Assessing vector processors

Vector processors: summary so far

Place data in vector registers for computation

Run same operation on every element of a vector

Necessary operations:

- Load and store data between memory and vector register
- Set vector length (setvl)
- Computations on vectors (add, multiply, reduce, compare, merge…)

Clarification: mask encoding

source

5.3.1. Mask Encoding

Where available, masking is encoded in a single-bit `vm` field in the instruction (`inst[25]`).

vm	Description
0	vector result, only where v0.mask[i] = 1
1	unmasked

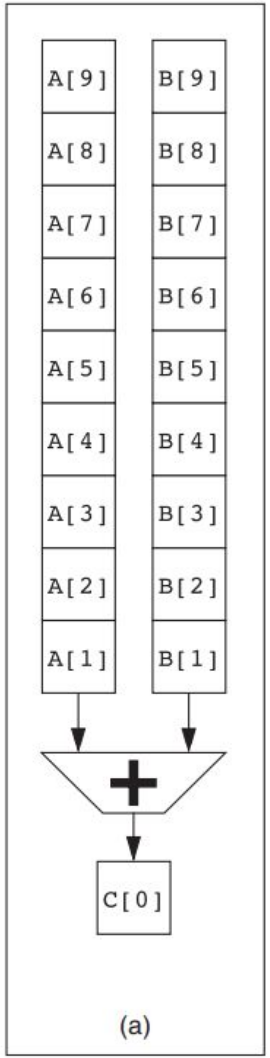
Vector masking is represented in assembler code as another vector operand, with `.t` indicating that the operation occurs when `v0.mask[i]` is `1` (`t` for "true"). If no masking operand is specified, unmasked vector execution (`vm=1`) is assumed.

```
vop.v*    v1, v2, v3, v0.t  # enabled where v0.mask[i]=1, vm=0
vop.v*    v1, v2, v3        # unmasked vector operation, vm=1
```

Mask example

```
for (int i = 0; i < 64; i++) {
    if (x[i] != 0) {
        y[i] = a * x[i];
    }
}
```

```
        li s0, a
        vld v1, s1
        vld v2, s2
        vmsne v0, v1, 0 # v2[i] = x[i] != 0 ? 1 : 0
        vmul.vx v1, v1, s0 # x[i] = a * x[i]
        vmerge v2, v2, v1, v0 # y[i] = v2[i] ? x[i] : y[i]
        vmul.vx v2, v1, s0, v0.t
        vst v1, s2
```



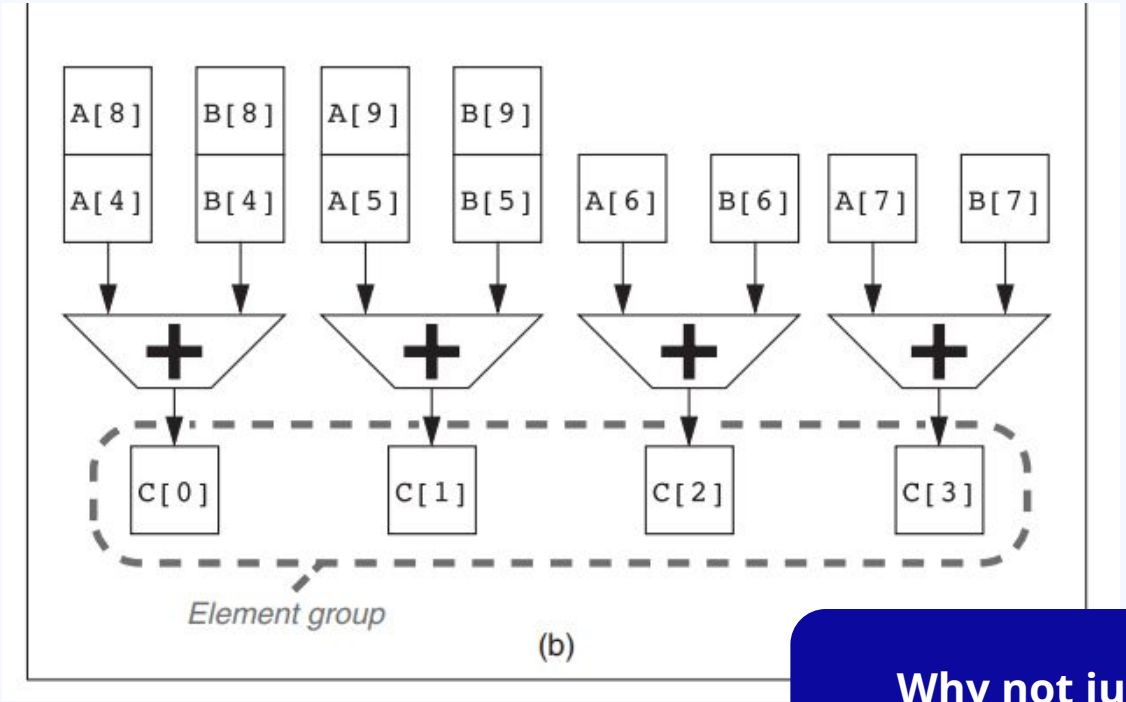
Simple approach: single pipelined FU

Upsides:

- Less hardware
- Smaller clock cycle
- One result/cycle
- Data within vector assumed independent: no hazards

H&P fig. 4.4

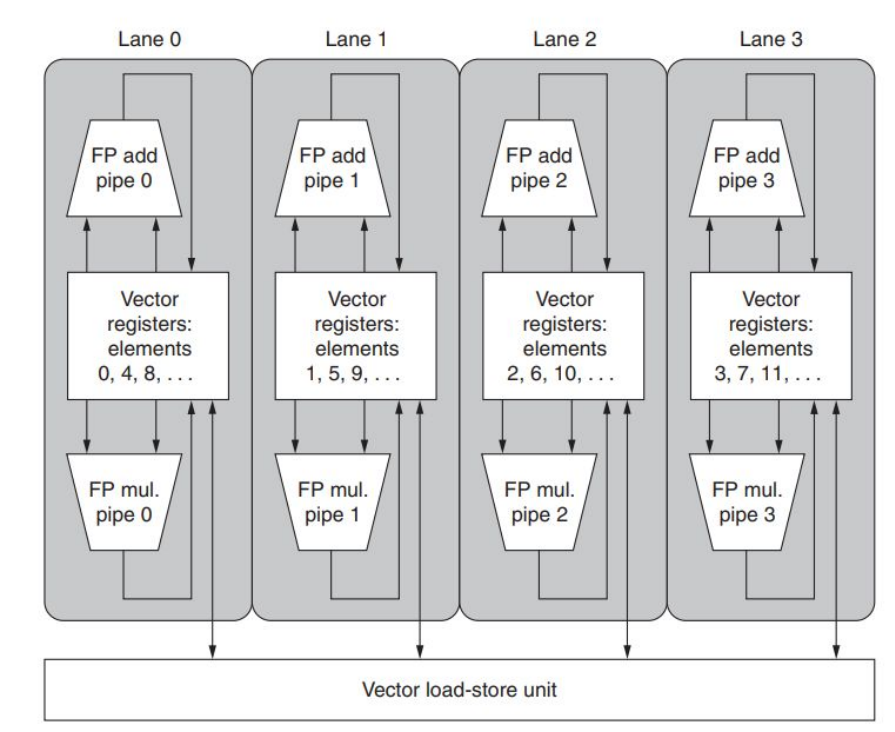
More efficient approach: multiple FUs



H&P fig. 4.4

Why not just have 64 unpipelined FUs?

Lanes



H&P fig. 4.5

Theory: if we can start a load every cycle, eventually, we get a throughput of 1 piece of data/cycle

Practice: how do we start a load every cycle if loads take multiple cycles?

Memory banks

Complication of memory access

How do we vectorize this code?

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

Strided loads/stores

7.5. Vector Strided Instructions

```
# Vector strided loads and stores

# vd destination, rs1 base address, rs2 byte stride
vlse8.v    vd, (rs1), rs2, vm  #    8-bit strided load
vlse16.v   vd, (rs1), rs2, vm  #   16-bit strided load
vlse32.v   vd, (rs1), rs2, vm  #   32-bit strided load
vlse64.v   vd, (rs1), rs2, vm  #   64-bit strided load

# vs3 store data, rs1 base address, rs2 byte stride
vsse8.v    vs3, (rs1), rs2, vm  #    8-bit strided store
vsse16.v   vs3, (rs1), rs2, vm  #   16-bit strided store
vsse32.v   vs3, (rs1), rs2, vm  #   32-bit strided store
vsse64.v   vs3, (rs1), rs2, vm  #   64-bit strided store
```

How do strided accesses complicate the advantages of memory banks?

Sparse accesses

Not all vector-like memory accesses use every element

```
for (int i = 0; i < n; i++) {
    X[m[i]] = X[m[i]] + Y[n[i]];
}
```

Solution: gather-scatter

Gather: collect all valid X[m[i]], Y[n[i]] in smaller vectors

Scatter: put the data back into X[m[i]], Y[n[i]]

In RISCV V: indexed load/stores, also the `vrgather` instruction

Also useful for avoiding computations on 0-valued elements (**why? where?**)

Measuring performance: chaining

What does it look like to execute this code w/ one load/store unit and one ALU/mul unit?

```
vld v0, s1
vmul.vx v1, v0, t0
vld v2, s2
vadd.vv v3, v1, v2
vst v3, t1
```

Measuring performance: convoys and chimes

Convoy: set of vector instructions that can potentially execute together

Chime: time it takes to execute a convoy

```
vld v0, s1
vmul.vx v1, v0, t0
vld v2, s2
vadd.vv v3, v1, v2
vst v3, t1
```

Approximation of runtime for this vector machine: 3 chimes (~3 * vlen clock cycles)
What complicates this metric?

Startup, dead time

Amdahl's law

Used to assess theoretical effectiveness of speedup

In a nutshell: gains in speeding up a portion of a program are limited by the fraction of time that portion is actually used

Mathematically:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

For parallelization: serial bottleneck (non-parallelizable code) limits effectiveness of vector processors

Compiler effectiveness

Processor	Compiler	Completely vectorized	Partially vectorized	Not vectorized
CDC CYBER 205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
Cray X-MP	CFT77 V3.0	69	3	28
Cray X-MP	CFT V1.15	50	1	49
Cray-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77 / SX V.040	66	5	29

Figure G.9 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the Cray X-MP show the large dependence on compiler technology.

Cray-1 Architecture

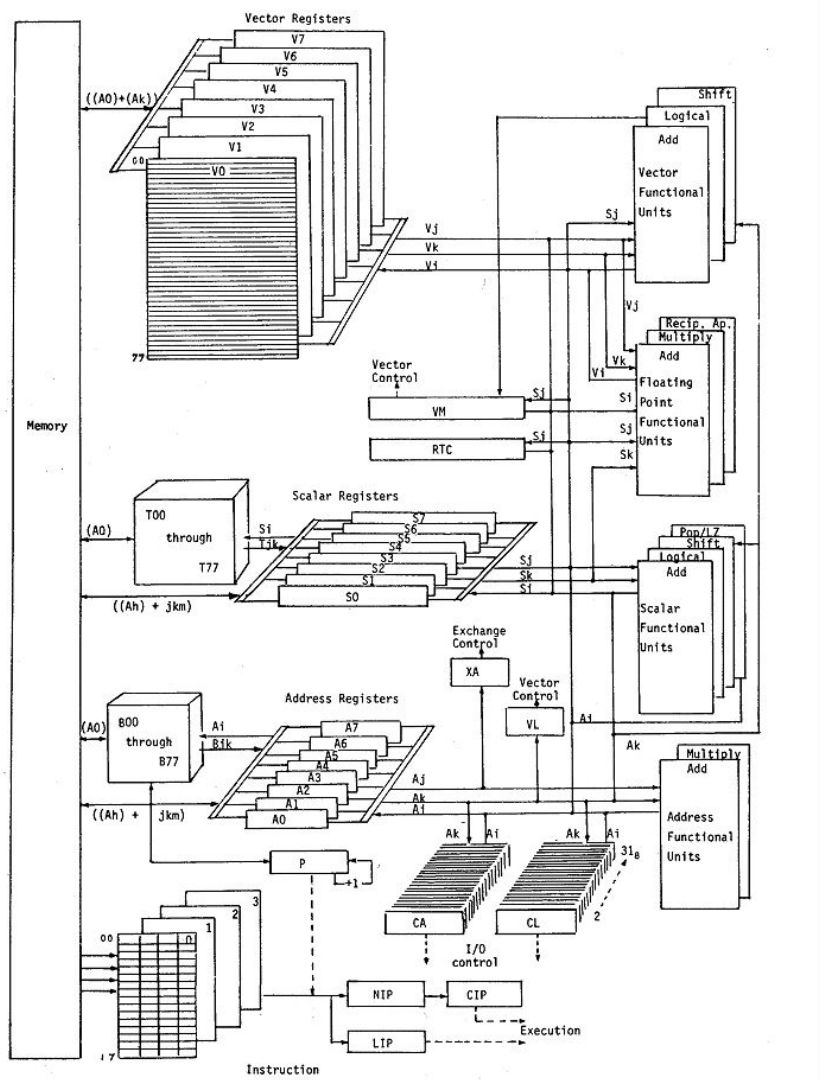


image source