This Unit: Data/Thread Level Parallelism

- Data-level parallelism
  - Vector processors
  - Message-passing multiprocessors
- Thread-level parallelism
  - Shared-memory multiprocessors
- Flynn Taxonomy

Unit: Data/Thread Level Parallelism

Readings

- H+P
  - Appendix E (skim)

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): \(\sim 10-20\)
  - Floating-point programs (e.g., face-rec, weather-sim): \(\sim 50-250\)
    + It does make sense to build 4-way and 8-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*Y[I] + Y[I];
```

- 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
    - ldv.v X(r1), v1: load vector
      for (I=0; I<MVL; I++) ldv 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];

Scalar SAXPY Performance

- Scalar version
  - 5-cycle mulf, 2-cycle addf, 1 cycle others
  - 100 iters * 11 cycles/iter = 1100 cycles

Vector Swaying Performance

- Vector version
  - 4 element vectors
  - 25 iters * 11 insns/iteration * = 275 cycles
  - Factor of 4 speedup
Not So Slow

- For a given vector operation
  - All MVL results complete after $T_{scalar} + (MVL / L) - 1$
  - First M results (e.g., $v_{1[0]}$ and $v_{1[1]}$) ready after $T_{scalar}$
  - Start dependent vector operation as soon as those are ready

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

---

Not So Fast

- A processor with 32-element vectors
  - 1 Kb ($32 \times 32$) to cache? 32 FP multipliers?
- No: vector load/store/arithemic 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
  - $T_{mult,vv} = 10 + (64 / 2) - 1 = 41$
  - Not bad for a loop with 64 10-cycle multiplies

---

Pipelined Vector SAXPY Performance

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

---

Chained Vector SAXPY Performance

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

---

CIS 501 (Martin/Roth): DLP
Variable Length Vectors

- **Vector Length Register (VLR):** $0 < \text{VLR} < \text{MVL}$
  - Implicit in all vector operations
  ```
  for (I=0; I<$\text{VLR}$; I++) { \text{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<$\text{N}$; I++)
  Z[I] = A*X[I]+Y[I];
  ```
  ```
  \text{VLR} = \text{N} \% \text{MVL};
  ```
  ```
  for (J=0; J<$\text{N}$; J+=\text{VLR}, \text{VLR}=\text{MVL})
  for (I=J; I<$\text{J}+\text{VLR}$; I++)
  Z[I] = A*X[I]+Y[I];
  ```

Vector Performance

- **Where does it come from?**
  - Fewer loop control insns: addi, bhi, 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 $T_{\text{vector}}/\text{MVL}$
    - $T_{\text{vector}} = T_{\text{scalar}} + (\text{MVL} / L) - 1$
    - $\text{MVL} = 1 \rightarrow (T_{\text{vector}}/\text{MVL}) = T_{\text{scalar}}$
    - $\text{MVL} = 1000 \rightarrow (T_{\text{vector}}/\text{MVL}) = 1$
    - But vector regfile becomes larger and slower

Amdahl’s Law

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

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

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

Vector Predicates

- **Vector Mask Register (VMR):** 1 bit per vector element
  - Implicit predicate in all vector operations
  ```
  \text{for (I=0; I<$\text{VLR}$; I++) \{ \text{vop...} \}}$
  ```
  - Used to vectorize loops with conditionals in them
    ```
    \text{seq.v, slt.v, slli.v, etc.: sets vector predicates}
    $$\text{cvmr: clear vector mask register (set to ones)}$
    ```
    ```
    $\text{for (I=0; I<$\text{N}$; I++)}
    $\text{if (X[I] != 0) Z[I] = A/X[I];}$
    $$\text{ldf X(v1),v1}$
    $$\text{sne.v v1,f0} \quad // 0.0 \text{ is in f0}$
    $$\text{divf.sv v1,f1,v2} \quad // A \text{ is in f1}$
    $$\text{stf.v v2,Z(v1)}$
    $$\text{cvmr}$
    ```
ILP vs. DLP

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

Short, Single-Cycle Vector Instructions

- Pipelining technique used in scientific vector architectures
  - Many elements per vector: 8 to 64
  - Large basic data type: 32- or 64-bit FP
  - Complex operations: `addf`, `vv`, `mulf`, `vs`
- More recently, multimedia vector architectures
  - Few elements per vector: 4 or 8 (64-bit or 128-bit vectors)
  - Short, simple basic data type: 8- or 16-bit integers
    - Entire vector can be “packed” into one 32- or 64-bit integer
    - Simple operations: `and`, `or`, `add`
      - Operations implemented in parallel in single ALU
      - Do 4 16-bit adds in 64-bit ALU by disabling some carries
  - This form of data-level parallelism called subword parallelism

History of Vectors

- Vector-register architectures: “RISC” vectors
  - Most modern vector supercomputers (Cray-1, 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
- Examples
  - 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)
Automatic Vectorization

- **Automatic vectorization**
  - Compiler conversion of sequential code to vector code
    - Very difficult
  - 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

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

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

Not Everything Easy To Vectorize

```c
for (I = 0; I < N; I++)
  for (J = 0; J < N; J++)
    for (K = 0; K < N; K++)
```

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

- Need to transform it
  ```c
  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++)
  ```

Exploiting DLP With Parallel Processing

```c
for (I = 0; I < 100; I++)
  for (J = 0; J < 100; J++)
    for (K = 0; K < 100; K++)
```

- 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

```
if (my_id() == 0) {
    memcpy(tmp_A, &A[0][0], 100);
    // 0 must initialize others with portions of A, B matrices
}
else {
    for (id = 1; id < 100; id++)
        if (id != my_id())
            send(id, &A[id][0], 100);
}

if (my_id() == 0) {
    for (J = 0; J < N; J++)
        for (K = 0; K < N; K++)
}
else
csv(0, &my_A, 100); csv(0, &my_B, 100);
```

- 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 \( B \)

Data communication

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

---

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: \( \sim 200 \) send * 150 cycle/send = \( 30K \) cycles
  - Communication: \( \sim 200 \) send * 150 cycle/send = \( 30K \) cycles
  - Collection: \( \sim 100 \) send * 150 cycle/send = \( 15K \) cycles
  - Total: \( 275K \) cycles
  - 73X speedup (not quite 100X)
  - 32% communication overhead
Parallel Performance

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

- 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 do
  - 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++)
```

- 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

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

Thread Level Parallelism (TLP)

```c
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt)
    dispense_cash();
accts[id].bal -= amt;
```

- 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`

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 unprocessors 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