Compiler Optimization
HPC Workshop – University of Kentucky
May 9, 2007 – May 10, 2007

Andrew Komornicki, Ph. D.
Balaji Veeraraghavan, Ph. D.
Agenda

- Code Optimization Examples
  - Outer loop unrolling
  - Inner loop unrolling
  - Stride one memory access
  - Cache blocking
  - Hardware prefetch
  - Memory store _dcbz
  - Optimization for arithmetic operations
    - Divide, SQRT, etc
Understanding Performance

- Fused floating multiply add functional units
  - Latency
  - Balance

- Memory Access
  - Stride
  - Latency
  - Bandwidth
FMA Functional Unit

- Multiply and Add
- Fused
- 6 clock period latency

\[ D = A + B \times C \]
Programming Concerns

- **Superscalar design**
  - Concurrency:
    - Branches
    - Loop control
    - Procedure calls
    - Nested “if” statements
  - Program “control” is very efficient

- **Cache based microprocessor**
  - Critical resource: memory bandwidth
    - Tactics:
      > Increase Computational Intensity
      > Exploit "prefetch"
Strategies for Optimization

- Exploit multiple functional units
- Exploit FMA capability
- Expose load/store "streams"
- Use optimized libraries
- Pipelined operations
Tactics for Optimization

- Cache reuse
  - Blocking
- Unit stride
  - Use entire loaded cache lines
- Limit range of indirect addressing
  - Sort indirect addresses
- Increase computational intensity of the loops
  - Ratio of flop's to byte's
- Load streams
Computational Intensity

- Ratio of floating point operations per memory references (loads and stores)
- Higher is better
- Example:

  ```
  do l=1,n
    A(l) = A(l) + B(l) * s + C(l)
  enddo
  ```

  - Loads and stores: 3 + 1
  - Floating point operations: 3
  - Computational intensity: 3/4
Loop Unrolling Strategy: Increase Computational Intensity

- Find variable which is constant with respect to outer loop
  - Unroll so that this variable is loaded once but used multiple times

- Unroll outer loop
  - Minimizes load/stores
Example 1: Outer Loop Unroll

- 2 flops / 2 loads
- Comp. Int.: 1

Unroll

- 8 flops / 5 Loads
- Comp. Int.: 1.6
Outer Loop Unroll Test

Theoretical peak = 1.3X4 = 5200 Gflop/s
Loop Unroll Analysis

- **Hand unroll:**
  - Unroll up to 8 times
    - Exploit 8 prefetch streams

- **Compiler unroll:**
  - Unrolls up to 4 times
    - “Near” optimal performance
  - Combines inner and outer loop unrolling

- **Goals of this example**
  - Demonstrate how compiler option works
  - Compiler can do fairly good job on loop unrolling
Inner Loop Unrolling Strategies

- **Benefit of Inner loop unroll**:  
  - Reduces data dependency  
  - Eliminate intermediate loads and stores  
  - Expose functional units  
  - Expose registers  
- **Examples**:  
  - Linear recurrences  
  - Simple loops  
    - Single operation
Example 2: Inner Loop Unroll: Turning Data Dependency into Data Reuse

- Enable data reuse
- Eliminate data dependency (half)
- Eliminate intermediate loads and stores
  - Compiler will do some of this at -O3 and higher

```plaintext
do i=2,n-1
    a(i+1) = a(i)*s1 + a(i-1)*s2
end do
```

Unroll

```plaintext
a(i) and a(i+1) used twice
```

```plaintext
do i=2,n-2,2
    a(i+1) = a(i)*s1 + a(i-1)*s2
    a(i+2) = a(i+1)*s1 + a(i)*s2
end do
```
Inner Loop Unroll: Dependencies

POWER4 1.3
Inner Loop Unroll: Dependencies

- Compiler unrolling helps
  - Does not help manually unrolled loops
    - Conflicts
- Turn off compiler unrolling if manually unrolled
Example 3: Inner Loop Unroll: Registers and Functional Units

- Expose functional units
- Expose registers
  - Compiler will do some of this at -O3 and higher

```
do j=1,n
   do i=1,m
      sum = sum + X(i)*A(i,j)
   end do
end do
```

Unroll

```
do j=1,n
   do i=1,m,2
      sum = sum + X(i)*A(i,j) & + X(i+1)*A(i+1,j)
   end do
end do
```
Inner Loop Unroll: Registers and Functional Units

POWER4 1.45
Example 4 Outer Loop Unrolling Strategies

- Expose prefetch streams
  - Up to 8 streams

```
do j=1,n,2
  do i=1,m
      sum = sum + X(i)*A(i,j) &
          + X(i)*A(i,j+1)
  end do
end do
```

```
do j=1,n
  do i=1,m
      sum = sum + X(i)*A(i,j)
  end do
end do
```

Unroll
Outer Loop Unroll: Streams

POWER4 1.45
Outer Loop Unroll: Streams

- Compiler does a “fair” job of outer loop unrolling
- Manual unrolling can outperform compiler unrolling
Summary: Turn on Loop Unrolling

- Clear performance benefit
- Consider both inner loop and outer loop unrolling
- Turn on the compiler options and let it do the work first
  - `-O3 -qhot -qunroll=yes`
  - `-qunroll=auto` is default, but it is not aggressive
- If you really want to do it by hand
  - Then you may want to turn off compiler unroll by using `-qnounroll`
New Topic: Maximize Cache Memory

- Caches are fast and more expensive memories. Let’s take full advantage of them !!

- A cache line is the minimum unit of data transfer.
Strided Memory Access

- Strided memory access uses only a portion of a cache line
  - Reduced efficiency

- Example, stride two memory access

```c
for (i=0; i<n; i+=2)
    sum+=a[i];
```
Strides

- **Power5 Cache line size is 128 bytes**
  - Double precision: 16 words
  - Single precision: 32 words

<table>
<thead>
<tr>
<th>Stride</th>
<th>Single</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>2</td>
<td>½</td>
<td>½</td>
</tr>
<tr>
<td>4</td>
<td>¼</td>
<td>¼</td>
</tr>
<tr>
<td>8</td>
<td>1/8</td>
<td>1/8</td>
</tr>
<tr>
<td>16</td>
<td>1/16</td>
<td>1/16</td>
</tr>
<tr>
<td>32</td>
<td>1/32</td>
<td>1/16</td>
</tr>
<tr>
<td>64</td>
<td>1/32</td>
<td>1/16</td>
</tr>
</tbody>
</table>

POWER4 1.3 GHz
Correcting Strides

- Interleave code

Example:
- Real and Imaginary part arithmetic

```
do i=1,n
  ... AIMAG(Z(i))
end do
do i=1,n
  ... REAL(Z(i))
end do
```

Only ½ of cache bandwidth is used

```
do i=1,n
  ... AIMAG(Z(i))
...
  ...REAL(Z(i))
end do
```

Full cache bandwidth is used
Example 5: Array Index

For optimal cache line usage, a Fortran code should make the LEFTMOST array index equal the inner most loop index. A C code should do the opposite: the RIGHTMOST array index should equal the innermost loop index.

- Perform loop Interchange

```
do i=1,n
  do j=1,m
    sum=sum+A(i,j)
  end do
end do
```

```
do j=1,m
  do i=1,n
    sum=sum+A(i,j)
  end do
end do
```
Correcting Strides: Loop Interchange

Loop interchange is easy, so let’s do it by hand
TLB - Translation Lookaside Buffer

- A buffer (or cache) in a CPU to improve the speed of virtual address translation
- Contains parts of the page table which translates from virtual into physical addresses.
- Has a fixed number of entries
- A TLB miss occurs when the page table data needed for translation is not in TLB.
  - Such translation will proceed via the page table
  - Take several more cycles to complete – particularly if the translation tables are swapped out into secondary storage
- Memory access with large stride = random memory access
  - Can easily cause TLB misses
  - Also has very low cache line reuse
Negative Effect of Large Strides - TLB Misses

POWER4 1.3 GHz
Working Set Size

- Roughly equals the memory used in a loop
- Reduce spanning size of random memory accesses
- More MPI tasks usually helps
Blocking

- Common technique used in linear algebraic operations such as BLAS
  - Similar to unrolling
  - Utilize cache lines
  - Linear Algebra
    - Typically 96-256
Blocking

Do j=1,jmax
   Do i=1,imax
      . . . .
   End do
End do

Do j=1,jmax,jblock
   Do i=1,imax,iblock
      do one block of work
      . . . .
   End do
End do
Blocking Example: Transpose

- Especially useful for bad strides

```plaintext
do i = 1,n
  do j = 1,m
    B(j,i) = A(i,j)
  end do
end do
```

Blocking:

```plaintext
do j1 = 1,n-nb+1,nb
  j2 = min(j1+nb-1,n)
do i1 = 1,m-nb+1,nb
  i2 = min(i1+nb-1,m)
do i = i1, i2
  do j = j1, j2
    B(j,i) = A(i,j)
  end do
end do
end do
```
Blocking Example: Transpose

POWER4 1.3 GHz
Hardware Prefetch

- Detects adjacent cache line references
- Forward and backward
- Up to eight concurrent streams
- Prefetches up to two lines ahead per stream
- Twelve prefetch filter queues prevents rolling
- No prefetch on store misses
  - (when a store instruction causes a cache line miss)
Prefetch: Stride Pattern Recognition

- Upon a cache miss:
- Biased guess is made as to the direction of that stream
- Guess is based upon where in the cache line the address associated with that miss occurred
- If it is in the first 3/4, then the direction is guessed as ascending
- If in the last 1/4, the direction is guessed descending
Memory Bandwidth

- **Bandwidth is proportional to number of streams**
  - Streams are roughly the number of right hand side arrays
  - Up to eight streams
Exploiting Prefetch

- **Merge Loops**
- **Strategy**
  - Combine loops to get up to 8 right hand sides

```
for (j=1; j<= n; j++)
for (j=1; j<= n; j++)
    D[j] = D[j+1]+C[j]*s
```

merge

```
for (j=1; j<= n; j++)
    { 
        D[j] = D[j+1]+C[j]*s
    }
```
Folding (Similar to Loop Unrolling)

- Fold loop to increase number of streams:

**Strategy**
- Fold loops to get up to 4 times

```
do i = 1,n
  sum = sum +A(i)
end do
```

```
do i = 1,n/4
  sum = sum +A(i)
  + A(i+1*n/4)
  + A(i+2*n/4)
  + A(i+3*n/4)
end do
```
Folding Example

POWER4 1.3 GHz
Effect of Precision

- Available floating point formats:
  - real (kind=4)
  - real (kind=8)
  - real (kind=16)

- Advantage of smaller data types:
  - Require less bandwidth
  - More effective cache use

```fortran
REAL*8 A,pi,e  
... 
do i=1,n 
  A(i) = pi*A(i) + e  
end do

REAL*4 A,pi,e  
... 
do i=1,n 
  A(i) = pi*A(i) + e  
end do
```
Effect of Precision (word size)

![Bar chart showing effect of precision on Mflop/s performance for small and large array sizes on a POWER4 1.3 GHz system. The chart compares REAL*4, REAL*8, and REAL*16 data types.]
Divide and Sqrt

- POWER special functions:
  - Divide
  - Sqrt
  - Use FMA functional unit
- 2 simultaneous divide or sqrt (or rsqrt)
  - NOT pipelined

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single Precision</td>
</tr>
<tr>
<td>Fma</td>
<td>6</td>
</tr>
<tr>
<td>Fdiv</td>
<td>22</td>
</tr>
<tr>
<td>Fsqrt</td>
<td>28</td>
</tr>
</tbody>
</table>
Hardware DIV, SQRT, RSQRT

POWER4 1.3 GHz
Intrinsic Function Vectorization
(Example: Program SPPM)

\[
\begin{align*}
\text{do} & \quad i = -\text{nbdy}+3, n+\text{nbdy}-1 \\
\text{prl} & = q\text{rprl}(i) \\
\text{pll} & = q\text{rmrl}(i) \\
\text{pavg} & = v\text{tmp1}(i) \\
\text{wllfac}(i) & = 5\ast \text{gammp1}\ast\text{pavg} + \text{gamma} \ast \text{pll} \\
\text{wrlfac}(i) & = 5\ast \text{gammp1} \ast \text{pavg} + \text{gamma} \ast \text{prl} \\
\text{hrholl} & = \text{rho}(1,i-1) \\
\text{hrhorl} & = \text{rho}(1,i) \\
\text{wll}(i) & = \frac{1}{\sqrt{\text{hrholl} \ast \text{wllfac}(i)}} \\
\text{wrl}(i) & = \frac{1}{\sqrt{\text{hrhorl} \ast \text{wrlfac}(i)}} \\
\text{end do}
\end{align*}
\]
Intrinsic Function Vectorization

allocate(t1,n+2*nbdy-3)
allocate(t2,n+2*nbdy-3)
do  i = -nbdy+3,n+nbdy-1
   prl  = qrprl(i)
...
t1(i) = hrholl * wllfac(i)
t2(i) = hrhorl * wrlfac(i)
end do
call __vrsqrt(t1,wrl,n+2*nbdy-3)
call __vrsqrt(t2,wll,n+2*nbdy-3)
Vectorization Analysis

- Dependencies
- Compiler overhead:
  - Generate (malloc) local temporary arrays
  - Extra memory traffic
- Moderate vector lengths required

<table>
<thead>
<tr>
<th>Crossover Length</th>
<th>REC</th>
<th>SQRT</th>
<th>RSQRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_{1/2}</td>
<td>45</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>80 ??</td>
<td>25 ??</td>
</tr>
</tbody>
</table>
## Function Timings

<table>
<thead>
<tr>
<th>Function</th>
<th>Hardware Clocks</th>
<th>Hardware Rate [Mop/s]</th>
<th>Pipelined Rate (Vec’ed) Clocks</th>
<th>Pipelined Rate [Mop/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recip.</td>
<td>32</td>
<td>81</td>
<td>20</td>
<td>130</td>
</tr>
<tr>
<td>SQRT</td>
<td>36</td>
<td>69</td>
<td>31</td>
<td>84</td>
</tr>
<tr>
<td>RSQRT</td>
<td>66</td>
<td>39</td>
<td>29</td>
<td>90</td>
</tr>
<tr>
<td>EXP</td>
<td>200</td>
<td>13</td>
<td>32</td>
<td>83</td>
</tr>
<tr>
<td>LOG</td>
<td>288</td>
<td>9</td>
<td>34</td>
<td>77</td>
</tr>
</tbody>
</table>
SQRT

- POWER4/POWER5 have hardware SQRT available
  - Default -qarch=com uses software library
  - Use: -qarch=pwr4
SQRT

- Hardware SQRT is 5x faster
- `-qhot` generates "vector sqrt"
**Divide**

- IEEE divide specifies actual divide
  - Do not use multiply by reciprocal (default)
  - Optimize with `-O3`

```plaintext
rs = 1/s
for i = 1 to n
  B(i) = A(i) * rs
end for
```
DIVIDE

POWER4 1.3 GHz
Vector Functions

- `-qhot` generates "vector" interface to intrinsic functions
- Monitor with "-qreport=hotlist"

```
do i=1,n
   B(i) = func(A(i))
end do
```

```
call __vfunc(B,A,n)
```
Vector Functions

POWER4 1.3 GHz
Power Function

- Computing power function: $a^{**b}$
  - if ( $b < 10$ and has integer value at compile time)
    - Use unrolling and successive multiply
  - else
    - ... __pow(...)

Test case:
do i=1,n
  A(i)=B(i)**b
end do
Power Function

POWER4 1.3 GHz
Power Function

- Compiler can transform integer power to multiply's
- Real to Real power ($r^r$) is expensive
- Real to Integer ($r^l$) is less expensive
Effect of Compiler Option Levels

POWER4 1.3 GHz
Effect of 32- and 64-bit Addressing

- 64-bit address mode enable use of 64-bit integer arithmetic
- Integer Arithmetic, especially (kind=8), is much faster with -q64

<table>
<thead>
<tr>
<th>Address Mode</th>
<th>Computation</th>
<th>Fortran</th>
<th>C, C++</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Integer*4</td>
<td>Integer*8</td>
</tr>
<tr>
<td>-q32</td>
<td>4 bytes</td>
<td>4 bytes</td>
<td>8 bytes</td>
</tr>
<tr>
<td>-q64</td>
<td>4 or 8 bytes</td>
<td>4 bytes</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>
32-bit vs. 64-bit Mode Integer Arithmetic

POWER4 1.3 GHz
Effect of 32-bit Floating Point

- Faster because of bandwidth
- Arithmetic operations are same speed as 64-bit
- More efficient use of cache
Use of Library Routines:  
Level 1

- **Single level loop constructs**
  - Examples
    - BLAS 1
    - Memory Copy
  - Prefer compiler generated code
    - Better than:
      - libc.a
        - memcpy
        - bcopy
        - memmove
      - ESSL
        - dcopy
Memory Copy Routines

POWER4 1.3 GHz
Use of Library Routines: Level 2

- Two level loop constructs
- Examples
  - Matrix Vector Multiply
    - DGEMV
- Prefer ESSL or liblapack
DGEMV

POWER4 1.3 GHz
Use of Library Routines: Level 3

- Three level loop constructs
- Examples
  - Matrix Matrix Multiply
  - DGEMM
- Prefer ESSL or liblapack
DGEMM

POWER4 1.3 GHz
Subscript Reorder

- Reference on second or higher subscript has bad strides
- Reduced performance
- Fixed by:
  - `qhot` can interchange some loops
  - `IBM* SUBSCRIPTORDER(A(2,1))`

```fortran
do i=1,n
  do j=1,m
    s = s + A(i,j)
  end do
end do
do i=1,n
  do j=1,m
    s = s + A(i,j)
  end do
end do
```
!IBM* SUBSCRIPTORDER

POWER4 1.3 GHz
Summary

- Use unrolling of outer loops
  - Compiler “might” do this
- Loop at number of prefetch streams
  - 8 is optimal
- Use –q64
  - Enables 64 bit integer arithmetic
- Use –qarch=auto
  - Enables divide and square root instructions
Questions ?