Optimising for the p690 memory system
Introduction

• As with all performance optimisation it is important to understand what is limiting the performance of a code.

• The Power4 is a very powerful micro-processor with a very high peak performance.

• In common with many modern processors the memory system can be a significant bottleneck
  - either memory bandwidth (bytes per second)
  - or latency (time to read a single value)
Memory

• Memory structures are typically completely defined by the programmer.
  - at best compilers can add small amounts of padding.
  - any performance impact from memory structures has to be addressed by the programmer or the hardware designer.

• With current hardware, memory access has become the most significant resource impacting program performance.
  - changing memory structures can have a big impact on code performance.
• Memory structures are typically global to the program
  - different code sections communicate via memory structures.
  - the programming cost of changing a memory structure can be very high.
  - for example, even changing the index order of arrays is tedious and error prone.
What can go wrong

• Poor cache/page use
  - lack of spatial locality
  - lack of temporal locality
  - cache thrashing

• Unnecessary memory accesses
  - pointer chasing
  - array temporaries

• Aliasing problems
  - Use of pointers can inhibit code optimisation
Reducing memory accesses

- Memory accesses are often the most important limiting factor for code performance.
  - many older codes were written when memory access was relatively cheap.
- Some things to look for:
  - unnecessary pointer chasing
    - pointer arrays that could be simple arrays
    - linked lists that could be arrays.
  - unnecessary temporary arrays.
  - cache conflicts
  - long loops without much work in them
  - tables of values that would be cheap to re-calculate.
Principal of locality

Cache based architectures rely heavily on locality:

Spatial locality
- Items with nearby addresses tend to be accessed close together in time.
  - e.g. if \( y(i) \) is read now, \( y(i+1) \) is likely to be read soon.

Temporal locality
- A recently accessed item is likely to be reused in the near future.
  - e.g. if \( x \) is read now, it is likely to be read again, or written, soon.
Improving spatial locality

- **Stride-1 accesses through arrays**
  - compiler may fix simple cases, but don’t rely on it:

**Classic example:**

```fortran
    do j=1,n
        do i=1,m
            a(j,i) = b(j,i) + c(j,i)
        end do
    end do
```
• Use of structures may improve spatial locality
  - place items which are accessed in the same pieces of code close to each other
    • also have to worry about alignment with cache block boundaries.
  - remove infrequently used fields and put them somewhere else.
• Avoid “gaps” in structures
  - structures may contain gaps to ensure the address of each variable is aligned with its size.
  - place items of same size next to each other
Spatial locality isn’t enough

- Spatial locality on its own does not give high performance

Example: STREAMS triad
\[ a(i) = b(i) + d * c(i) \]

- 8 Gbyte/s bandwidth on 8 processors (if arrays don’t fit into L3)
- 2 flops requires 24 bytes of data
- 83 Mflop/s per processor
- less than 2% of peak performance
  - increases to 3-4% of peak for data in L3
Loop merging

- Old vector code often had many simple loops with intermediate results in temporary arrays

```fortran
REAL V(1024,3), S(1024), U(3)
DO I=1,1024
   S(I) = U(1)*V(I,1)
END DO
DO I=1,1024
   S(I) = S(I) + U(2)*V(I,2)
END DO
DO I=1,1024
   S(I) = S(I) + U(3)*V(I,3)
END DO
DO J=1,3
   DO I=1,1024
      V(I,J) = S(I) * U(J)
   END DO
END DO
```
• Can merge loops and use a scalar

```fortran
REAL V(1024,3), S, U(3)
DO I=1,1024
    S = U(1)*V(I,1) + U(2)*V(I,2) + U(3)*V(I,3)
    DO J=1,3
        V(I,J) = S * U(J)
    END DO
END DO
```

• Vector compilers are good at turning scalars into vector temporaries but the reverse operation is hard

• Loop merging useful technique to improve temporal locality
  - compiler may do simple cases
  - *but* very large loop bodies may result in register pressure
Cache blocking

- Once data is loaded into cache, we want to do as much as possible with it before it gets ejected.
- Can be useful in cases with non-unit stride as well as cases with genuine reuse

```fortran
do j=1,n
  do i=1,n
    s = s + a(j,i) + b(i,j)
  end do
end do
```

```fortran
do ii=1,n,nb
  do j=1,n
    do i=ii,ii+nb-1
      s = s + a(j,i) + b(i,j)
    end do
  end do
end do
```
• Can consider doing this at quite a high level in the application
  - e.g. at the domain decomposition level
  - divide into smaller domains so that each one fits in cache

• Predicting the correct block size is difficult
  - best determined by experiment
  - expose tunable parameter(s) for maximum convenience.
p690 caches

• Recall that the p690 has 3 levels of cache
  - Separate L1 data/instruction cache
  - Unified L2 shared between the 2 cpus on a chip
  - Global L3 cache
L1 Instruction cache

• 64Kb direct mapped cache
  - Addresses that are 64Kb apart map to the same location in cache.

• Very few issues with using the I-cache effectively
  - Most HPC codes are dominated by a small number of loops.

• Loop bodies must not exceed 64Kb
  - this is rarely a problem

• Sometimes may have problems if a loop calls a subroutine that maps to same location as loop body.
  - Fixed by in-lining subroutine.
L1 Performance issues

- 32Kb 2-way set associative, 128-byte lines, 4-5 cycle memory latency.
- This is very small for a data cache.
  - most arrays will be bigger than this
  - need to re-use data within the inner loop.
- Addresses from cache lines 16K apart may cause cache conflicts
  - the 2 sets help somewhat but not if more than 2 competing values.
- Note only one store per cycle.
  - FPU can perform 2 multiply-add operations per cycle: easy to become store bound.
• Compiler should try to schedule loads to hide the L1 cache latency
  - cannot always do this if there are too many dependencies between instructions.
  - loop unrolling and software pipelining is vital if this is going to work.
Conflict example

• Example:

```fortran
REAL A(1024), B(1024), C(1024), X
COMMON /DAT/ A,B,C ! Contiguous
DO I=1,1024
   X = X + A(I) * B(I) + C(I)
END DO
```

• Corresponding array elements all map to the same set so each access causes a cache miss.
  - 3 elements but only 2 blocks in the set
Conflicts

- If you have a cache conflict problem you can:
  - avoid arrays with power-of-2 inner dimension
  - insert padding to remove the conflict
  - change the loop order
  - permute index order in array (global edit).
L2 cache

- 1440 Kb, 8-way set associative, 128 bytes lines
- shared by both CPUs on the chip.

- For an L2 cache, this is rather small and has a long latency (14-20 cycles)
- Conflicts are not a problem
- Block algorithms should block to fit in L2 not L1.
  - L1 is too small: overheads of short loops outweigh improved locality
  - remember the cache is shared: don’t tune for block size in a parallel code on one processor!
Array initialisation

- Large array initializations may be particularly slow when using write allocate caches.
  - only want to perform lots of writes to overwrite junk data.
  - cache will carefully load all the junk data before overwriting it.
  - especially nasty if the array is sized generously but everything is initialized

- Workarounds
  - use special features to zero the array (compiler directive).
  - combine initialization with the first access loop
    - increases the chance of a programming error so have a debugging options to perform original initialization as well
• L2 cache allocates on store operations.
  - where you have the choice, overwrite an input rather than introduce an additional array.
  - try not to initialise array in separate loop.
    • reset elements to zero just after last use
  - use CACHE_ZERO directive if you are sure array not cached.

```fortran
  do j=1,n,32
    !IBM* CACHE_ZERO (a(j))
    do i=j,j+31
      a(i) = 0
    enddo
  end do
```
• Note: `CACHE_ZERO` will zero everything on the cache line: need to take care at the beginning and end of arrays.

• Can also spread store operations across the 3 cache controllers by interleaving writes to 3 consecutive cache lines:

```fortran
  do j=1,n,48
    do i=j,j+15
      a(i) = b
      a(i+16) = b
      a(i+32) = b
    enddo
  enddo
end do
```
L3 cache

- 128Mb per MCM, 8-way set-associative.
- 100+ cycles latency

- Seems huge, but it’s only 16Mb per CPU
- Latency is very high
- Not all L3 misses result in the data being loaded to L3
  - benchmarks see a gradual performance degradation as working set approaches L3 size!
L3 performance issues

- L3 cache especially good when not all CPUs are in use
  - OpenMP master regions
  - Single CPU benchmarks.
- This is because all CPUs can access all of the L3 cache.
- L3 does improve performance but there is not much you can do to tune for it.
  - difficult to block for, as there is no sharp cutoff in performance
- Again, beware of tuning parallel code on 1 CPU.
TLB

- TLB misses are expensive
  - need to access the OS page tables
  - loop fusion and blocking also help to reduce TLB misses
  - L2 is smaller than TLB span, so optimising for L2 will take care of most TLB problems as well.
Detecting cache problems

- hpmcount is your friend!

- can tell you how much data was loaded from each level of cache

- miss rates can be misleading: a high miss rate is OK if the number of misses is small
Prefetching

- The processor will stall when:
  - an instruction attempts to use the result of the load that caused the cache miss.
  - more than 8 cache misses are outstanding

- Prefetching tries to avoid stalls by loading the caches ahead of time
Using streams

- To utilize stream hardware use linear access patterns where possible
  - only the order of cache block accesses needs to be linear, not each word access.
- Most loops will require multiple streams
  - if the loop requires more than 8 streams then some data will not be prefetched.
  - consider splitting the loop.
  - if a loop has fewer than 8 streams, then loop merging or bisection can be used to increase the number of streams.
31 March, 2004

Optimising for the p690 memory system

\[
\begin{align*}
\text{do } i &= 1, n/2 \\
& \text{s1 = s1 + a(i)*b(i)} \\
& \text{s2 = s2 + a(i+n/2)*b(i+n/2)} \\
\text{end do} \\
\text{do } i &= 1, n \\
& \text{s = s + a(i)*b(i)} \\
\text{end do} \\
& \text{s = s1+s2}
\end{align*}
\]

- No prefetch on writes
- Prefetching is most useful when not all the CPUs are active at the same time.
  - otherwise memory bandwidth is saturated anyway.
  - single CPU tuning can be misleading
Pointers

• Pointers are useful but can seriously inhibit code performance.
• Compilers try very hard to reduce memory accesses.
  – Only loading data from memory once.
  – Keep variables in registers and only update memory copy when necessary.
• Pointers could point anywhere, to be safe:
  – Reload all values after write through pointer
  – Synchronize all variables with memory before read through pointer
Pointers and Fortran

- F77 had no pointers
- Arguments passed by reference (address)
  - Subroutine arguments are effectively pointers
  - But it is illegal Fortran if two arguments overlap. IBM compilers seem sensitive to this.
- F90/F95 has restricted pointers
  - Pointers can only point at variables declared as a “target” or at the target of another pointer
  - Compiler therefore knows more about possible aliasing problems
- Try to avoid F90 pointers for performance critical data structures.
Pointers and C

• In C pointers are unrestricted
  - can therefore seriously inhibit performance

• Almost impossible to do without pointers
  - malloc requires the use of pointers.
  - pointers used for call by reference. Alternative is call by value where all data is copied!

• Explicit use of scalar temporaries may reduce the problem
Summary

• The three most important things to remember when tuning for the p690 are:

   **LOCALITY, LOCALITY, LOCALITY**

• It’s even more important than on previous generation machines

• Tune parallel codes on 8 processors, not 1