Vectorization

Introduction to vector processing
Vector libraries
Vectorization of the N-Body problem

Vector Processing

- In computer science a vector is a generalization of the 3D vectors of the laws of motion in physics
  - Physics: \( \vec{r}_i = (x_i, y_i, z_i) \)
  - CS: 1D array of any length
  - CSE: contents are numbers
    - integer, real, complex
    - int, float, double, complex

- Algorithms that are organized as a series of operations on vectors can be much more efficient
  - scientific applications (e.g. PDE solvers, N-Body)
  - graphics (rendering, image processing)
  - audio

Running Example: Inner Product

- To see why vectors can be efficient, consider a simple loop to compute an inner product

\[
p = \sum_{i=0}^{n-1} x_i y_i
\]

- \( x, y \) are vectors of length \( n \)
- In C/C++:

```c
float p = 0.0;
for (int i = 0; i < n; i++)
  p += x[i] * y[i];
```

Scalar Processor

- Here is an assembly language version (MIPS R3000)

```assembly
ip0: slt $t3, $s0, $s2       # compare i to n
    beqz $t3, ip1          # exit loop when i >= n
    sll $t4, $s0, 2        # scale i (multiply by 4)
    lw $t0, x($t4)         # load x[i] into $t0
    lw $t1, y($t4)         # y[i] in $t1
    mul $t2, $t1, $t0      # x[i] * y[i] in $t2
    add $s1, $s1, $t2      # sum += x[i] * y[i]
    add $s0, $s0, 1        # i++
    b ip0
```
Here is an assembly language version (MIPS R3000)

```
ip0: slt $t3, $s0, $s2  # compare i to n
   beqz $t3, ip1       # exit loop when i >= n
   sll $t4, $s0, 2     # scale i (multiply by 4)
   lw $t0, x($t4)      # load x[i] into $t0
   lw $t1, y($t4)      # load y[i] into $t1
   mul $t2, $t1, $t0   # x[i] * y[i] in $t2
   add $s1, $s1, $t2   # sum += x[i] * y[i]
   add $s0, $s0, 1     # i++
   b ip0
```

Loop overhead (test end condition, manage iterator var, branches)

Machine registers hold single (scalar) values

Performance Factors

Items that determine the performance of this program:

- machine cycle time
- memory latency (time to read a single word)
  - cache hit: ~2 cycles
  - cache miss: ~100 or more cycles
- loop overhead

A simple metric: CPI (cycles per instruction)

- if instructions are executed sequentially, CPI is a measure of instruction complexity
- e.g. for $n = 1000$

\[
\frac{4 \times 4001 + 3 \times 2001 + 6 \times 2000 + 9 \times 1000}{9002} = 4.77
\]

mul: 9 cycles
other arith: 4 cycles
lw: 6 cycles (all hits)
branch: 3 cycles

Vector elements must be fetched from memory
Performance Factors

- Items that determine the performance of this program:
  - machine cycle time
  - memory latency (time to read a single word)
    - cache hit: ~2 cycles
    - cache miss: ~100 or more cycles
  - loop overhead
  - A simple metric: CPI (cycles per instruction)
    - if instructions are executed sequentially, CPI is a measure of instruction complexity
    - e.g. for n = 1000:
      \[
      \frac{4 \times 4001 + 3 \times 2001 + 6 \times 2000 + 9 \times 1000}{9002} = 4.77
      \]

Vector Processors

- Computer architects in the early ‘70s designed high performance machines with vector instructions
  - Example: add v1, v2, v3
    - operands are vectors, not scalars
    - pairwise addition: \(v1[i] = v2[i] + v3[i]\)
  - CPU fetches, executes one instruction, which triggers a sequence of arithmetic operations: no loop overhead
  - high throughput memory channel: no cache misses (maybe no cache)
  - Cray-1 (1977):
    - vector registers in addition to scalar registers
    - to load a vector into a register: \(ld \$vi, X\)
      - reads up to 64 words starting in memory location X

Instruction Level Parallelism

- A key to high performance on vector machines: pipelined execution
  - aka “instruction level parallelism” (ILP)
  - data pipeline: multipliers, etc break operations into independent stages

```
x
\(x[7]\) \(x[6]\) \(x[5]\) \(x[4]\) \(x[3]\)
```

```
y
\(y[7]\) \(y[6]\) \(y[5]\) \(y[4]\) \(y[3]\)
```

- instruction pipeline: fetch, decode, issue, ...

```
ops
\(add\) \(mul\) \(lw\) \(lw\) \(sll\) \(sll\)
```

```
fetch\ decode\ issue\ exec\ write
```

Superscalar Processors

- Pioneered by early supercomputer architects, ILP techniques have “trickled down” to microprocessors
  - example: IBM PowerPC 970 (Apple G5, IBM JS20 “Blade” cluster)
  - fetch up to 8 instructions from i-cache in one cycle
  - issue up to 12 instructions per cycle (2 FP, 2 int, 2 mem, ...)
    - TPP: 215 instructions in progress at one time
    - TPP: theoretical peak performance; “the performance the manufacturer guarantees you will not exceed”
  - “velocity engine” -- 128-bit SIMD processor
  - It’s a scalar architecture because the instruction set has scalar operands
  - Extensive ILP plus optimizing compiler gives performance close to that of vector processor on similar programs
**ILP and Superscalar Processors**

- When pipelines are fed the right sequence of operands, they can produce one result per clock cycle
  - one pair of products each cycle from the data pipeline
  - CPI = 1.0 in the instruction pipeline

But:
- data has to be organized properly: cache misses, pointer dereferences, address calculation all limit ILP
- instructions have to be organized properly: minimize data hazards (RA, RW, etc) and control hazards (branch delays)

```
mul $t2, $t1, $t0
add $s1, $s1, $t2
add $s0, $s0, 1
```

**Loop Unrolling**

- One way compilers order instructions for better performance is called “loop unrolling”
- In C++ an unrolled loop would look like:

  ```
  original                  unrolled once
  for (i = 0; i < n; i++) {
    ip += x[i] * y[i];
  }
  ```

- Benefits:
  - less loop overhead (branches, index updates)
  - more “raw material” for instruction scheduling to minimize delays

**Summary**

- Programs that operate on vectors (1D arrays) have regular structure
- CPUs and compilers can take advantage of regularity to provide better performance
  - instruction level parallelism
  - instruction scheduling
  - loop unrolling and other higher level “code motion” optimizations
- This is especially true when the vectors contain numeric operands -- i.e. in CSE applications
Programming Issues

- One way to improve application performance: “think vectors”
- For programs that repeatedly access the same data:
  - use collections of simple arrays
  - avoid pointers (i.e. trees, lists, other dynamic data)
- Programmers for vector processors wrote code that could be “vectorized”
  - learned rules about what sorts of loops compilers could translate into vector instructions
  - e.g. try to avoid code like
    
    ```c
    a += x[i-1] + x[i] + x[i+1];
    ```
  - FORTRAN90 has a special syntax, e.g.
    
    ```fortran
    A(1:N) = A(1:N) * B(1:N)
    ```

Vector Libraries

- Programs compiled for superscalar architectures don’t need vectorizing compilers
- But programs that lend themselves to vectorization are the same sorts of programs that can be optimized (e.g. with loop unrolling)
- Another benefit of “thinking vector”: vector libraries
- Example: Basic Linear Algebra Subroutines (BLAS)
  - originally written in FORTRAN
  - still has FORTRAN-like API
    - cryptic 6-letter names, column-major data layout, ...
  - now hand-coded in assembler
  - can be linked with C, C++, other languages

BLAS Example

- To compute the inner product of vectors x and y:
  
  ```fortran
  z = sdot(n,x,1,y,1)
  ```
  - first argument is vector size (same for x and y)
  - second, fourth are vector names
  - third, fifth are increments (update to loop index)
- Other “level 1” (vector-scalar and vector-vector routines): np
  - copy $x \leftarrow y$
  - swap $x \leftarrow y$
  - scal $x \leftarrow \alpha x$
  - axpy $y \leftarrow \alpha x + y$
- Example: multiply every element of x (vector of doubles) by 2.0:
  
  ```fortran
  daxpy(1000,2.0,X,1)
  ```

Summary

- General advice:
  - start with a method that has good asymptotic behavior
    - performance is most likely an issue for large data sets, meaning large $n$...
  - write “vectorized” or “vectorizable” code
    - for SPMD (MPI or OpenMP) applications make sure the individual processes are as efficient as possible...
  - look for ways to parallelize the best sequential algorithm
    - a “step back” to a different sequential algorithm may lead to a better final result, but it’s not the first avenue to explore...
Vectors and the N-Body Project

- One of the advanced options for Project 3 is to “vectorize” your N-Body code
  - the other option is to generalize the MPI code for more than one body per process
- The goal is to reorganize data structures so operations like computing the distance between bodies is more efficient

Hints:
- use separate 1D arrays for \(rx, ry, rz\)
- \(rx[i]\) is the x coordinate of body \(i\)
- compute the distance between body \(i\) and every other body in one set of vector operations
- after filling distance vector, use vector operations to compute acceleration, velocity

Hints (cont’d)
- Example (using FORTRAN90 notation)
  - set scalars \(bx, by, and bz\) to coordinates of body \(i\)
    \[
    dx(1:n) = rx(1:n) - bx; \\
    dy(1:n) = ry(1:n) - by; \\
    dz(1:n) = rz(1:n) - bz;
    \]
  - Problem: this method will set one of the difference values to 0, and this will cause a divide by 0 in a later step
  - Idea: overwrite \(d(i)\) with \(d(n)\), iterate \(1:n-1\) in remaining loops
- Important: avoid special cases in the body of vector loops

BLAS

- Your vectorized program should have substantial speedups in simulations of large numbers of bodies
- For even better performance (?) substitute calls to BLAS functions for some of your loops
- Where is BLAS?
  - Mac OS/X: preinstalled in 10.2 and above; “Accelerate” framework
  - IBM p690: /opt/ibmmath/essl/4.2 (?)
    - from absoft (http://www.absoft.com/Products/Libraries/essl.html)
  - CLAPACK http://www.netlib.org/blas
  - GSL GNU Scientific Library), available from GNU mirror sites
- BLAS web site (sparse): http://www.netlib.org/blas