y[i]++;
}
```



Memory access modes

- Real-world codes can be even more chaotic! They can have **indexed** accesses
- Codes with indirections are not rare:

```
int A[8]
int I[8]
for (int i=0; i<8; ++i){
  A[ I[i] ] += 4
}
```



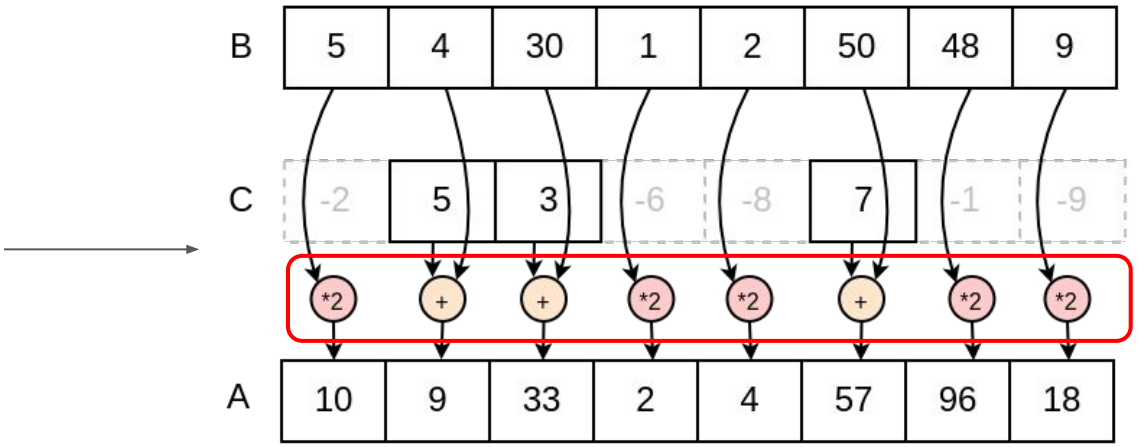
- Accesses might not be ordered
- Accesses might all point to:
  - Different cache lines
  - Different TLB pages

Indexed operations are expensive (long) … Avoid them when possible!

What about conditionals in loops?

- Loops usually have conditional flows, where not all elements are treated equally

```
int A[N]
int B[N]
int C[N]
for (int i=0; i<N; ++i){
  if (C[N] < 0){
     A[i] = B[i]*2;
  }else{
     A[i] = B[i] + C[i];
  }
}
```

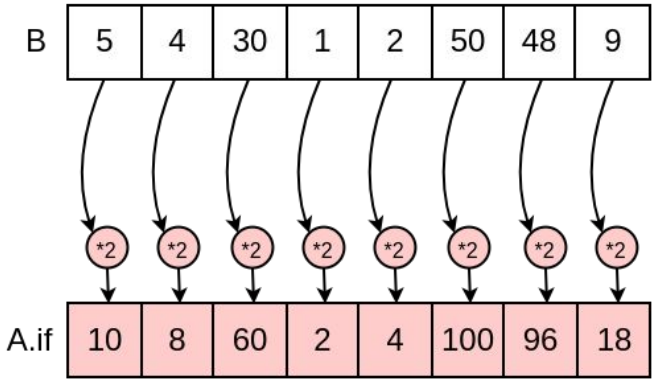


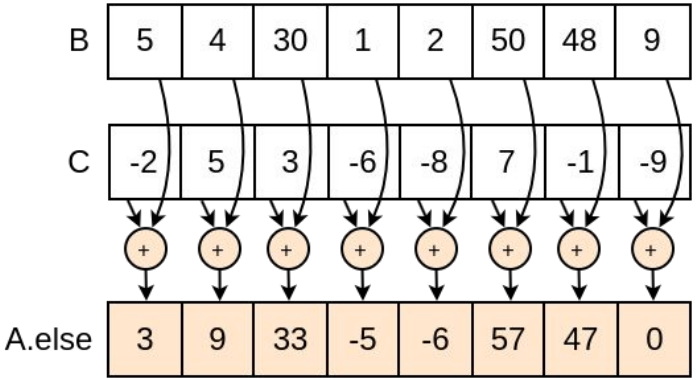
- One of the principles of vector computing is to **apply the same operation** to all elements, so we **cannot vectorize this code directly**

Going both ways at the same time

- We can compute both flows separately:

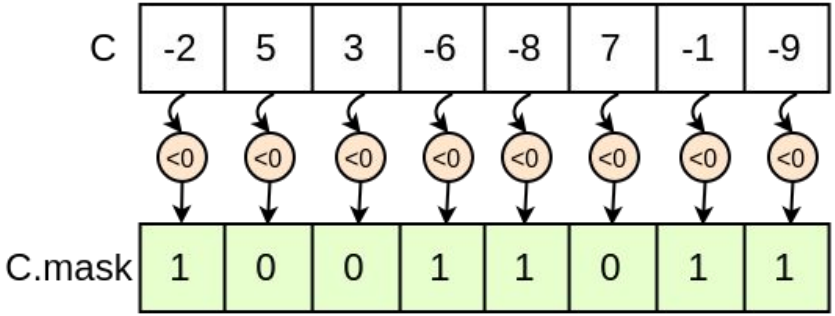
```
int A[N]
int B[N]
int C[N]
for (int i=0; i<N; ++i){
  if (C[N] < 0){
     A[i] = B[i]*2;
  }else{
     A[i] = B[i] + C[i];
  }
}
```





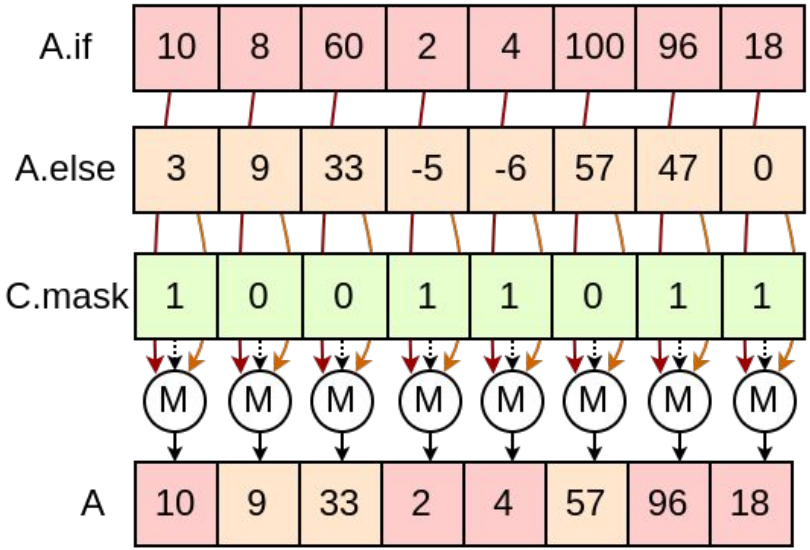
Creating a vector mask

- Then evaluate the conditional, creating a vector mask




```
for (int i=0; i<N; ++i){
  if (C[N] < 0){
      A[i] = B[i]*2;
  }else{
      A[i] = B[i] + C[i];
  }
}
```

- And finally merge the results



Efficiency of masks

- How many vector instructions did we need in the end?

```
int A[N]
int B[N]
int C[N]
for (int i=0; i<N; ++i){
  if (C[N] < 0){
     A[i] = B[i]*2;
  }else{
     A[i] = B[i] + C[i];
  }
}
```

```
vector_load v0, &B[0]
vector_mult v1, v0, 2    // B*2
vector_load v2, &C[0]
vector_sum v3, v0, v2    // B+C
vector_ltzero vm, v2
vector_merge v4, v1, v2, vm
vector_store v4, &A[0]
```

Quite a few! And this was one simple conditional…

Conclusion →Conditional can be vectorized, but come at a cost

- Beside Merges, other instructions can use masks:
  - **Arithmetic** instructions: When you want to **avoid** some **computations** (e.g. division by zero)
  - **Memory** instructions: When you want to **avoid** accessing some **addresses** (e.g. in an indexed)

Hardware support for masks

- The hardware can implement masks in two ways:

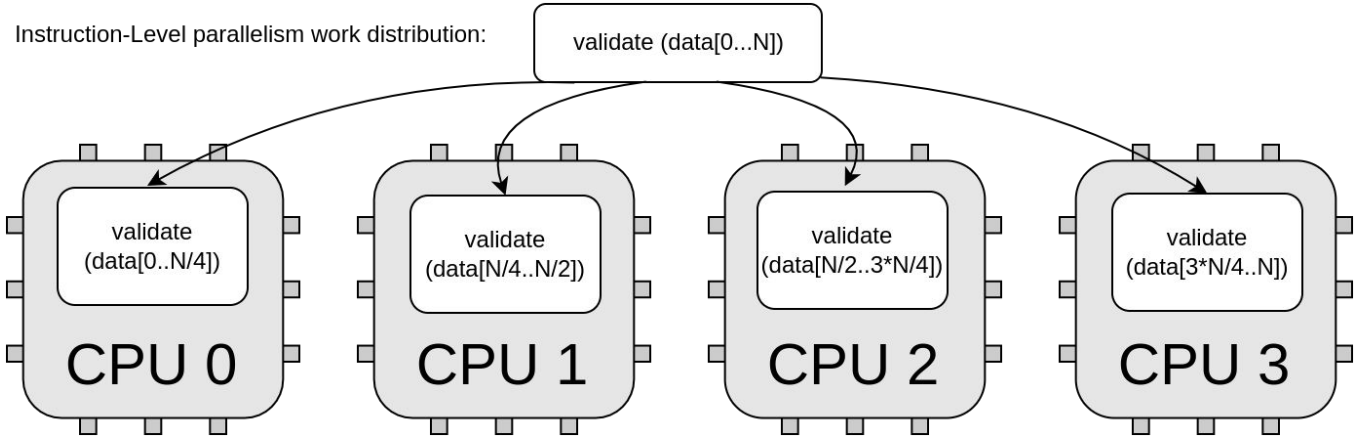
- **Option #1**: Perform the operation, but deactivate the writeback.
  - Simple solution
  - A fully masked-out vector instruction will still use the pipeline for all its cycles.

- **Option #2**: Scan the mask beforehand, skip masked elements
  - Needs extra logic
  - Requires to rapidly look for active element in the mask
  - Can improve performance when elements are masked

Limitations of SIMD

- We need long loops to take advantage of long vectors
- We cannot vectorize function calls (which could be parallelized):

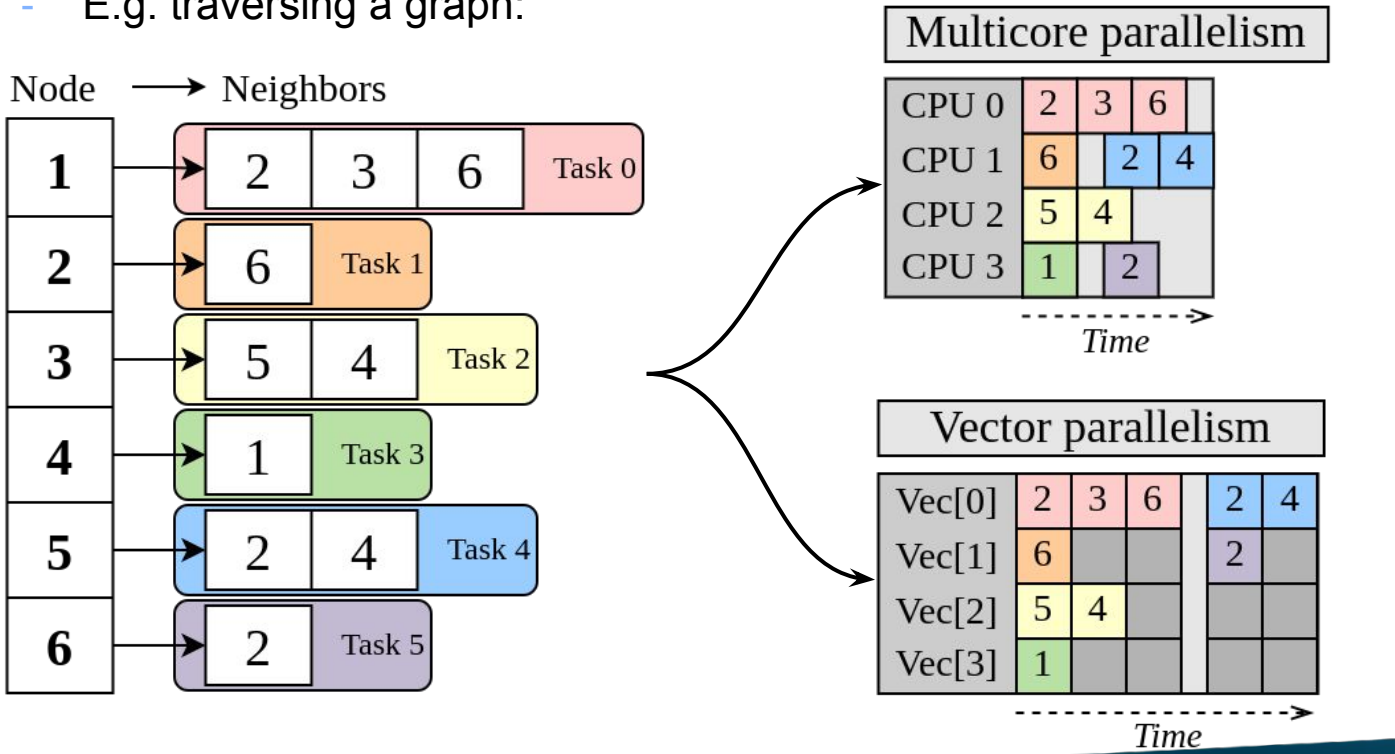
```
void validate(data * d){
  //...
}

for(int i=0; i<N; ++i){ ← Can't vectorize
   validate(data[i]);
}
```



Vector instructions only perform operations and memory accesses!
No function calls, no branches!

Limitations of SIMD

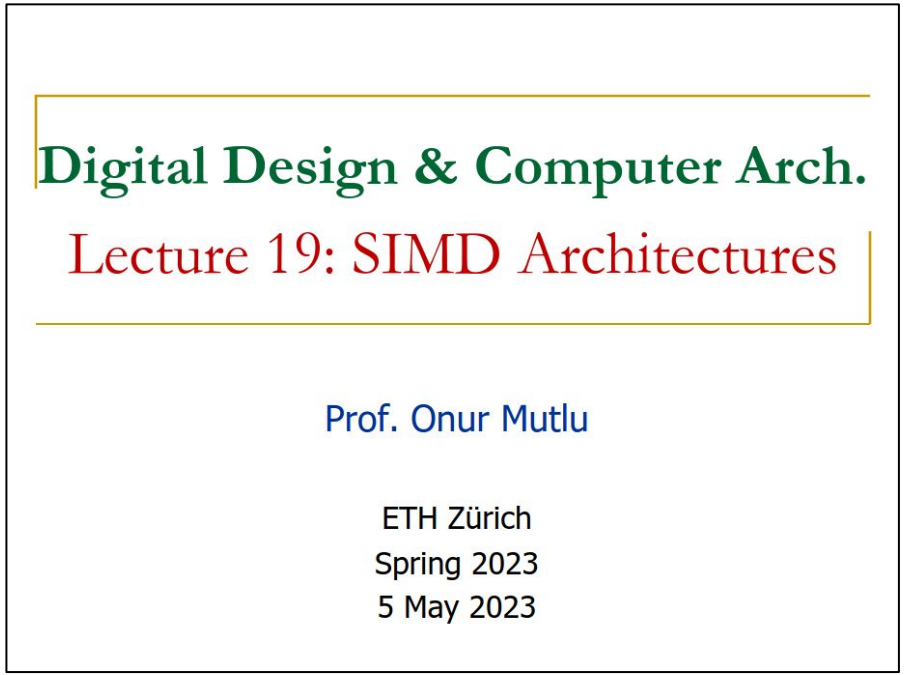
- Iterations must be regular to efficiently exploit data-level parallelism
- E.g. traversing a graph:



A lot of idle time!

Want to know more?

- I recommend Prof. Onur Mutlu's lecture:



https://safari.ethz.ch/digitaltechnik/spring2023/lib/exe/fetch.php?media=onur-ddca-2023-lecture19-simd-afterlecture.pdf

Also available at Youtube

https://www.youtube.com/live/gkMaO3yJMz0

Conclusions

- SIMD architectures are a mixture of **Array Computing** and **Vector Computing.**
- Loops are optimized by
  - Removing control flow instructions, fetching, decoding, …
  - Using multiple Functional Units at the same time
- Modern vector ISAs have flexible memory access modes and masking.
- But not all problems can be solved efficiently with loop vectorization!