

# Optimising with the IBM compilers



- Introduction
- Optimisation techniques
  - compiler flags
  - compiler hints
  - code modifications
- Optimisation topics
  - locals and globals
  - conditionals
  - data types
  - CSE
  - divides and square roots
  - register use and spilling
  - loop unrolling/pipelining

- Unless we write assembly code, we are always using a compiler.
- Modern compilers are (quite) good at optimisation
  - memory optimisations are an exception
- Usually much better to get the compiler to do the optimisation.
  - avoids machine-specific coding
  - compilers break codes much less often than humans
- Even modifying code can be thought of as "helping the compiler".

- Typical compiler has hundreds of flags/options.
  - most are never used
  - many are not related to optimisation
- Most compilers have flags for different levels of general optimisation.
  - `-O1`, `-O2`, `-O3`, ....
- When first porting code, switch optimisation off.
  - only when you are satisfied that the code works, turn optimisation on, and test again.
  - but don't forget to use them!
  - also don't forget to turn off debugging, bounds checking and profiling flags...

- `-O0` or `-qnoopt` No optimisation
- There is no `-O1`
- `-O2`
  - low level optimisation
  - e.g. redundant code elimination, invariant code removal from loops, some loop unrolling + pipelining, ....
  - consider using `-qmaxmem=-1` to allows more memory for space intensive optimisations
    - This is the default for optimisation > 2
- `-O3`
  - more extensive optimisation
  - e.g some floating point reordering, deeper loop unrolling, ...
  - consider using `-qstrict` to avoid numerical differences

- **-O4** includes all the optimisations of **-O3**, plus:
  - **-qhot** (for Fortran), **-qipa**, **-qarch=auto**,  
**-qtune=auto**, **-qcache=auto**
- **-qhot**
  - **Higher Order Transformations**
  - **optimises F90 array syntax statements**
  - **changes loop nests to improve cache behaviour**
  - **allows user to**
    - **transform loops to vector library calls (MASS)**
    - **introduce array padding**
  - **syntax**
    - **-qhot[=[no]vector | arraypad[=n]]**

- **-qipa**
  - expands the scope of optimisation to an entire program unit, allows inlining
  - `-qipa[=level=n | inline=name | fine tuning]`
- **-O5**
  - includes all of -O4 optimisations plus: `-qipa=level=2`
- **Target machine options**
  - specifies a particular processor and memory system
    - `-qarch`: restricts compiler to a particular instruction set
    - `-qtune`: optimisation based on a particular machine
    - `-qcache`: defines a particular cache / memory geometry
    - we are compiling on Power4s, so defaults should be fine.

- Enabled with `-qipa` option
  - can also use `-Q` for routines in the same compilation unit
- Not always beneficial: may help to be selective
  - can list functions to be inlined
  - also makes profiling harder....
- Very important for OO code
  - OO design encourages methods with very small bodies
  - `inline` keyword in C++ can be used as a hint

- The compiler has to make conservative assumptions about aliasing
  - storage associated data in Fortran
  - pointers in C (and Fortran)
- You can tell the compiler that aliasing is not present in a program unit with the `-qalias` flag.
  - various options for different types and levels of aliasing
  - see man pages and online documentation.

- Note that highest levels of optimisation may
  - cause your code to break.
    - might be a compiler bug, but more likely because your code is not strictly standard conforming
    - e.g. passing same argument twice
  - make your code go slower
    - -O5 is not always best.
- Isolate routines and flags which cause the problem.
  - binary chop
  - try high levels on key routines only
  - one routine per file may help

- A mechanism for giving additional information to the compiler, e.g.
  - values of variables (e.g. loop trip counts)
  - independence of loop iterations
  - independence of index array elements
  - aliasing properties
- Appear as comments (Fortran), or preprocessor pragmas (C)
  - don't affect portability
- IBM hints are only taken into account if `-qhot` or `-qsmp` (autoparallelization) is used.
  - not much use for sequential C/C++ code

- Gives compiler extra information about a loop
  - trip count
  - independence of iterations

```
!IBM* ASSERT (NODEPS,ITERCNT(50))  
DO I=1,N  
    A(I) = A(I) + FUNC(I)  
END DO
```

- Tells the compiler that no procedure called in the loop has a loop carried dependency
  - weaker than `ASSERT(NODEPS)` as the loop itself may still have dependencies

```
! IBM* CNCALL  
DO I=2,N  
    A(I) = A(I) + A(I-1) + FUNC(I)  
END DO
```

- Tells the compiler that an array contains no repeated values in the scope of a loop

```
! IBM* PERMUTATION (IDX)  
DO I=1,N  
    A(IDX(I)) = A(IDX(I)) + B(I)  
END DO
```

- When flags and hints don't solve the problem, we will have to resort to code modification.
- Be aware that this may
  - introduce bugs.
  - make the code harder to read/maintain.
  - only be effective on certain architectures.
- Try to think about
  - what optimisation the compiler is failing to do
  - how can rewriting help

- How can we work out what the compiler has done?
  - eyeball assembly code
  - use diagnostics flags
- Increasingly difficult to work out what actually occurred in the processor.
  - superscalar, out-of-order, speculative execution
- Can estimate expected performance
  - count flops, load/stores, estimate cache misses
  - compare actual performance with expectations

- To dump assembly code, compile with `-s` option
  - produces `.s` file
- To produce a loop optimisation report (from `-qhot`), compile with `-qreport`
  - produces `.lst` file
  - contains transformed source code, showing unrolled loops, address computations, array syntax expanded into loops
    - not very pretty!
  - notes on loop transformations (e.g. fusion, unrolling)

- Compiler analysis is more effective with local variables
- Has to make worst case assumptions about global variables
- Globals could be modified by any called procedure (or by another thread).
- Use local variables where possible
- Automatic variables are stack allocated: allocation is essentially free.
- In C, use file scope globals in preference to externals

- Even with sophisticated branch prediction hardware, branches are bad for performance.
- If you can't eliminate them, at least try to get them out of the critical loops.
- Simple example:

```
do i=1,k
  if (n .eq. 0) then
    a(i) = b(i) + c
  else
    a(i) = 0.
  endif
end do
```



```
if (n .eq. 0) then
  do i=1,k
    a(i) = b(i) + c
  end do
else
  do i=1,k
    a(i) = 0.
  end do
endif
```

- A little harder for the compiler.....

```
do i=1,k
  if (i .le. j) then
    a(i) = b(i) + c
  else
    a(i) = 0.
  endif
end do
```



```
do i=1,j
  a(i) = b(i) + c
end do
do i = j+1,k
  a(i) = 0.
end do
```

- Performance can be affected by choice of data types
  - complicated by trade-offs with memory usage and cache hit rates
  - avoid `INTEGER*1`, `INTEGER*2` and `INTEGER*8` (unless compiling with `-q64`)
- Avoid unnecessary type conversions
  - especially integer to floating point and vice-versa
  - N.B. some type conversions are implicit

- Compilers are generally good at Common Subexpression Elimination.
- A couple of cases where they might have trouble:

## Different order of operands

```
d = a + c
e = a + b + c
```

## Function calls

```
d = a + func(c)
e = b + func(c)
```

- Divides and square roots are expensive
  - not pipelined
  - divide takes 30 cycles
  - sqrt takes 36 cycles
  - can use both FPUs in parallel
- Can often reduce divide count by storing reciprocals
- Use of look-up tables is a trade-off between computation and memory access time.

- Most compilers make a reasonable job of register allocation.
- Can have problems in some cases:
  - loops with large numbers of temporary variables
    - such loops may be produced by inlining or unrolling
  - array elements with complex index expressions
    - can help compiler by introducing explicit scalar temporaries

```
for (i=0;i<n;i++){  
    b[i] += a[c[i]];  
    c[i+1] = 2*i;  
}
```

```
tmp = c[0];  
for (i=0;i<n;i++){  
    b[i] += a[tmp];  
    tmp = 2*i;  
    c[i+1] = tmp;  
}
```

- If compiler runs out of registers it will generate spill code.
  - store a value and then reload it later on
- Examine your source code and count how many loads/stores are required
- Compare with assembly code
- May need to distribute loops

- Loop unrolling and software pipelining are two of the most important optimisations for scientific codes on modern RISC processors.
- Compilers generally good at this.
- If compiler fails, usually better to try and remove the impediment, rather than unroll by hand.
  - cleaner, more portable, better performance

- Replace loop body by multiple copies of the body
- Modify loop control
  - take care of arbitrary loop bounds

Example:

```
do i=1,n
  a(i)=b(i)+d*c(i)
end do
```



```
do i=1,n-3,4
  a(i)=b(i)+d*c(i)
  a(i+1)=b(i+1)+d*c(i+1)
  a(i+2)=b(i+2)+d*c(i+2)
  a(i+3)=b(i+3)+d*c(i+3)
end do
do j = i,n
  a(j)=b(j)+d*c(j)
end do
```

```
for (i=0;i<n;i++){  
    a(i) += b;  
}
```



```
for (i=0;i<n;i++){  
    t1 = a(i);    //L i  
    t2 = b + t1; //A i  
    a(i) = t2;   //S i  
}
```



```
//prologue  
t1 = a(0);    //L 0  
t2 = b + t1;  //A 0  
t1 = a(1);    //L 1  
  
for (i=0;i<n-2;i++){  
    a(i) = t2;    //S i  
    t2 = b + t1; //A i+1  
    t1 = a(i+2); //L i+2  
}  
  
//epilogue  
a(n-2) = t2;    //S n-2  
t2 = b + t1;    //A n-1  
a(n-1) = t2;    //S n-1
```

- **Function calls**
  - except in presence of good interprocedural analysis and inlining
- **Conditionals**
  - especially control transfer out of the loop
- **Pointer/array aliasing**

```
for (i=0;i<m;i++){  
    a[i] += c[i] * a[n];  
}
```

- Compiler may not know that `a[i]` and `a[n]` don't overlap
- Can be unrolled, but with limited benefits, as instructions from different iterations can't be interleaved.
- Rewrite:

```
tmp = a[n];  
for (i=0;i<m;i++){  
    a[i] += c[i] * tmp;  
}
```

- If we have a loop nest, then it is possible to unroll one of the outer loops instead of the innermost one.
- Compiler may do this

```
do i=1,n
  do j=1,m
    a(j,i)=c*d(j)
  end do
end do
```



```
do i=1,n,4
  do j=1,m
    a(j,i)=c*d(j)
    a(j,i+1)=c*d(j)
    a(j,i+2)=c*d(j)
    a(j,i+3)=c*d(j)
  end do
end do
```

2 loads for 1 flop

5 loads for 4 flops

- Remember compiler is always there.
- Try to help compiler, rather than do its job for it.
- Use flags and hints as much as possible
- Minimise code modifications

- XL Fortran for AIX: User's Guide
- XL Fortran for AIX: Language Reference
- C for AIX: C/C++ Language Reference
- C for AIX: Compiler Reference

All available locally from HPCx User Guide  
- "References and Further Reading" section

or from [publib.boulder.ibm.com](http://publib.boulder.ibm.com)

- click on "IBM Publications Center" and search