Threaded Programming

Lecture 9: Introduction to performance optimisation

Why?

- Large computer simulations are becoming common in many scientific disciplines.
- These often take a significant amount of time to run.
  - Sometimes they take *too long*.
- There are three things that can be done
  - Change the science (compromise the research)
  - Change the computer (spend more money)
  - Change the program (this is performance optimisation)

What?

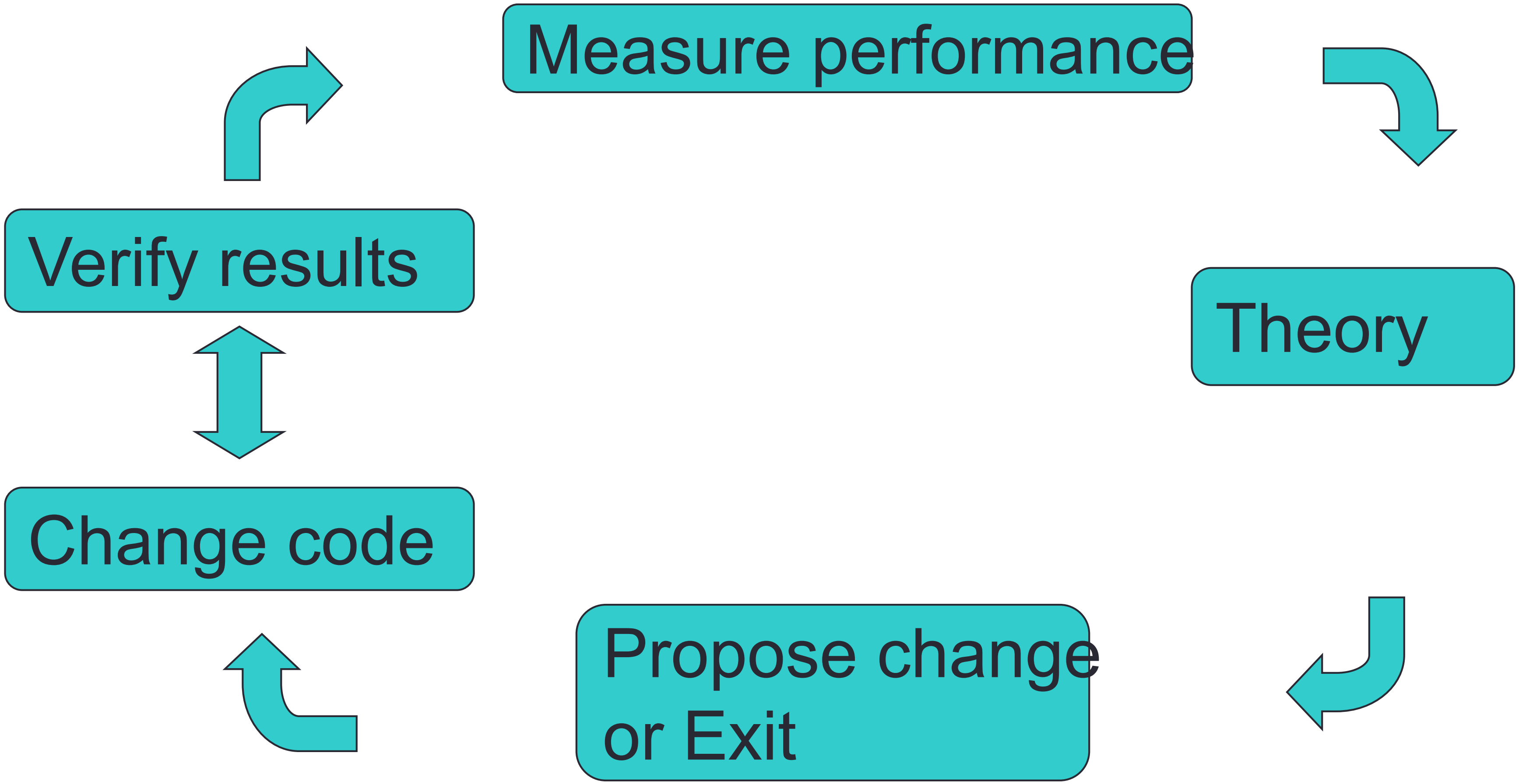
- There are usually many different ways you can write a program and still obtain the correct results.
- Some run faster than others.
  - Interactions with the computer hardware.
  - Interactions with other software.
- **Performance optimisation is the process of making an existing working computer program run faster in a particular Hardware and Software environment.**
  - Converting a sequential program to run in parallel is an example of optimisation under this definition!

When?

- Performance optimisation can take large amounts of development time.
- Some optimisations improve program speed at the cost of making the program harder to understand (increasing the cost of future changes)
- Some optimisations improve program speed at the cost of making the program more specialised and less general purpose.
- It is always important to evaluate the relative costs and benefits when optimising a program
  - This requires the ability to estimate potential gains in advance

How?

- Performance optimisation usually follows a cycle:



Measuring performance

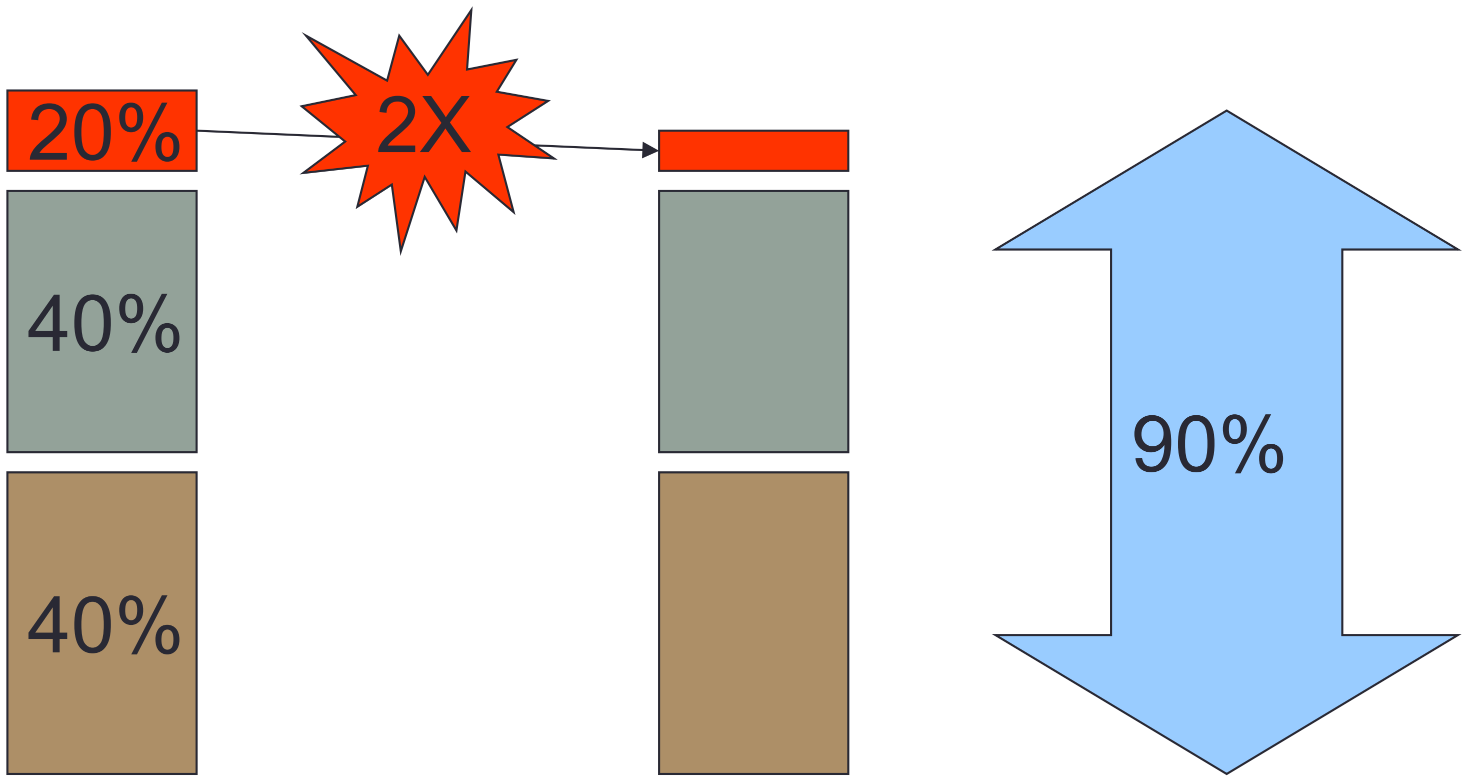
- It is not enough to just measure overall speed
  - You need to know where the time is going.
- There are tools to help you do this
  - They are called profiling tools.
- They give information about:
  - Which sections of code are taking the time
    - Sometimes line by line but usually only subroutines.
  - Sometimes the type of operation
    - memory access
    - floating point calculations
    - file access
- Make sure you understand how variable your results are
  - Are the results down to my changes or just random variation?

Input dependence

- Many codes perform differently with different input data.
- Use multiple sets of input data when measuring performance.
- Make sure these are representative of the problems where you want the code to run quickly.

Only optimise important sections

- Its only worth working on parts of the code that take a lot of time.
- Large speed-up of unimportant sections have little impact on the overall picture.
  - Amdahl's law is this concept applied to parallel processing.
  - Same insight applies to other forms of optimisation.


20%
2X
40%
40%
90%

What is profiling?

- Analysing your code to find out the proportion of execution time spent in different routines.

- Essential to know this if we are going to target optimisation.

- No point optimising routines that don't significantly contribute to the overall execution time.
  - can just make your code less readable/maintainable

Code profiling

- Code profiling is the first step for anyone interested in performance optimisation
- Profiling works by instrumenting code at compile time
  - Thus it's (usually) controlled by compiler flags
  - Can reduce performance
- Standard profiles return data on:
  - Number of function calls
  - Amount of time spent in sections of code
- Also tools that will return hardware specific data
  - Cache misses, TLB misses, cache re-use, flop rate, etc…
  - Useful for in-depth performance optimisation

Sampling and tracing

- Many profilers work by sampling the program counter at regular intervals (normally 100 times per second).
  - low overhead, little effect on execution time
- Builds a statistical picture of which routines the code is spending time in.
  - if the run time is too small (< ~10 seconds) there aren't enough samples for good statistics
- Tracing can get more detailed information by recording some data (e.g. time stamp) at entry/exit to functions
  - higher overhead, more effect on runtime
  - unrestrained use can result in huge output files

Standard Unix profilers

- Standard Unix profilers are prof and gprof
- Many other profiling tools use same formats
- Usual compiler flags are **`-p`** and **`-pg`**:
  - **`ftn -p mycode.F90 -o myprog`** for prof
  - **`cc -pg mycode.c -o myprog`** for gprof
- When code is run it produces instrumentation log
  - **`mon.out`** for prof
  - **`gmon.out`** for gprof
- Then run prof/gprof *on your executable program*
  - eg. **`gprof myprog`** (*not* **`gprof gmon.out`**)

Standard profilers

- **prof myprog** reads **mon.out** and produces this:

```
 %Time  Seconds  Cumsecs   #Calls    msec/call   Name
  32.4     0.71     0.71       14      50.7      relax_
  28.3     0.62     1.33       14      44.3      resid_
  11.4     0.25     1.58        3      83.       __f90_close
   5.9     0.13     1.71  1629419       0.0001   _mcount
   5.0     0.11     1.82   339044       0.0003   __f90_slr_i4
   5.0     0.11     1.93   167045       0.0007
  __inrange_single
   2.7     0.06     1.99      507       0.12     _read
   2.7     0.06     2.05        1      60.       MAIN_
```

epcc

THE UNIVERSITY OF EDINBURGH

Standard profilers

- `gprof myprog` reads `gmon.out` and produces something very similar
- `gprof` also produces a program calltree sorted by inclusive times
- Both profilers list all routines, including obscure system ones
  - Of note: `mcount`(), `_mcount`(), `moncontrol`(), `_moncontrol`() `monitor`() and `_monitor`() are all overheads of the profiling implementation itself
  - `_mcount`() is called every time your code calls a function; if it's high in the profile, it can indicate high function-call overhead
  - `gprof` assumes calls to a routine from different parents take the same amount of time – may not be true

The Golden Rules of profiling

- **Profile your code**
  - The compiler/runtime will NOT do all the optimisation for you.
- **Profile your code yourself**
  - Don't believe what anyone tells you. They're wrong.
- **Profile on the hardware you want to run on**
  - Don't profile on your laptop if you plan to run on ARCHER.
- **Profile your code running the full-sized problem**
  - The profile will almost certainly be qualitatively different for a test case.
- **Keep profiling your code as you optimise**
  - Concentrate your efforts on the thing that slows your code down.
  - This will change as you optimise.
  - So keep on profiling.

Theory

- Optimisation is an experimental process.
- You propose reasons why a code section is slow.
- Make corresponding changes.
- The results may surprise you
  - Need to revise the theory
- Never “optimise” without measuring the impact.

Exit ?

- It is important to know when to stop.
- Each time you propose a code change consider:
  - The likely improvement
    - Code profile and Amdahl`s law helps here.
    - Take account of how long much use you expect for the optimised code. Single use programs are rarely worth optimising.
  - The likely cost
    - Programming/debugging time.
    - Delay to starting simulation
    - “Damage” to the program

Changing code

- Many proposed changes will turn out not be useful.
- You may have to undo your changes.
  - At the very least keep old versions
  - Better to use revision control software.
- Always check the results are still correct !!
  - No point measuring performance if the results are wrong
  - A good test framework will help a lot

Damaging code

- Performance changes can damage other desirable aspects of the code.
  - Loss of encapsulation.
  - Loss of clarity
  - Loss of flexibility
- Think about down-side of changes.
- Look for alternative changes with same performance benefit but less damage.

Experimental frameworks

- Like any experiment, you need to keep good records.
- You will be generating large numbers of different versions of the code.
  - You need to know exactly what the different version were.
  - How you compiled them.
  - Did they get the correct answer.
  - How did they perform.
- You may need to be able to re-run or reproduce your experiments
  - You discover a bug
  - A new compiler is released
  - A new hardware environment becomes available.
  - Etc.

Making things easier

- Keep everything under version control (including results)
- Script your tests so they are easy to run and give a clear yes/no answer.
- Write timing data into separate log-files in easily machine readable format.
- Keep notes.

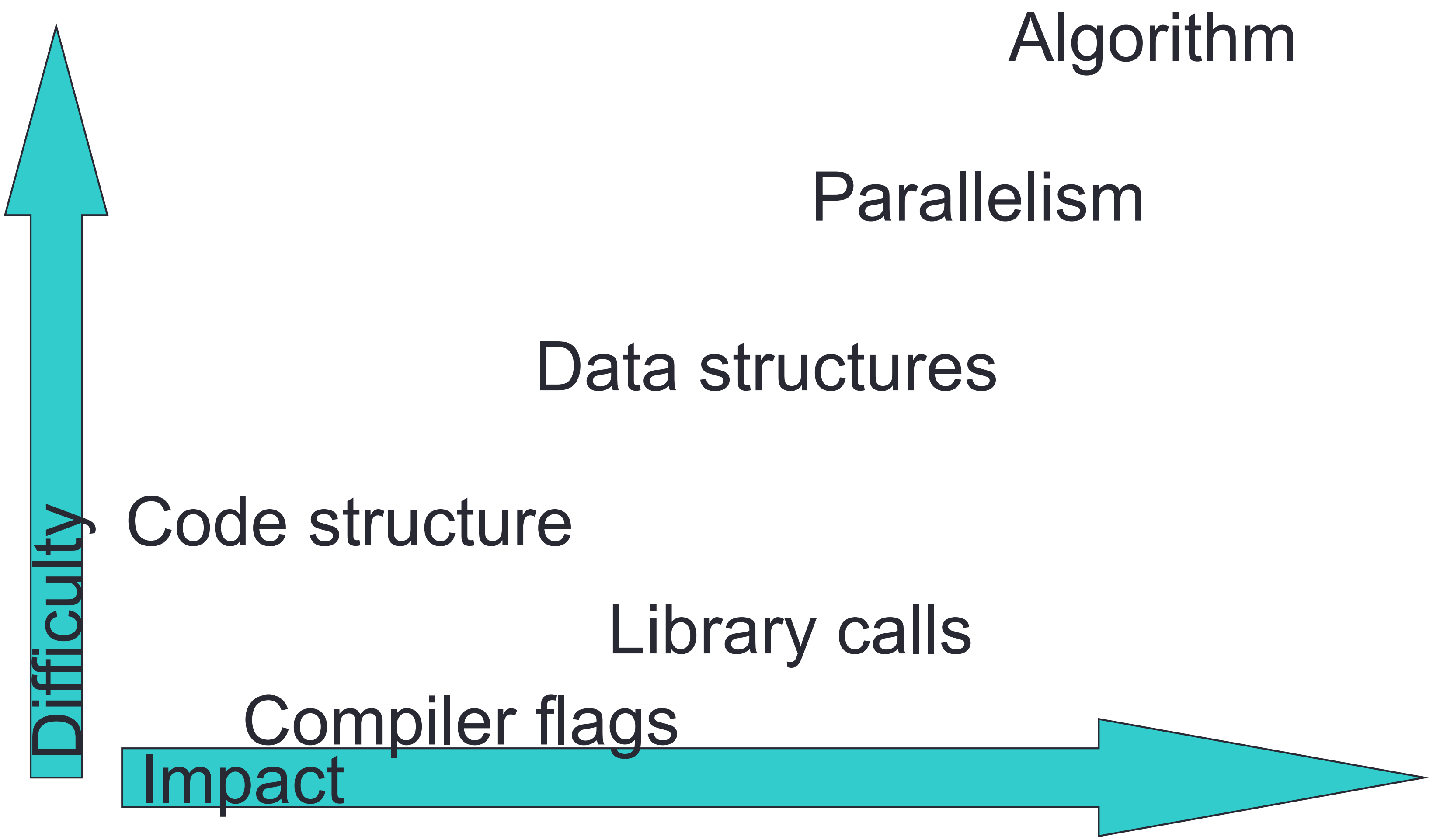
Architecture trends

- Optimisation is the process of tuning a code to run faster in a particular Hardware and Software environment.
- The hardware environment consists of many different resources
  - FPU
  - Cache
  - Memory
  - I/O
- Any of these resources could be the limiting factor for code performance
  - Which one depends on the application

CPU resources

- In the early days of computing memory accesses were essentially free.
  - Optimisation consisted of reducing the instruction count.
- This is no longer the case, and is getting worse
  - CPU performance increases at approx. 80% per year (though recently this has been due to increasing core count rather than clock speed)
  - memory speed increases at approx. 7% per year
- Most HPC codes/systems are memory bandwidth limited.

Types of optimisation



Compiler flags

- Easiest thing to change are the compiler flags
- Most compilers will have good optimisation by default.
  - Some compiler optimisations are not always beneficial and need to be requested explicitly.
  - Some need additional guidance from user (e.g. inter-file in-lining)
  - Some break language standards and need to be requested explicitly
    - E.g. a/2 -> a*0.5 is contrary to Fortran standard but is usually safe.
    - Usually worthwhile to read compiler manual pages before optimising.

Library calls

- The easiest way to make a big impact on code performance is to re-use existing optimised code.
- Libraries represent large amount of development effort
  - Somebody else has put in the effort so you don’t have to,
- Code libraries exist for commonly used functionality (e.g. linear algebra, FFTs etc.).
  - Often possible to access highly optimised versions of these libraries.
  - Even if the application code does not use a standard library it is often easy to re-write to use the standard calls.

Algorithm

- The biggest performance increases typically require a change to the underlying algorithm.
  - Consider changing an O(N) sort algorithm to a O(log(N)) algorithm.
  - This is a lot of work as the relevant code section usually needs a complete re-write.
- A warning
  - The complexity of an algorithm O(N), O(log(N)), O(N log(N)) etc. is related to number of operations and is not always a reliable indication of performance.
    - Pre-factor may make a “worse” algorithm perform better for the value of N of interest.
    - The “worse” algorithms may have much better cache re-use

Code structure

- Most optimisations involve changes to code structure
  - Loop unrolling
  - Loop fusion
  - routine in-lining.
- Often overlap with optimisations attempted by the compiler.
  - Often better to help the compiler to do this than perform change by hand.
- Easier to implement than data changes as more localised.
  - Performance impact is often also smaller unless the code fragment is a major time consumer.
- Performance improvement often at the expense of code maintainability.
  - Try to keep the unoptimised version up to date as well.

Helping the compiler

- 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”.

Compiler flags

- 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...

Compiler flags (cont.)

- Note that highest levels of optimisation may
  - break your code.
  - give different answers, by bending standards.
  - make your code go slower.

- Always read documentation carefully.

- Isolate routines and flags which cause the problem.
  - binary chop
  - one routine per file may help

Code modification

- 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 and compiler versions.

- Try to think about
  - what optimisation the compiler is failing to do
  - What additional information can be provided to compiler
  - how can rewriting help

Conditionals

- Even with sophisticated branch prediction hardware, branches are bad for performance.
- Try to avoid branches in innermost loops.
  - if you can't eliminate them, at least try to get them out of the critical loops.

```
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
```

Data types

- Performance can be affected by choice of data types
  - often a difference between 32-bit and 64-bit arithmetic (integer and floating point).
  - complicated by trade-offs with memory usage and cache hit rates
- Avoid unnecessary type conversions
  - e.g. int to long, float to double
  - N.B. some type conversions are implicit
  - However sometimes better than the alternative e.g.
    - Use DP reduction variable rather than increase array precision.

epcc

THE UNIVERSITY OF EDINBURGH

CSE

- 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)
```

Register use

- Most compilers make a reasonable job of register allocation.
  - But only limited number available.
- 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, most compilers will use a register for an explicit scalar in preference to an implicit CSE.

```
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;
}
```

Spilling

- 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

- 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

- Compiler has to determine independence of iterations

Loop unrolling

- Loops with small bodies generate small basic blocks of assembly code
  - lot of dependencies between instructions
  - high branch frequency
  - little scope for good instruction scheduling

- Loop unrolling is a technique for increasing the size of the loop body
  - gives more scope for better schedules
  - reduces branch frequency
  - make more independent instructions available for multiple issue.

Loop unrolling

- Replace loop body by multiple copies of the body
- Modify loop control
  - take care of arbitrary loop bounds
- Number of copies is called unroll factor

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
```

- Remember that this is in fact done by the compiler at the IR or assembly code level.
- If the loop iterations are independent, then we end up with a larger basic block with relatively few dependencies, and more scope for scheduling.
  - also reduce no. of compare and branch instructions
- Choice of unroll factor is important (usually 2,4,8)
  - if factor is too large, can run out of registers
- Cannot unroll loops with complex flow control
  - hard to generate code to jump out of the unrolled version at the right place

Impediments to unrolling

- Function calls
  - except in presence of good interprocedural analysis and inlining

- Conditionals
  - especially control transfer out of the loop
  - Lose most of the benefit anyway as they break up the basic block.

- Pointer/array aliasing
  - Compiler can’t be sure different values don’t overlap in memory

Example

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

- Compiler doesn't know that `a[indx[i]]` and `a[ip]` don't overlap
- Could try hints
  - tell compiler that `indx` is a permutation
  - tell compiler that it is OK to unroll
- Or could rewrite:

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

Inlining

- Compilers very variable in their abilities

- Hand inlining possible
  - very ugly (slightly less so if done via pre-processor macros)
  - causes code replication

- Compiler has to know where the source of candidate routines is.
  - sometimes done by compiler flags
  - easier for routines in the same file
  - try compiling multiple files at the same time

- Very important for OO code
  - OO design encourages methods with very small bodies
  - inline keyword in C++ can be used as a hint

Vector Instructions (Vectorisation)

- Modern CPUs can perform multiple operations each cycle
  - Use special SIMD (Single Instruction Multiple Data) instructions
    - e.g. SSE, AVX
  - Operate on a "vector" of data
    - typically 2 or 4 double precision
    - potentially gives speedup in floating point operations
  - Usually only one loop is vectorisable in loop nest
    - And most compilers only consider inner loop

- Optimising compilers will use vector instructions
  - Relies on code being vectorisable
  - ...or in a form that the compiler can convert to be vectorisable
  - Some compilers are better at this than others
  - But there are some general guidelines about what is likely to work...

Requirements for vectorisation

- Loops must have determinable (at run time) trip count
  - rules out most while loops
- Loops must not contain function/subroutine calls
  - unless the call can be inlined by the compiler
  - maths library functions usually OK
- Loops must not contain braches or jumps
  - guarded assignments may be OK
  - e.g. `if (a[i] != 0.0) b[i] = c * a[i];`
- Loop trip counts needs to be long, or else a multiple of the vector length

- Loops must no have dependencies between iterations
  - reductions usually OK, e.g. `sum += a[i];`
  - avoid induction variables e.g. `indx += 3;`
  - use `restrict`
  - may need to tell the compiler if it can't work it out for itself
- Aligned data is best
  - e.g. AVX vector loads/stores operate most effectively on 32-bytes aligned address
  - need to either let the compiler align the data....
  - ..or tell it what the alignment is
- Unit stride through memory is best

Multiple Optimisation steps

- Sometimes multiple optimisation steps are required.
  - Multiple levels of in-lining.
  - In-lining followed by loop un-rolling followed by CSE.
- The compiler may not be able to perform all steps at the same time
  - You may be able to help the compiler by performing some of the steps by hand.
  - Look for the least damaging code change that allows the compiler to complete the rest of the necessary changes.
  - Ideally try each step in isolation before attempting to combine hand-optimisations.

Data structures

- Changing the programs data structures can often give good performance improvements
  - These are often global changes to the program and therefore expensive.
    - Code re-writing tools can help with this.
    - Easier if data structures are reasonably opaque, declared once
      - objects, structs, F90 types, included common blocks.
  - As memory access is often the major performance bottleneck the benefits can be great.
    - Improve cache/register utilisation.
    - Avoid pointer chasing
  - May be able to avoid memory access problems by changing code structure in key areas instead.

Programmer’s perspective:

- Memory structures are the programmers responsibility
  - At best the compiler can add small amounts of padding in limited circumstances.
  - Compilers can (and hopefully will) try to make best use of the memory structures that you specify (e.g. uni-modular transformations)
- Changing the memory structures you specify may allow the compiler to generate better code.

Arrays

- Arrays are large blocks of memory indexed by integer index
- Probably the most common data structure used in HPC codes
- Good for representing regularly discretised versions of dense continuous data

$$f(x,y,z) \rightarrow F[i][j][k]$$

Arrays

- Many codes loop over array elements
  - Data access pattern is regular and easy to predict
- Unless loop nest order and array index order match the access pattern may not be optimal for cache re-use.
  - Compiler can often address these problems by transforming the loops.
  - But sometimes can do a better job when provided with a more cache-friendly index order.

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.
- Things to look for:
  - Unnecessary pointer chasing
    - pointer arrays that could be simple arrays
    - linked lists that could be arrays.
  - Unnecessary temporary arrays.
  - Tables of values that would be cheap to re-calculate.

Utilizing caches

- Want to use all of the data in a cache line
  - loading unwanted values is a waste of memory bandwidth.
  - structures are good for this
  - or loop fastest over the corresponding index of an array.
- Place variables that are used together close together
  - Also have to worry about alignment with cache block boundaries.
- Avoid "gaps" in structures
  - In C structures may contain gaps to ensure the address of each variable is aligned with its size.

Arrays and caches

Bad: non-contiguous memory accesses

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

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

Good: contiguous memory accesses

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

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

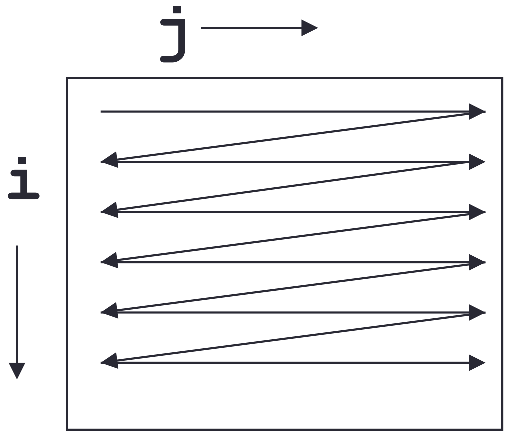
Cache blocking

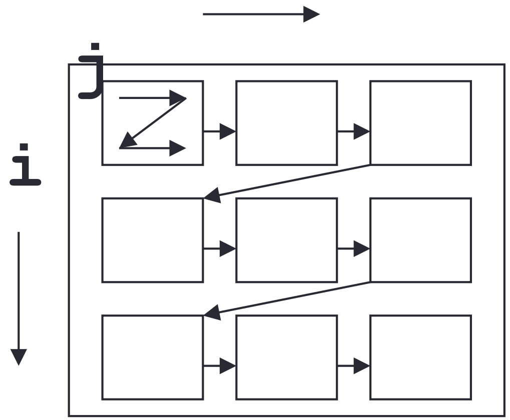
- A combination of:
  - strip mining (also called loop blocking, loop tiling...)
  - loop interchange
- Designed to increase data reuse:
  - temporal reuse: reuse array elements already referenced
  - spatial reuse: good use of cache lines
- Many ways to block any given loop nest
  - Which loops should be blocked?
  - What block size(s) will work best?

Blocking example

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

```
for (ii=0;ii<n;ii+=B){
   for (jj=0;jj<n;jj+=B){
      for (i=ii;i<ii+B;i++){
         for (j=jj;j<jj+B;j++){
            a[i][j]+=b[i][j];
         }
      }
   }
}
```





Pointer aliasing

- Pointers are variables containing memory addresses.
  - 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, so to be safe compiler will:
  - 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
- 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!
- Use the C99 `restrict` keyword where possible
- ...or else use compiler flags
- Explicit use of scalar temporaries may also reduce the problem

Key points to remember

- Optimisation tunes a code for a particular environment
  - Not all optimisations are portable.
- Optimisation is an experimental process.
- Need to think about cost/benefit of any change.
- Always verify the results are correct.