Data-level parallelism

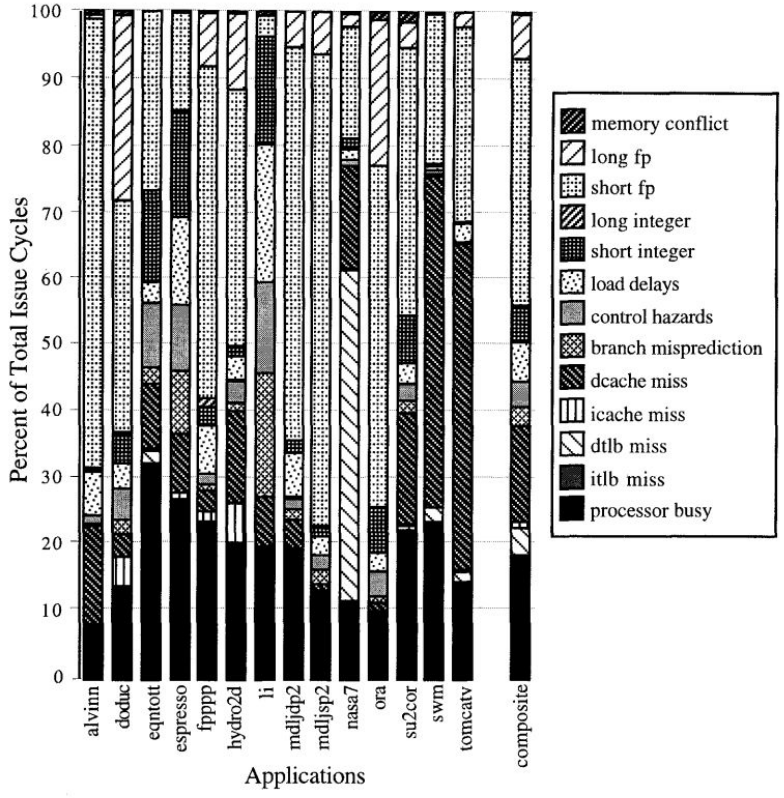
Parallelism so far

Our view of ILP (implementation + use) has largely been **application-agnostic**

Thread-level parallelism is also *largely* application-agnostic*, but performance varies by workload

What can we gain if our workload itself exhibits parallelism?

Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous multithreading: maximizing on-chip parallelism. In Proceedings of the 22nd annual international symposium on Computer architecture (ISCA '95). Association for Computing Machinery, New York, NY, USA, 392–403. link



? ? ?

What are examples of workloads that exhibit parallelism?

Workloads + their HW support

Workload	HW support
Basic sequential programs	**ILP techniques**
Task-level parallel workloads	**ILP, SMT, multicore, request-level parallelism (OS, servers, DB…)**
Data-level parallel workloads	**ILP and SMT *can* help vector processors and GPUs**

Matrix operations, some loops (scientific, multimedia, AI/ML applications)

IAXPY loop

“integer a*x + y” (also: saxpy, daxpy for single precision/doubles)

```
for (int i = 0; i < n; i++) {
    y[i] = a * x[i] + y[i]; // aX + Y
}
```

With (static or dynamic) ILP techniques:

load x	load y	inc i	
mul	load x	load y	inc i
add	mul	load x	load y
store	add	mul	inc i
branch	store	add	
branch	store		

load x
mul
load y
add
store
inc i
branch
load x
mul
load y
add
store
inc i
branch
load x
mul
load y

SIMD

"Single instruction, multiple data"

Perform same operation on different (independent) data in parallel

Requires additions to ISA (and compiler/programmer) *and* hardware

load x	load x	load x
mul	mul	mul
load y	load y	load y
add	add	add
store	store	store
inc i	inc i	inc i
branch	branch	branch

? ? ?

What advantages might SIMD operation have over basic ILP techniques?

load x	load y	inc i	
mul	load x	load y	inc i
add	mul	load x	load y
store	add	mul	inc i
branch	store	add	
branch	store		

load x	load x	load x
mul	mul	mul
load y	load y	load y
add	add	add
store	store	store

Flynn's taxonomy

Data stream \ Instruction stream	*Single*	*Multiple*
Single	SISD (More or less) what we've been studying so far	MISD Doesn't really exist commercially
Multiple	SIMD Different data goes into FUs performing same operation at same time	MIMD Independent processing units operating on independent data

Vector architectures

Hail from the 60s, popular in the the supercomputers of the 70s (Cray)

Place data in *vector registers* for computation

- Cray-1 (1976): 8 vector registers of 64 values each

Vector loads/stores can be pipelined: amortize latency

SIMD operation, but different from a "SIMD unit"… we'll come back to this

image source

Vector instructions (RISCV V ext)

Suffixes: .vv (vector-vector), .vx (vector-scalar), .vi (vector-immediate)

Lots of operations

At the minimum: load/store, operations on vectors

- Arithmetic/logical/shift: vadd, vsub, vrsub, etc
- Compare: vmseq, vmsne, vms{l,g}{t,e}[u]
- Max/min: vmin[u], vmax[u]
- Multiply-add (like dot product): vmacc, vnmsac, vmadd, vnmsub
- Reductions: vredsum, vredand, vredor, vredxor

Non-vectorized IAXPY loop

Y = αX + Y (|X|, |Y| = 64)

```
li s0, a # s0 = a
addi t0, s1, 256 # t0 = X + (64 * 4) (end address)
loop: lw t1, 0(s1) # t1 = x[i]
mul t1, t1, s0 # x[i] = x[i] * a
lw t2, 0(s2) # t2 = y[i]
add t2, t2, t1 # t2 = x[i] * a + y[i]
sw t2, 0(s2) # y[i] = t2 (x[i] * a + y[i])
addi s1, s1, 4 # increment x*
addi s2, s2, 4 # increment y*
bne s1, t0, loop
```

Vectorized IAXPY loop

```
li s0, a
addi t0, s1, 256
loop: lw t1, 0(s1)
mul t1, t1, s0
lw t2, 0(s2)
add t2, t2, t1
sw t2, 0(s2)
addi s1, s1, 4
addi s2, s2, 4
bne s1, t0, loop
```

```
li s0, a
vle32.v v0, s1 # v0 = X
vle32.v v1, s2 # v1 = Y
vmul.vx v0, v0, s0 # X = a * X
vadd.vv v1, v0, v1 # Y = a * X + Y
vse32.v v1, s2
```

OR JUST:

```
li s0, a
vle32.v v0, s1
vle32.v v1, s2
vmacc.vx v1, s0, v0 # Y = a * X + Y
vse32.v v1, s2
```

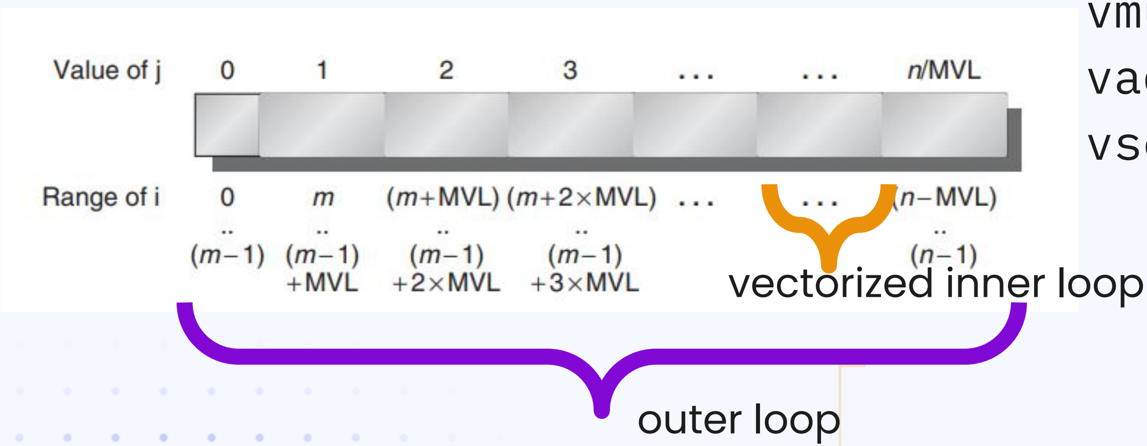
What assumptions are we making about our data here?

How to handle a loop like this?

```
for (int i = 0; i < 50; i++) {
    y[i] = a * x[i] + y[i];
}
```

```
vsetvli t0, s1, e32
# vl, t0 = min(MVL, s1)
li s0, a
vle32.v v0, s1
vle32.v v1, s2
vmul.vx v0, v0, s0
vadd.vv v1, v0, v1
vse32.v v1, s2
```

Used in **strip mining**:



What about these loops?

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

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

Masked/conditional instructions:

```
li s0, a
vle32.v v1, s1
vle32.v v2, s2
vmsne.vi v0, v1, 0
# v0[i] = x[i] != 0 ? 1 : 0
vmul.vx v2, v1, s0, v0.t
vse32.v v1, s2
```

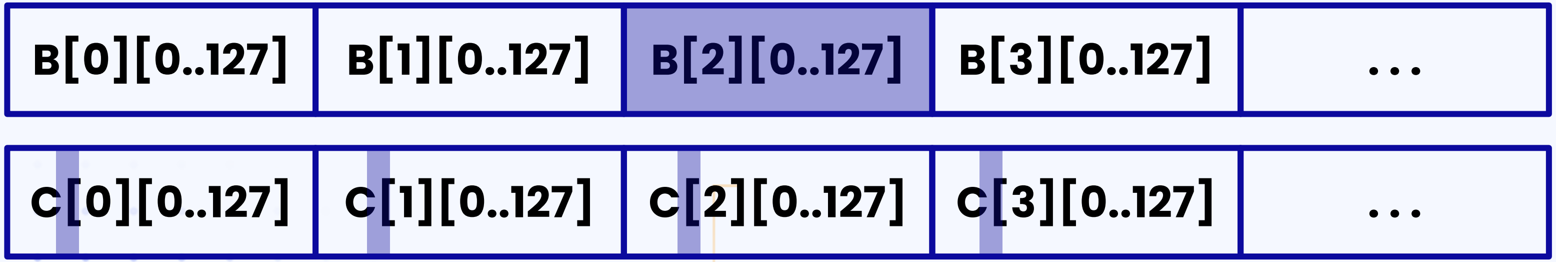
Gather: collect all valid X[m[i]], Y[n[i]] in smaller vectors

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

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

Matrix multiplies

```
for (int i = 0; i < 128; i++) {
    for (int j = 0; j < 128; j++) {
        A[i][j] = 0;
        for (int k = 0; k < 128; 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
```

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.