U. Wisconsin CS/ECE 752
# Advanced Computer Architecture I

Prof. Guri Sohi

Unit 7: Vectors for Data-Level Parallelism

Slides developed by Amir Roth of University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

Slides enhanced by Milo Martin, Mark Hill, and David Wood with sources that included Profs. Asanovic, Falsafi, Hoe, Lipasti, Shen, Smith, Sohi, Vijaykumar, and Wood

This Unit: Data/Thread Level Parallelism

Application	
OS	
Compiler	Firmware

CPU	I/O
Memory	
Digital Circuits	
Gates & Transistors	
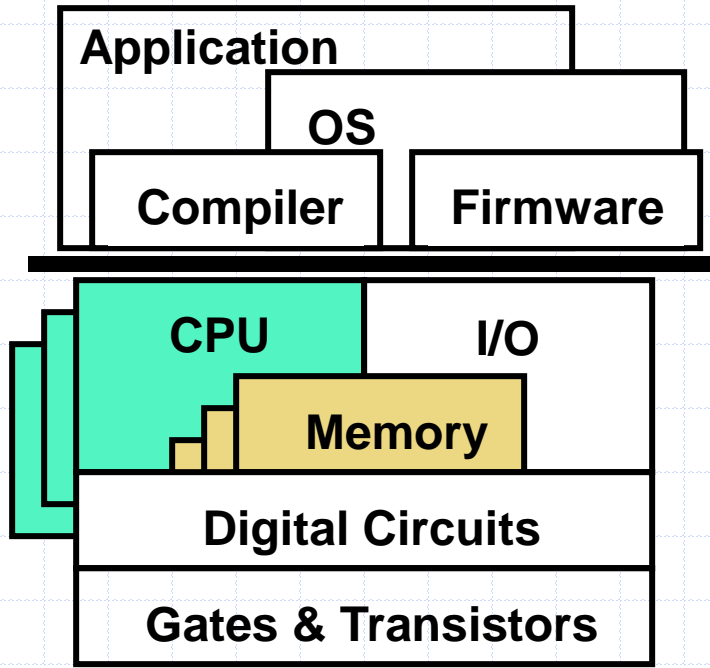
- Data-level parallelism
  - Vector processors
  - Message-passing multiprocessors
- Thread-level parallelism
  - Shared-memory multiprocessors
- Flynn Taxonomy

Latency, Bandwidth, and Parallelism

- **Latency**
  - Time to perform a single task
  - Hard to make smaller

- **Bandwidth**
  - Number of tasks that can be performed in a given amount of time
  - + Easier to make larger: overlap tasks, execute tasks in parallel

- One form of parallelism: **insn-level parallelism (ILP)**
  - Parallel execution of insns from a single sequential program
  - **Pipelining**: overlap processing stages of different insns
  - **Superscalar**: multiple insns in one stage at a time
  - Have seen

Exposing and Exploiting ILP

- ILP is out there…
  - Integer programs (e.g., gcc, gzip): ~10–20
  - Floating-point programs (e.g., face-rec, weather-sim): ~50–250
  - + It does make sense to build at least 4-way superscalar

- …but compiler/processor work hard to exploit it
  - Independent insns separated by branches, stores, function calls
  - Overcome with dynamic scheduling and speculation
  - – Modern processors extract ILP of 1–3

Fundamental Problem with ILP

- Clock rate and IPC are at odds with each other
  - Pipelining
    - + Fast clock
    - – Increased hazards lower IPC
  - Wide issue
    - + Higher IPC
    - – N^2 bypassing slows down clock

- Can we get both fast clock and wide issue?
  - Yes, but with a parallelism model less general than ILP
- **Data-level parallelism (DLP)**
  - Single operation repeated on multiple data elements
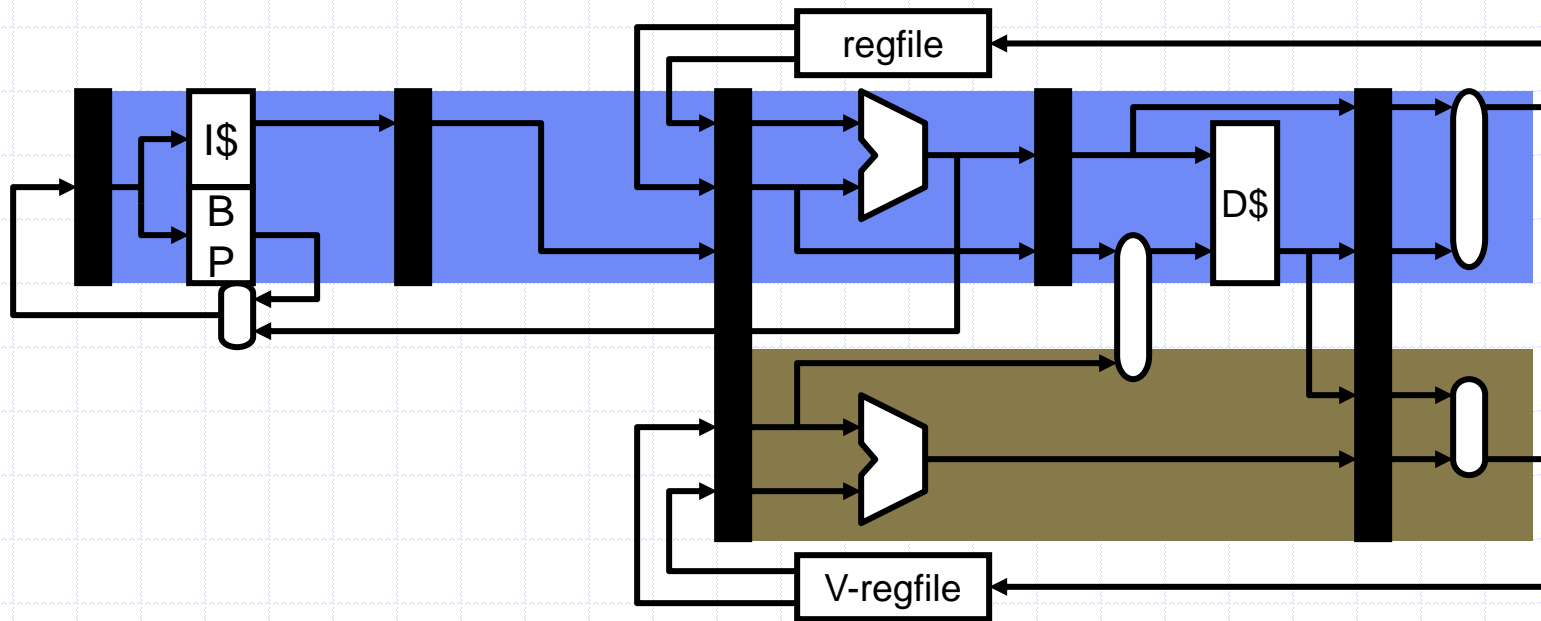 - Less general than ILP: parallel insns are same operation

Data-Level Parallelism (DLP)

```
for (I = 0; I < 100; I++)
   Z[I] = A*X[I] + Y[I];


0:ldf X(r1),f1              // I is in r1
   mulf f0,f1,f2           // A is in f0
   ldf Y(r1),f3
   addf f2,f3,f4
   stf f4,Z(r1)
   addi r1,4,r1
   blti r1,400,0
```

- One example of DLP: **inner loop-level parallelism**
  - Iterations can be performed in parallel

Exploiting DLP With Vectors

- One way to exploit DLP: **vectors**
  - Extend processor with **vector "data type"**
  - Vector: array of MVL 32-bit FP numbers
    - **Maximum vector length (MVL)**: typically 8–64
  - **Vector register file**: 8–16 vector registers (`v0`–`v15`)

Vector ISA Extensions

- Vector operations
  - Versions of scalar operations: `op.v`
  - Each performs an implicit loop over MVL elements
    ```
    for (I=0;I<MVL;I++) op[I];
    ```
  - Examples
  - `ldf.v X(r1),v1`: load vector
    ```
    for (I=0;I<MVL;I++) ldf X+I(r1),v1[I];
    ```
  - `stf.v v1,X(r1)`: store vector
    ```
    for (I=0;I<MVL;I++) stf v1[I],X+I(r1);
    ```
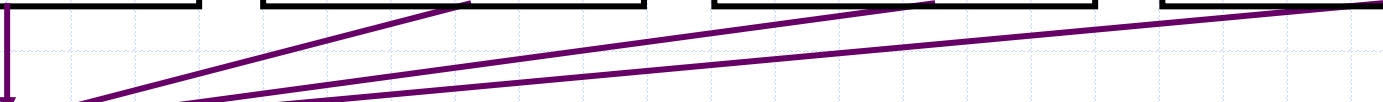 - `addf.vv v1,v2,v3`: add two vectors
    ```
    for (I=0;I<MVL;I++) addf v1[I],v2[I],v3[I];
    ```
  - `addf.vs v1,f2,v3`: add vector to scalar
    ```
    for (I=0;I<MVL;I++) addf v1[I],f2,v3[I];
    ```

Vectorizing SAXPY

```
ldf X(r1),f1        ldf X(r1),f1        ldf X(r1),f1        ldf X(r1),f1
mulf f0,f1,f2       mulf f0,f1,f2       mulf f0,f1,f2       mulf f0,f1,f2
ldf Y(r1),f3        ldf Y(r1),f3        ldf Y(r1),f3        ldf Y(r1),f3
addf f2,f3,f4       addf f2,f3,f4       addf f2,f3,f4       addf f2,f3,f4
stf f4,Z(r1)        stf f4,Z(r1)        stf f4,Z(r1)        stf f4,Z(r1)
addi r1,4,r1        addi r1,4,r1        addi r1,4,r1        addi r1,4,r1
blti r1,400,0       blti r1,400,0       blti r1,400,0       blti r1,400,0
```

```
ldf.v X(r1),v1
mulf.vs v1,f0,v2
ldf.v Y(r1),v3
addf.vv v2,v3,v4
stf.v v4,Z(r1)
addi r1,16,r1
blti r1,400,0
```

- Pack **loop body** into vector insns
  - Horizontal packing changes execution order
- Aggregate **loop control**
  - Add increment immediates

Scalar SAXPY Performance

```
ldf X(r1),f1
mulf f0,f1,f2
ldf Y(r1),f3
addf f2,f3,f4
stf f4,Z(r1)
addi r1,4,r1
slti r1,400,r2
bne Loop
```

- Scalar version
  - 5-cycle `mulf`, 2-cycle `addf`, 1 cycle others

  - 100 iters * 11 cycles/iter = 1100 cycles

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
ldf X(r1),f1	F	D	X	M	W														
mulf f0,f1,f2		F	D	d*	E*	E*	E*	E*	E*	W									
ldf Y(r1),f3			F	p*	D	X	M	W											
addf f2,f3,f4					F	D	d*	d*	d*	E+	E+	W							
stf f4,Z(r1)						F	p*	p*	p*	D	X	M	W						
addi r1,4,r1										F	D	X	M	W					
blt r1,r2,0											F	D	X	M	W				
ldf X(r1),f1												F	D	X	M	W			

Vector SAXPY Performance

```
ldf.v X(r1),v1
mulf.vs v1,f0,v2
ldf.v Y(r1),v3
addf.vv v2,v3,v4
stf.v v4,Z(r1)
addi r1,16,r1
slti r1,400,r2
bne r2,Loop
```

- Vector version
  - 4 element vectors
  - 25 iters * 11 insns/iteration * = 275 cycles
  + Factor of 4 speedup

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`ldf.v X(r1),v1`	F	D	X	M	W														
`mulf.vv v1,f0,v2`		F	D	d*	E*	E*	E*	E*	E*	W									
`ldf.v Y(r1),v3`			F	p*	D	X	M	W											
`addf.vv v2,v3,v4`					F	D	d*	d*	d*	E+	E+	W							
`stf.v f4,Z(r1)`						F	p*	p*	p*	D	X	M	W						
`addi r1,4,r1`										F	D	X	M	W					
`blt r1,r2,0`											F	D	X	M	W				
`ldf X(r1),f1`												F	D	X	M	W			

Not So Fast

- A processor with 32-element vectors
  - 1 Kb (32 * 32) to cache? 32 FP multipliers?
- No: vector load/store/arithmetic units are **pipelined**
  - Processors have L (1 or 2) of each type of functional unit
    - L is called number of vector **lanes**
  - Micro-code streams vectors through units M data elements at once
- Pipelined vector insn timing
  - $T_{vector} = T_{scalar} + (MVL / L) - 1$
  - Example: 64-element vectors, 10-cycle multiply, 2 lanes
  - $T_{mulf.vv} = 10 + (64 / 2) - 1 = 41$
  - + Not bad for a loop with 64 10-cycle multiplies

Pipelined Vector SAXPY Performance

```
ldf.v X(r1),v1
mulf.vs v1,f0,v2
ldf.v Y(r1),v3
addf.vv v2,v3,v4
stf.v v4,Z(r1)
addi r1,16,r1
slti r1,400,r2
bne r2,Loop
```

- Vector version
  - 4-element vectors, 1 lane
  - 4-cycle `ldf.v`/`stf.v`
  - 8-cycle `mulf.sv`, 5-cycle `addf.vv`
  - 25 iters * 20 cycles/iter = 500 cycles
  - Factor of 2.2 speedup

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`ldf.v X(r1),v1`	F	D	X	**M**	M	M	M	W											
`mulf.sv v1,f0,v2`		F	D	d*	d*	d*	d*	**E***	**E***	**E***	**E***	**E***	E*	E*	E*	W			
`ldf.v Y(r1),v3`			F	p*	p*	p*	p*	D	X	**M**	M	M	M	W					
`addf.vv v2,v3,v4`								F	D	d*	d*	d*	d*	d*	d*	**E+**	**E+**	E+	E+
`stf.v f4,Z(r1)`									F	p*	p*	p*	p*	p*	p*	D	X	d*	d*
`addi r1,4,r1`																F	D	p*	p*
`blt r1,r2,0`																	F	p*	p*
`ldf.v X(r1),f1`																			

Not So Slow
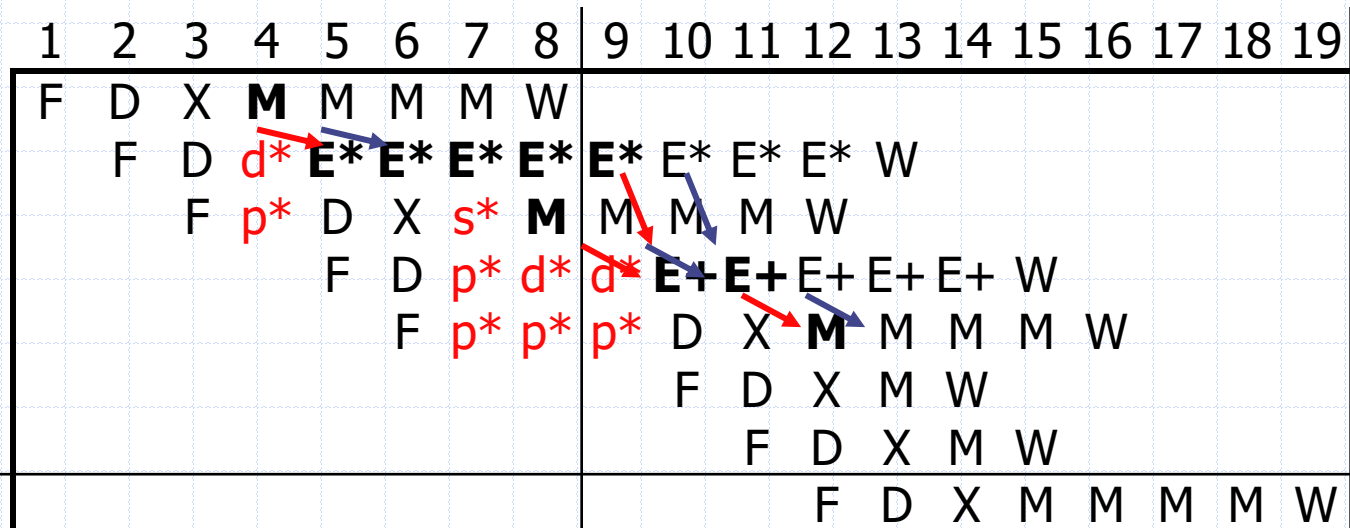
- For a given vector operation
  - All MVL results complete after $T_{scalar} + (MVL / L) - 1$
  - First M results (e.g., `v1[0]` and `v1[1]`) ready after T_{scalar}
  - Start dependent vector operation as soon as those are ready

- **Chaining**: pipelined vector forwarding
  - $T_{vector1} = T_{scalar1} + (MVL / L) - 1$
  - $T_{vector2} = T_{scalar2} + (MVL / L) - 1$
  - $T_{vector1} + T_{vector2} = T_{scalar1} + T_{scalar2} + (MVL / L) - 1$

Chained Vector SAXPY Performance

```
ldf.v X(r1),v1
mulf.vs v1,f0,v2
ldf.v Y(r1),v3
addf.vv v2,v3,v4
stf.v v4,Z(r1)
addi r1,16,r1
slti r1,400,r2
bne r2,Loop
```

- Vector version
  - 1 lane
  - 4-cycle `ldf.v`/`stf.v`
  - 8-cycle `mulf.sv`, 5-cycle `addf.vv`
  - 25 iters * 11 cycles/iter = 275 cycles
  + Factor of 4 speedup again

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`ldf.v X(r1),v1`	F	D	X	M	M	M	M	W											
`mulf.vv v1,f0,v2`		F	D	d*	E*	E*	E*	E*	E*	E*	E*	E*	W						
`ldf.v Y(r1),v3`			F	p*	D	X	s*	M	M	M	M	W							
`addf.vv v2,v3,v4`				F	D	p*	d*	d*	E+	E+	E+	E+	E+	W					
`stf.v f4,Z(r1)`					F	p*	p*	p*	D	X	M	M	M	M	W				
`addi r1,4,r1`									F	D	X	M	W						
`blt r1,r2,0`										F	D	X	M	W					
`ldf.v X(r1),f1`												F	D	X	M	M	M	M	W

Vector Performance

- Where does it come from?
  - + Fewer loop control insns: `addi`, `blt`, etc.
    - Vector insns contain implicit loop control
  - + RAW stalls taken only once, on "first iteration"
    - Vector pipelines hide stalls of "subsequent iterations"

- How does it change with vector length?
  - + Theoretically increases, think of T_{vector}/MVL
    - $T_{vector} = T_{scalar} + (MVL / L) - 1$
    - $MVL = 1 \rightarrow (T_{vector}/MVL) = T_{scalar}$
    - $MVL = 1000 \rightarrow (T_{vector}/MVL) = 1$
  - – But vector regfile becomes larger and slower

Amdahl's Law

- **Amdahl's law**: the law of diminishing returns
  - $\text{speedup}_{total} = 1 / [\%_{vector} / \text{speedup}_{vector} + (1-\%_{vector})]$
  - Speedup due to vectorization limited by **non-vector portion**
  - In general: optimization speedup limited by unoptimized portion

  - Example: $\%_{opt} = 90\%$
    - $\text{speedup}_{opt} = 10 \rightarrow \text{speedup}_{total} = 1 / [0.9/10 + 0.1] = 5.3$
    - $\text{speedup}_{opt} = 100 \rightarrow \text{speedup}_{total} = 1 / [0.9/100 + 0.1] = 9.1$
    - $\text{Speedup}_{opt} = \infty \rightarrow \text{speedup}_{total} = 1 / [0.9/\infty + 0.1] = 10$

  - CRAY-1 rocked because it had fastest vector unit …
  - … and the fastest scalar unit

Variable Length Vectors

- **Vector Length Register (VLR)**: 0 < VLR < MVL
  - Implicit in all vector operations

    ```
    for (I=0; I<VLR; I++) { vop… }
    ```

  - Used to handle vectors of different sizes
  - General scheme for cutting up loops is **strip mining**
    - Similar to loop blocking (cuts arrays into cache-sized chunks)

    ```
    for (I=0; I<N; I++)
        Z[I] = A*X[I]+Y[I];


    VLR = N % MVL;
    for (J=0; J<N; J+=VLR, VLR=MVL)
        for (I=J; I<J+VLR; I++)
            Z[I] = A*X[I]+Y[I];
    ```

Vector Predicates

- **Vector Mask Register (VMR)**: 1 bit per vector element
  - Implicit predicate in all vector operations
    ```
    for (I=0; I<VLR; I++) if (VMR[I]) { vop… }
    ```
  - Used to vectorize loops with conditionals in them
    `seq.v`, `slt.v`, `slti.v`, etc.: sets vector predicates
    `cvmr`: clear vector mask register (set to ones)

    ```
    for (I=0; I<32; I++)
        if (X[I] != 0) Z[I] = A/X[I];


    ldf X(r1),v1
    sne.v v1,f0          // 0.0 is in f0
    divf.sv v1,f1,v2     // A is in f1
    stf.v v2,Z(r1)
    cvmr
    ```

Reductions

- Reduce vector to scalar

  ```
  S = 0;
  for (I=0; I<64; I++)
      S = S + X[I];   // other ops include max, min, etc.
  ```

- Not vectorizable?

  - Binary partitioning

    ```
    ldf.v X(r1),v1
    ldf.v X+32*8(r1),v2
    addf.vv v1,v2,v3
    stf.v v3,tmp(r1)        // tmp holds X[0]+X[32], X[1]+X[33]…
    ```

  - Repeat until faster to use scalar instructions

ILP vs. DLP

- Recall: fundamental conflict of ILP
  - High clock frequency **or** high IPC, not both
  - High clock frequency $\rightarrow$ deep pipeline $\rightarrow$ more hazards $\rightarrow$ low IPC
  - High IPC $\rightarrow$ superscalar $\rightarrow$ complex issue/bypass $\rightarrow$ slow clock
- DLP (vectors) sidesteps this conflict
  - + Key: operations within a vector insn are parallel $\rightarrow$ no data hazards
  - + Key: loop control is implicit $\rightarrow$ no control hazards
  - High clock frequency $\rightarrow$ deep pipeline + no hazards $\rightarrow$ high IPC
  - High IPC $\rightarrow$ natural wide issue + no bypass $\rightarrow$ fast clock

History of Vectors

- **Vector-register architectures**: "RISC" vectors
  - Most modern vector supercomputers (Cray, Convex)
  - Like we have talked about so far
  - Optimized for short-medium sized (8–64 element) vectors
- **Memory-memory vector architectures**: "CISC" vectors
  - Early vector supercomputers (TI ASC, CDC STAR100)
  - Optimized for (arbitrarily) long vectors
  - All vectors reside in memory
  - Require a lot of memory bandwidth
  - Long startup latency

Modern Vectors

- Both floating-point and integer vectors common today
  - But both of the parallel (not pipelined) variety
- Integer vectors
  - Image processing: a pixel is 4 bytes (RGBA)
  - Also: speech recognition, geometry, audio, tele-communications
- Floating-point vectors
  - Useful for geometry processing: 4x4 translation/rotation matrices
  - Also: scientific/engineering programs, digital signal processing
- Sub-word multimedia vectors
  - Intel MMX: 64-bit integer (2x32b, 4x16b, 8x8b)
  - Intel SSE: 64-bit FP (2x32b)
  - Intel SSE2: 128-bit FP (2x64b, 4x32b)
  - Motorola AltiVEC: 128-bit integer/FP (2x64b, 4x32b, 8x16b, 16x8b)

MMX [Peleg & Weiser, IEEE Micro, 8/1996]

- Goal: 2x performance in multimedia (audio, video, etc.)
- Sub-word vector in 64b FP register:
  - 8x8-bit bytes, 4x16-bit words, 2x32-bit, or 1x64-bit

- E.g., addb (for byte)
-         17   87   100   …. (5 more)
-        +17   13   200   ….
- --
-         34   100   255   ….

Note: saturating arithmetic common: 100 + 200 → max

- Also move to/from memory & pack/unpack w.r.t. normal

Array-based Processors

- Many processor nodes operating in parallel
  - Each with scalar or vector processor
  - Memory with each node
  - Vectors/Matrices distributed across nodes

- Early machines (Illiac IV)
  - E.g., 64 scalar PEs
- Later machines (Thinking Machines CM-5)
  - Thousands of vector PEs
- Current machines (IBM Blue Gene/L)
- Emerging machines (IBM Cell)
  - 8 SPEs and one PowerPC
- General Purpose GPUs (GPGPUs)

Automatic Vectorization

- **Automatic vectorization**
  - Compiler conversion of sequential code to vector code
    - Very difficult in general
  - Vectorization implicitly reorders operations
  - Invariably, loads and stores are some of those operations
  - How to tell whether load/store reordering is legal?
    - Possible in languages without references: e.g., FORTRAN
    - Hard (impossible?) in languages with references: e.g., C, Java
    - Compiler directives ("just do it!")

  - Compilers don't generate MMX and SSE code
  - Libraries of routines that exploit MMX and SSE are hand assembled

Not Everything Easy To Vectorize

```
for (I = 0; I < N; I++)
    for (J = 0; J < N; J++)
        for (K = 0; K < N; K++)
            C[I][J] += A[I][K] * B[K][J];
```

- Matrix multiply difficult to vectorize
  - Vectorization works on **inner loops**
  - The iterations in this inner loop are not independent

- Need to transform it

```
for (I = 0; I < N; I++)
    for (J = 0; J < N; J+=MVL)
        for (K = 0; K < N; K++)
            for (JJ = 0; JJ<MVL; JJ++)
                C[I][J+JJ] += A[I][K] * B[K][J+JJ];
```

Vector Energy

- Vectors are more power efficient than superscalar
    - For a given loop, vector code…
        + Fetches, decodes, issues fewer insns (obvious)
        + Actually executes fewer operations too (loop control)
    - Also remember: clock frequency is not power efficient
        + Vectors can trade frequency (pipelining) for parallelism (lanes)

- In general: hardware more power efficient than software
    - Custom circuits more efficient than insns on general circuits
    - Think of vectors as custom hardware for array-based loops

Summary

- Data-level parallelism (DLP)
  - + Easier form of parallelism than ILP
  - – Hard to exploit automatically

- Vectors (SIMD)
  - Extend processor with new data type: vector
  - + Very effective
  - – Only handles inner-loop parallelism

Cray-1 Implementation Challenges

- High density machine for speed
- Heat dissipation
  - Freon cooling
    - Complex metallurgy to bond aluminum to stainless steel
    - "loss of Freon is not itself a problem"….
- Power distribution and circuit reliability
  - Used current balanced design
    - differential outputs, all outputs are terminated
    - steer current to correct output, no di/dt
  - Required using simple gates
    - all logic used 5/4 AND/NAND gates (high and low speed grades)
    - all memory was 16x4 or 1024x1 SRAMs

Cray-1 Implementation Challenges

- High density machine for speed
- Heat dissipation
  - Freon cooling
    - Complex metallurgy to bond aluminum to stainless steel
    - "loss of Freon is not itself a problem"….
- Power distribution and circuit reliability
  - Used current balanced design
    - differential outputs, all outputs are terminated
    - steer current to correct output, no di/dt
  - Required using simple gates
    - all logic used 5/4 AND/NAND gates (high and low speed grades)
    - all memory was 16x4 or 1024x1 SRAMs

Exploiting DLP With Parallel Processing

```
for (I = 0; I < 100; I++)
   for (J = 0; J < 100; J++)
      for (K = 0; K < 100; K++)
         C[I][J] += A[I][K] * B[K][J];
```

- Matrix multiplication can also be **parallelized**

- **Outer loop parallelism**
  - **Outer loop** iterations are parallel
  - Run entire I or J loop iterations in parallel
  - Each iteration runs on a different processor
  - Each processor runs all K inner loop iterations sequentially

- Which is better? Do both!

Parallelizing Matrix Multiply
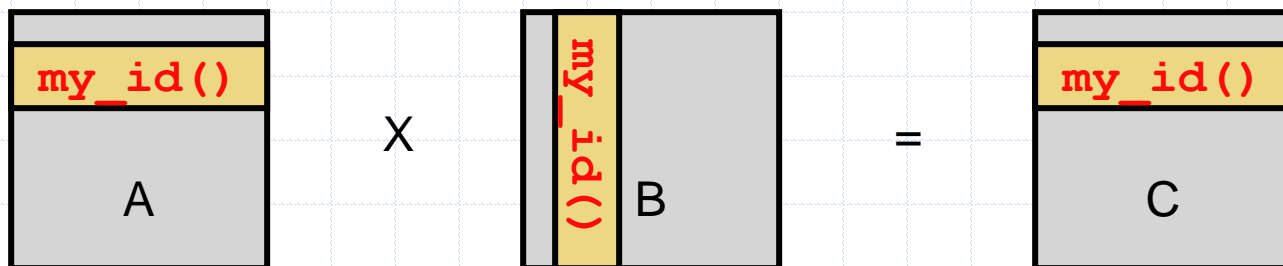

```
for (J = 0; J < N; J++)
    for (K = 0; K < N; K++)
        C[my_id()][J] += A[my_id()][K] * B[K][J];
```

- How to parallelize matrix multiply over N processors?
  - Or N machines in a cluster
- One possibility: give each processor an 1 iteration
  - Each processor runs copy of loop above
    - `my_id()` function gives each processor ID from 0 to N
    - Parallel processing library (e.g., MPI) provides this function
- Have to also divide matrices between N processors
  - Each processor gets row `my_id()` of `A`, `C`, column `my_id()` of `B`

Parallelizing Matrix Multiply

```
for (J = 0; J < 100; J++) {
   if (J == my_id()) {
      memcpy(tmp_B, my_B, 100);
      for (id = 0; id < 100; id++)
         if (id != my_id())
            send(id, &my_B, 100);
   }
   else recv(J, &tmp_B, 100);
   for (K = 0; K < 100; K++)
      my_C[J] += my_A[K] * tmp_B[K];
}
```

- **Data communication**
  - Processors send their portions of **B** (**my_B**) to other processors
  - Library provides **send()**, **recv()** functions for this

Parallelizing Matrix Multiply

```
if (my_id() == 0) {
    memcpy(tmp_A, &A[I][0], 100);
    memcpy(tmp_B, &B[0][J], 100);
    for (id = 1; id < 100; id++)
    { send(id, &A[id][0], 100); send(id, &B[0][id], 100); }
}
else { recv(0, &my_A, 100); recv(0, &my_B, 100); }


if (my_id() == 0)
    for (id = 1; id < 100; id++)
        recv(id, &C[id][0], 100);
else send(0, &my_C, 100);
```

- **Data initialization/collection**
  - Processor 0 must initialize others with portions of **A, B** matrices
  - Processor 0 must collect **C** matrix portions from other processors

Parallel Matrix Multiply Performance

- Gross assumptions
  - 10 cycles per FP instruction, all other instructions free
  - 50 cycles + 1 cycle for every 4 B to send/receive a message
- Sequential version: no communication
  - **Computation**: 2M FP-insn * 10 cycle/FP insn = **20M cycles**
- Parallel version: calculate for processor 0 (takes longest)
  - **Computation**: 20K FP-insn * 10 cycle/FP-insn = **200K cycles**
  - **Initialization**: ~200 send * 150 cycle/send = **30K cycles**
  - **Communication**: ~200 send * 150 cycle/send = **30K cycles**
  - **Collection**: ~100 send * 150 cycle/send = **15K cycles**
  - **Total: 275K cycles**
  - + 73X speedup (not quite 100X)
  - – 32% communication overhead

Parallel Performance

P (peak speedup)	10	100	1000
Computation	200,000*10=2M	20,000*10=200K	2000*10=20K
Initialization	20*(50+1000)=21K	200*(50+100)=30K	2000*(50+10)=120K
Communication	20*(50+1000)=21K	200*(50+100)=30K	2000*(50+10)=120K
Collection	10*(50+1000)=11K	100*(50+100)=15K	1000*(50+10)=60K
Total	2.05M	275K	320K
Actual speedup	9.7	73	63
Actual/Peak	97%	73%	6.3%

- How does it scale with number of processors P?
  - 97% efficiency for 10 processors, 73% for 100, 6.3% for 1000
  - 1000 processors actually slower than 100
    - Must initialize/collect data from **too many processors**
    - **Each transfer is too small**, can't amortize constant overhead
- Amdahl's law again
  - Speedup due to parallelization limited by **non-parallel portion**

Automatic Parallelization?

- Same as automatic vectorization: hard
  - Same reason: difficult to analyze memory access patterns
  - Maybe even harder
    - Outer loop analysis harder than inner loop analysis

Message Passing

- Parallel matrix multiply we saw uses **message passing**
    - Each copy of the program has a private virtual address space
    - Explicit communication through messages
        - Messages to other processors look like I/O
    - + Simple hardware
        - Any network configuration will will do
        - No need to synchronize memories
    - – Complex software
        - Must orchestrate communication
        - Only programs with regular (static) communication patterns

- Message passing systems called **multi-computers**

Shared Memory

```
"shared" float A[100][100], B[100][100], C[100][100];
for (J = 0; J < 100; J++)
    for (K = 0; K < 100; K++)
        C[my_id()][J] += A[my_id()][K] * B[K][J];
```

- Alternative: **shared memory**
  - All copies of program share (part of) an address space
  - **Implicit (automatic) communication via loads and stores**
  - + Simple software
    - No need for messages, communication happens naturally
      - Maybe too naturally
    - Supports irregular, dynamic communication patterns
  - – Complex hardware
    - Create a uniform view of memory
    - More complex on with caches

Issues for Shared Memory

- Shared memory not without issues
  - Cache coherence
  - Synchronization
  - Something called "memory consistency model"
  - Not unrelated to each other
  - Not issues for message passing systems
  - Topic of next unit

Thread Level Parallelism (TLP)

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;                                    0: addi r1,&accts,r3
if (accts[id].bal >= amt)                      1: ld 0(r3),r4
{                                              2: blt r4,r2,6
   accts[id].bal -= amt;                       3: sub r4,r2,r4
   dispense_cash();                            4: st r4,0(r3)
}                                              5: call dispense_cash
```

- But can also exploit **thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared loosely, dynamically
  - Dynamically allocate tasks to processors
- Example: database server (each query is a thread)
  - **accts** is **shared**, can't register allocate even if it were scalar
  - **id** and **amt** are private variables, register allocated to **r1**, **r2**
- Confusion: outer-loop DLP sometimes also called TLP

Summary: Flynn Taxonomy

- **Flynn taxonomy**: taxonomy of parallelism
  - Two dimensions
    - Number of instruction streams: single vs. multiple
    - Number of data streams: single vs. multiple

- **SISD**: single-instruction single-data
  - Pipelining and ILP on a uniprocessor
- **SIMD**: single-instruction multiple-data
  - DLP on a vector processor
- **MIMD**: multiple-instruction multiple-data
  - DLP, TLP on a parallel processor
  - **SPMD**: single-program multiple data

SISD vs. SIMD vs. SPMD

- SISD ruled the 1990s
  - ILP techniques found in all processors

- SIMD has its niche
  - Multimedia, tele-communications, engineering

- SPMD is starting to dominate commercially
  - + Handles more forms of parallelism
    - Inner-loop DLP, outer-loop DLP, and **TLP**
  - + More economical: just glue together cheap uniprocessors
  - + Better scalability: start small, add uniprocessors

Summary

- Data-level parallelism (DLP)
  - + Easier form of parallelism than ILP
  - – Hard to exploit automatically
- Vectors (SIMD)
  - Extend processor with new data type: vector
  - + Very effective
  - – Only handles inner-loop parallelism
- Parallel Processing (MIMD)
  - Multiple uniprocessors glued together
    - Glue? explicit messages or shared memory
  - + The way of the future: inner-loop and outer-loop DLP and TLP
  - + The way of the future: inner-loop and outer-loop DLP and TLP