Scalar Performance Optimization

April 20, 2011
Outline

Why optimize

Instruction Flows

Pipeline Impediments

Memory Hierarchy

Basic Optimization

Branch avoidance

Loop optimization
Scalar Optimization

- Single processor performance
- Improve machine utilization
- Balance memory traffic and CPU
## Performance Evolution

### Small Test Problem

<table>
<thead>
<tr>
<th>make</th>
<th>user time</th>
<th>systime</th>
<th>wall time</th>
<th>CPU</th>
<th>time/stp/pt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sec</td>
<td></td>
<td></td>
<td>%</td>
<td>microsec</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>---------</td>
<td>-----------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>sola Opteron</td>
<td>28.75</td>
<td>0.01</td>
<td>28.76</td>
<td></td>
<td>0.492</td>
</tr>
<tr>
<td>pgf90 -Mfree -byteswapio -fastsse -Mipa=fast</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>pasha Opteron</td>
<td>41.76</td>
<td>0.01</td>
<td>41.79</td>
<td></td>
<td>0.715</td>
</tr>
<tr>
<td>pgf90 -Mfree -byteswapio -fastsse -Mipa=fast</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>ostrogoth intel</td>
<td>122.43</td>
<td>0.03</td>
<td>2:02.55</td>
<td>99.5</td>
<td>2.096</td>
</tr>
<tr>
<td>pgf90 -Mfree -byteswapio -fast -Mipa=fast</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>Ultra Sparc 5</td>
<td>274.23</td>
<td>0.04</td>
<td>4:35.52</td>
<td>99.5</td>
<td>4.695</td>
</tr>
<tr>
<td>(cp12 360 MHz) f90 -fast -native -dalign -O5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>Ultra Sparc 5</td>
<td>419.60</td>
<td>0.07</td>
<td>7:00.72</td>
<td>99.7</td>
<td>7.184</td>
</tr>
<tr>
<td>(cp12 360 MHz) f77 -fast -native -dalign -O5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>Ultra Sparc</td>
<td>568.28</td>
<td>0.07</td>
<td>9:32.44</td>
<td>99.2</td>
<td>9.730</td>
</tr>
<tr>
<td>(vasa) f77 -fast -native -dalign -O5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>make</td>
<td>user time</td>
<td>systime</td>
<td>wall time</td>
<td>CPU</td>
<td>time/stp/pt</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>---------</td>
<td>-----------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>sec</td>
<td></td>
<td></td>
<td>%</td>
<td>microsec</td>
</tr>
<tr>
<td>Alphaserver DS10</td>
<td>219.32</td>
<td>0.14</td>
<td>3:49.44</td>
<td>95.6</td>
<td>3.755</td>
</tr>
<tr>
<td>f77 -arch host -tune host -O5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerChallenge</td>
<td>248.69</td>
<td>0.81</td>
<td>4:12.74</td>
<td>98.7</td>
<td>4.257</td>
</tr>
<tr>
<td>R10000 F90 -O3 -mips4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerChallenge</td>
<td>335.00</td>
<td></td>
<td></td>
<td></td>
<td>5.735</td>
</tr>
<tr>
<td>R8000 F90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 590 (nqs)</td>
<td>405.90</td>
<td>0.10</td>
<td>12:25.00</td>
<td>54.0</td>
<td>6.949</td>
</tr>
<tr>
<td>xlf90 -O3 -qnostrict -qarch=pwrx -qhot -qalias=noaryovrlp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 590 (nqs)</td>
<td>414.00</td>
<td>0.20</td>
<td>9:21.00</td>
<td>73.8</td>
<td>7.087</td>
</tr>
<tr>
<td>xlf90 -O3 -qnostrict -qarch=pwrx -qhot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 590 (nqs)</td>
<td>579.00</td>
<td>0.01</td>
<td>21:05.00</td>
<td>45.8</td>
<td>9.912</td>
</tr>
<tr>
<td>xlf90 -O3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha 600 5/266 F90</td>
<td>660.00</td>
<td></td>
<td></td>
<td></td>
<td>11.299</td>
</tr>
<tr>
<td>System</td>
<td>Time</td>
<td>Size</td>
<td>Power</td>
<td>Temperature</td>
<td>Speedup</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>IBM 590 (nqs job)</td>
<td>727.00</td>
<td>0.02</td>
<td>22:18.00</td>
<td>54.4</td>
<td>12.446</td>
</tr>
<tr>
<td>eagle8 xlf90 O3</td>
<td>1267.00</td>
<td>0.02</td>
<td>21:31.00</td>
<td>98.3</td>
<td>21.690</td>
</tr>
<tr>
<td>IBM 560 (nqs) xlf90 -O3</td>
<td>1239.57</td>
<td>0.210</td>
<td>20:51.77</td>
<td>99.0</td>
<td>21.211</td>
</tr>
<tr>
<td>IBM 560 xlf90 O3</td>
<td>1417.73</td>
<td>0.10</td>
<td>23:50.75</td>
<td>99.0</td>
<td>24.258</td>
</tr>
<tr>
<td>IBM 560</td>
<td>1477.73</td>
<td>0.31</td>
<td>24:48.69</td>
<td>99.2</td>
<td>25.285</td>
</tr>
<tr>
<td>HP 755</td>
<td>1504.72</td>
<td>0.17</td>
<td>25:15.09</td>
<td>99.3</td>
<td>25.747</td>
</tr>
<tr>
<td>IBM 550</td>
<td>1479.72</td>
<td>0.18</td>
<td>24:45.67</td>
<td>99.6</td>
<td>25.332</td>
</tr>
<tr>
<td>IBM 550 xlf90 O3</td>
<td>1694.46</td>
<td>0.15</td>
<td>28:55.91</td>
<td>97.6</td>
<td>29.008</td>
</tr>
<tr>
<td>DEC ALPHA 3400S</td>
<td>2131.08</td>
<td>79.58</td>
<td>39:32.00</td>
<td>93.0</td>
<td>36.483</td>
</tr>
<tr>
<td>Kenai 3400</td>
<td>2181.20</td>
<td>2.16</td>
<td>36:49.00</td>
<td>98.0</td>
<td>37.337</td>
</tr>
<tr>
<td>IBM Power PC</td>
<td>2702.69</td>
<td>0.52</td>
<td>45:20.02</td>
<td>99.3</td>
<td>46.268</td>
</tr>
<tr>
<td>System</td>
<td>Time (s)</td>
<td>Throughput (GIPS)</td>
<td>Speedup</td>
<td>Efficiency</td>
<td>Speedup</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>-------------------</td>
<td>---------</td>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>SGI.Indigo.extrm</td>
<td>2923.10</td>
<td>15.60</td>
<td>49:36.00</td>
<td>98.0</td>
<td>50.041</td>
</tr>
<tr>
<td>SUN 20</td>
<td>3470.74</td>
<td>10.53</td>
<td>58:25.97</td>
<td>99.2</td>
<td>59.417</td>
</tr>
<tr>
<td>f77 -O4 -native</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUN 20</td>
<td>3761.65</td>
<td>11.47</td>
<td>1:03:12.26</td>
<td>99.4</td>
<td>64.397</td>
</tr>
<tr>
<td>f77 -O3 -native</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP 712</td>
<td>4113.90</td>
<td>0.30</td>
<td>1:08:38.00</td>
<td>99.0</td>
<td>70.412</td>
</tr>
<tr>
<td>Stardent vector</td>
<td>5130.07</td>
<td>16.43</td>
<td>1:25:54.03</td>
<td>99.8</td>
<td>87.822</td>
</tr>
<tr>
<td>Sun 10/41</td>
<td>5681.90</td>
<td>0.60</td>
<td>1:35:12.00</td>
<td>99.0</td>
<td>97.255</td>
</tr>
<tr>
<td>Cray YMP EL-92</td>
<td>6384.12</td>
<td>254.75</td>
<td>3:24:14.00</td>
<td>52.1</td>
<td>109.290</td>
</tr>
<tr>
<td>Sun 690</td>
<td>12676.38</td>
<td>43.16</td>
<td>3:32:04.72</td>
<td>99.9</td>
<td>217.004</td>
</tr>
<tr>
<td>Stardent scalar</td>
<td>14939.34</td>
<td>64.95</td>
<td>4:10:12.67</td>
<td>99.9</td>
<td>255.745</td>
</tr>
</tbody>
</table>
## Performance Evolution

### Large Test Problem

<table>
<thead>
<tr>
<th>make</th>
<th>user time</th>
<th>systime</th>
<th>wall time</th>
<th>CPU</th>
<th>time/stp/pt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(sec)</td>
<td></td>
<td></td>
<td></td>
<td>(microsec)</td>
</tr>
<tr>
<td>sola Opteron</td>
<td>25.73</td>
<td>0.07</td>
<td>0:25.80</td>
<td></td>
<td>0.560</td>
</tr>
<tr>
<td>pgf90 -fastsse -Mipa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pasha Opteron</td>
<td>39.20</td>
<td>0.09</td>
<td>0:39.09</td>
<td></td>
<td>0.854</td>
</tr>
<tr>
<td>pgf90 -fastsse -Mipa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGI PChal R10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f90 -O3 -mips4</td>
<td>332.34</td>
<td>2.26</td>
<td>5:50.10</td>
<td>95.5</td>
<td>7.241</td>
</tr>
<tr>
<td>SGI PowerChal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f90 -O3</td>
<td>383.99</td>
<td>2.23</td>
<td>6:27.96</td>
<td></td>
<td>8.367</td>
</tr>
<tr>
<td>IBM 590 (nqs)</td>
<td>348.40</td>
<td>0.50</td>
<td>11:43.00</td>
<td>49.0</td>
<td>7.591</td>
</tr>
<tr>
<td>-O3 -qnostrict -qarch=pwrx -qhot -qalias=naoryovrlp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 590 (nqs)</td>
<td>387.60</td>
<td>0.60</td>
<td>12:40.00</td>
<td>51.0</td>
<td>8.445</td>
</tr>
<tr>
<td>xlf90 -O3 -qnostrict -qarch=pwrx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 590 (nqs)</td>
<td>448.00</td>
<td>0.60</td>
<td>8:30.00</td>
<td>87.0</td>
<td>9.761</td>
</tr>
<tr>
<td>xlf90 -O3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 560 (nqs)</td>
<td>1166.53</td>
<td>1.40</td>
<td>42:42.33</td>
<td>45.5</td>
<td>25.405</td>
</tr>
<tr>
<td>xlf90 -O3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 550 (nqs)</td>
<td>1442.29</td>
<td>2.24</td>
<td>24:37.00</td>
<td>96.2</td>
<td>31.425</td>
</tr>
<tr>
<td>xlf90 -O3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Optimization Strategies

1. Start with working code
2. Identify performance bottle necks
   Use profiling tools
3. Improve CPU-intensive parts
   3.1 Experiment with compiler options
   3.2 Replace with optimized library
   3.3 Rewrite code
   3.4 Replace algorithm
4. Check new code for validity and performance
Why optimize Instruction Flows Pipeline Impediments Memory Hierarchy Basic Optimization Branch avoidance Loop optimization

**Stored-Program Computer Architecture**

- Instructions and Data are numbers stored in Memory
- Control Unit reads and executes instructions
- Computations are done in Arithmetic & Logic Units
- I/O facilities enable communication with users
- Arrows indicate data & instruction traffic between units
- Performance depends on units and traffic speeds

**Figure:** Conceptual sketch of stored-program computer architecture. The program and data are stored in memory and are fed to the CPU.
Typical Instruction Flow

Fetch | Decode | Execute | WriteBack

Clock Cycle

1 Instruction every 4 cycles, IPC = 0.25
Pipeline Execution

- Overlap Stages like assembly line
- 1 Instructions every cycle, IPC = 1.0
Superscalar Execution

- EX uses independent functional units
- Overlap multiple EX stages
  - Instruction and decode process multiple instructions
  - These are dispatched to execution units
  - Processed in parallel
  - Instruction-level parallelism
Superscalar Execution

Clock Cycle

- Execute multiple instructions per cycle
- 3 Instructions every cycle, IPC = 3.0
Vector Processing
Single Instruction Multiple Data: SIMD

- Consider the code

```plaintext
real a(1000), b(1000), c(1000)
do i = 1, 1000
   a(i) = b(i) + s * c(i)
endo
```

- Simple operation applied to multiple independent data
- Operate on groups of data at a time
- memory bandwidth is limiting factor
integer, parameter :: N=4000000
real*8 :: a(N), b(N), c(N), d(N), time, MFLOPS, MBAND
integer:: tick1, TickStart, TickEnd, i

call TimingInit(tick1) ! initialize counters
call TimingPrintClockInfo(6) ! print counter info
a=0.d0; b=1.d0; c=2.d0; d=3.d0 ! initialize arrays
call system_clock(COUNT=TickStart) ! record start tick
do i = 1,N
    a(i) = b(i)+c(i)*d(i) ! 2 flops, 4 memory transfers
endo
call system_clock(COUNT=TickEnd) ! record end tick
time = TimingGetWallTime(TickEnd, TickStart) ! get time in seconds
MFLOPS = (2.d0*N/time)*1.d-6 ! MegaFLOPs/sec

● Compilation: pgf90 -fastsse -O4 -Mipa=fast,safe
● -fastsse Enable vector SIMD instructions
● -O4 Optimization level 4
● -Mipa Enable interprocedural analysis
Figure: Measured MFLOPS versus array size on a Intel Xeon CPU E5620 @ 2.40GHz Speed: 1,600.00 MHz, 6 CPUs and 11.7 GB of RAM. The cache is 32 KB L1 Instruction cache, 32 KB L1 Data cache, 256KB L2 cache, and 12288KB L3 cache.

- Performance varies with array size $N$, but generally drops
- Peak is 2.7 GFLOPS for $N \sim 10^3$
- Sustained is 0.7 GFLOPS $N > 10^6$
Instruction Dependencies & Latencies
Impediments to Optimal Performance

• Consider following code

\[ a = b \times c \]
\[ d = a + 1 \]

• \( d \) depends on \( a \)
• \( d \) cannot execute in parallel with \( a \)
• Pipeline stalls until \( a \) completes.
• Latency: time from execution stage till WB completes
• Deeper pipelines lengthens the latency
• Worst latency associated with memory load/store
Branches & Branch Prediction

Branches

- Consider the following code

```plaintext
if (a > 5) {
  b = c;
} else {
  b = d;
}
end
```

- Pipeline fetches I3 and I4 when I2 reach EX
- But I3 and I4 may not be needed depending on `cmp`
- Possible that I5 and I6 are needed instructions
- Pipeline stalls if CPU waits for result
- Don’t wait take, guess branch and cross fingers
Branches & Branch Prediction

Branch Prediction (Guessing)

- Compiler decides on prediction (static)
- CPU guesses at run time
  - "smart" compiler
    - CPU keeps track of most common branch
    - takes extra resources
- 90% accurate prediction still result in 30% loss
Branches & Branch Prediction

Eliminating Branches with Predication

\begin{align*}
\text{cmp } a,5 & \quad ; \ a > 5 \ ? \\
\text{mov } c,b & \quad ; \ b = c \ ? \\
\text{cmovle } d,b & \quad ; \ \text{if le, then } b = d
\end{align*}

- Initially set \( b=c \)
- Instructions 1 and 2 independent (parallel exec)
- New instruction: conditional move: \texttt{cmovle}
- Result of Inst 1 ready for Inst 3.
- Pipeline does not stall
- works well if branches are small
Why optimize

Instruction Flows

Pipeline Impediments

Memory Hierarchy

Basic Optimization

Branch avoidance

Loop optimization

Improve CPU utilization

- branches and long latency instructions stall pipeline
- re-order program flow to improve CPU utilization: OOO
  - data analysis for dependencies
  - register renaming
  - instruction rescheduling
  - increases complexity of chip if done in hardware
- re-order at compile time
- Compiler can analyze larger code sections
- Need both hardware and compiler capabilities to be effective
Memory Latency

Widening gap between processor and memory

![Graph showing the widening gap between CPU clock speed and DRAM speed](image-url)
Memory Hierarchy

Non-uniform memory access

- Memory classes

<table>
<thead>
<tr>
<th>Type</th>
<th>location</th>
<th>access time</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>on chip</td>
<td>O(1) ns</td>
<td>B</td>
</tr>
<tr>
<td>L1 cache</td>
<td>on chip</td>
<td>O(1) ns</td>
<td>KB</td>
</tr>
<tr>
<td>L2 cache (SRAM)</td>
<td>off chip</td>
<td>O(10) ns</td>
<td>MB</td>
</tr>
<tr>
<td>main (DRAM)</td>
<td>off chip</td>
<td>O(100) ns</td>
<td>GB</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>off chip</td>
<td>O(10) ms</td>
<td>TB</td>
</tr>
</tbody>
</table>

- Hit rate: percentage memory reference is in cache
- L1: 75% (10 ns), L2: 20% (30 ns) and main 5% (300 ns)
- Average Performance
  \[0.75 \times 10 + 0.20 \times 30 + 0.05 \times 300 = 28.5 \text{ ns}\]
Cache Organization

Cache lines

Main Memory

Address
Cache Organization

1. Caches exploit time and space locality
   - time: likely to reuse data that was just accessed
   - space: likely to need nearby data
2. Keep track of data in cache
3. Simplify address translation for fast access
Cache Organization

1. Direct mapped cache allows chunks of memory in only 1 cache line
2. subject to cache thrashing if two variables occupy same cache line
3. set associative cache allow memory to occupy multiple cache line
4. more flexible (less thrashing) but slower because of slightly more complicated address calculation
Constant propagation

program cnst
  integer, parameter :: M=100,N=0
  integer :: L=200, MM=M+1, J,S
  J = L+M
  S = N*M
  stop
end program cnst

• Compute constants at compile (and not) run time.
• Use parameter attribute in Fortran to help compiler
• Declare \( L \) and \( M \) as parameters if value does not change
program deadcode
  integer, parameter :: M=100
if (M < 100) then
  call SubA(M)
else
  call SubB(M)
endif
stop
end program deadcode

- Subroutine SubB will never be executed
- Eliminate it from executable to shrink its size
- Rely on Loader/Compiler to do it.
Strength Reduction

Replace some operations with cheaper alternatives

\[
fc = \frac{1.0}{6.0}
\]

do i = 1, 200
  a(i) = fc * a(i) ! cheaper than \( a(i) = a(i)/6.0 \)
endo

Eliminate common subexpressions \( \tanh r = \frac{e^r - e^{-r}}{e^r + e^{-r}} = \frac{1-e^{-2r}}{1+e^{-2r}} : \)

for (i=0; i<200; i++){
  r = 2*i;
  ep = exp(-2.0*r); /* save exponential */
  th = (1.0-ep)/(1.0+ep) /* tanh function */
}
Avoid needless type conversion

The constant 2 has to be promoted to a real first

```fortran
real  a(N)
do i = 1,N
  a(i) = 2*a(i)
enddo
```

Use proper variable type

```fortran
do i = 1,N
  a(i) = 2.0*a(i)
enddo
```
Reduce call overhead

do i = 1,200
  call mult(a(i), b(i), c(i))
enddo

subroutine mult(a,b,c)
  c = a*b
end subroutine mult

Manually in-line small subroutines called frequently

  c(1:200) = a(1:200) * b(1:200)

Better yet use compiler to in-line procedure

pgf90 -Minline=mult prog.f90
pgcc -Mautoinline prog.c
Loop index-dependent conditionals

do i = 1,N
    if (mod(i,2)==0) then
        a(i) = funcA(i,b,c)
    else
        a(i) = funcB(i,b,c)
    endif
enddo

Replace with

do i = 1,N,2
    a(i) = funcA(i,b,c)
enddo

endo

do i = 2,N,2
    a(i) = funcB(i,b,c)
enddo
Loop independent conditionals

do i = 1,K
  if (a == 0.0) then
    f(i) = 1.0
  else
    f(i) = sin(a*i)/(a*i)
  endif
enddo

Replace with

if (a == 0.0) then
  f = 1.0
else
  do i = 1,K
    f(i) = sin(a*i)/(a*i)
  enddo
endif
Motivation

- Most intensive portion of the calculations
- Analysis to determine mix of operations
  - load/store arithmetic
  - branches
  - floating point operations
  - address calculations
- Maximize independent calculations
- Minimize memory traffic
Memory access pattern

- Step consecutively through memory for efficient cache use
- Matrices are column major-order in fortran
  
  ```fortran
  do j = 1,N
    do i = 1,M
      a(i,j) = a(i,j) + c*b(i,j)
    enddo
  enddo
  ```

- Matrices are row major-order in C
  
  ```c
  for (i=0; i<M; i++) {
    for (j=0; i<N; j++) {
      a[i][j] = a[i][j] + c*b[i][j];
    }
  }
  ```
Memory access pattern

- Matrix multiplication tough on cache

```plaintext
do j = 1,N  
do i = 1,N  
  sum = 0.0  
do k = 1,N  
    sum = sum + A(i,k) * B(k,j)  
  enddo  
  C(i,j) = sum  
enddo  
enddo
```

- There are $N^3$ references to $A(i,k)$
Memory access pattern

- Rewrite as
  
  \[
  C(1:N, 1:N) = 0.0 \\
  \text{do } k = 1,N \\
  \quad \text{do } j = 1,N \\
  \quad \quad \text{scale} = B(k,j) \quad ! \quad N^2 \text{ non-unit stride} \\
  \quad \text{do } i = 1,N \\
  \quad \quad C(i,j) = C(i,j) + A(i,k) \times \text{scale} \\
  \quad \text{enddo} \\
  \text{enddo} \\
  \text{enddo}
  \]

- Use optimized vendor supplied libraries for optimal speed
- Algorithms use blocking so that portions of A,B, and C fit in cache
unroll loops to uncover potential parallel computations

\[
\text{do } j = 1, N
\]
\[
C(j) = C(j) + A(j) \times \text{scale}
\]
\[
C(j+1) = C(j+1) + A(j+1) \times \text{scale}
\]
\[
C(j+2) = C(j+2) + A(j+2) \times \text{scale}
\]
\[
C(j+3) = C(j+3) + A(j+3) \times \text{scale}
\]
\[\text{enddo}\]

predictable program flow
compiler analyzes cost/benefit
“fat” loops are not unrolled
compiler cannot unroll loops with subroutine/function calls
References


• Read compiler manual