ARCHER Single Node Optimisation

Optimising with the compiler

Slides contributed by Cray and EPCC

Overview

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

Introduction

- 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

Compiler flags (cont.)

- Many compilers are designed for an instruction set architecture, not one machine.
  - flags to target ISA versions, processor versions, cache configurations
  - defaults may not be optimal, especially if cross-compiling

- Some optimisation flags may not be part of `-O`*n*
  - check documentation
  - use sparingly (may only be beneficial in some cases)

Compiler hints

- 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

Incremental compilation

- Compilers can only work with the limited information available to them.
- Most compilers compile code in an incremental fashion
  - Each source file is compiled independently of each other.
    - Most compilers ignore all source files other than those specified on the command line (or implicitly referenced via search paths, e.g. include files)
  - Routines from other source files treated as “black-boxes”
    - Make worst case assumptions based on routine prototype.
- You can help by providing more information
  - Information in routine prototypes
    - INTENT, PURE, const, etc.
  - Compiler hints
  - Command line flags

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

- 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

Locals and globals

- 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

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
```

- 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
```

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.

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)
```

CSE including function calls.

- To extract a CSE containing a function call the compiler has to be sure of various things:
  - The function always returns the same value for the same input.
  - The function does not cause any side effects that would be effected by changing the number of times the function is called:
    - Modifying its inputs.
    - Changing global data.
- Need to be very careful with function prototypes to allow compiler to know this.

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

Outer loop unrolling

- If we have a loop nest, then it is possible to unroll one of the outer loops instead of the innermost one.
- Can improve locality.

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

→

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

2 loads for 1 flop

5 loads for 4 flops

Variable expansion

- Variable expansion can help break dependencies in unrolled loops
  - improves scheduling opportunities
- Close connection to reduction variables in parallel loops

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

unroll

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

expand `b`

```
for (i=0,i<n,i+=2){
    b1+=a[i];
    b2+=a[i+1];
}
b=b1+b2;
```

Register renaming

- Registers may be reused within a basic block introducing unnecessary dependencies.
- Using two (or more) different registers can preserve program correctness, but allow more scheduling flexibility
  - Some CPUs perform register rename and reschedule in hardware, this can utilise additional registers not visible to compiler.

```
add %f2,1,%f1
st [%o1],f1
add %f3,2,%f1
st [%o2],f1
```

rename →

```
add %f2,1,%f1
st [%o1],f1
add %f3,2,%f27
st [%o2],f27
```

reschedule →

```
add %f2,1,%f1
add %f3,2,%f27
st [%o1],f1
st [%o2],f27
```

Software pipelining

- Problem with scheduling small loop bodies is that there are dependencies between instructions in the basic block.

- Potentially possible to start executing instructions from the next iteration before current one is finished.

- Idea of software pipelining is to construct a basic block that contains instructions from different loop iterations.
  - fewer dependencies between instructions in block
  - needs additional code at start and end of loop

Software pipelining

```
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
```

archer

epcc

At instruction level

```
L:  ld      [%r1],%f0
    fadd    f0,f1,f2
    st      [%r1],f2
    add     %r1,4,%r1
    cmp     %r1,%r3
    bg      L
    nop
```

```
    ld      [%r1],%f0
    fadd    f0,f1,f2
    ld      [%r1+4],%f0

L:  st      [%r1],f2
    fadd    f0,f1,f2
    ld      [%r1+8],%f0
    cmp     %r1,%r3-8
    bg      L
    add     %r1,4,%r1

    st      [%r1],f2
    add     %r1,4,%r1
    fadd    f0,f1,f2
    st      [%r1],f2
```

`st` must wait for `fadd` to complete: pipeline stall for data hazard

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

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.

General Cray Compiler Flags

- Optimisation Options
  - **-O2** — optimal flags [enabled by default]
  - **-O3** — aggressive optimization
  - **-O ipaN (ftn)** or **-hipaN (cc/CC)** — inlining, N=0-5 [default N=3]
- Create listing files with optimization info
  - **-ra (ftn)** or **-hlist=a (cc/CC)** — creates a listing file with all optimization info
  - **-rm (ftn)** or **-hlist=m (cc/CC)** — produces a source listing with loopmark information
- Parallelization Options
  - **-O omp (ftn)** or **-h omp (cc/CC)** — Recognize OpenMP directives [default]
  - **-O threadN (ftn)** or **-h threadN (cc/CC)** — control the compilation and optimization of OpenMP directives, N=0-3 [default N=2]

➔ More info: man crayftn, man craycc, man crayCC

Recommended CCE Compilation Options

- Use default optimization levels
  - It's the equivalent of most other compilers -O3 or -fast
  - It is also our most thoroughly tested configuration
- Use -O3,fp3 (or -O3 -hfp3, or some variation) if the application runs cleanly with these options
  - **-O3** only gives you slightly more than the default **-O2**
  - Cray also test this thoroughly
  - **-hfp3** gives you a lot more floating point optimization (default is **-hfp2**)
- If an application is intolerant of floating point reordering, try a lower -hfp number
  - Try **-hfp1** first, only **-hfp0** if absolutely necessary (**-hfp4** is the maximum)
  - Might be needed for tests that require strict IEEE conformance
  - Or applications that have 'validated' results from a different compiler
- Do not use too aggressive optimizations , e.g. -hfp4
  - Higher numbers are not always correlated with better performance

OpenMP

- OpenMP is ON by default
  - This is the opposite default behavior that you get from GNU and Intel compilers
  - Optimizations controlled by **-OthreadN (ftn)** or **-hthreadN (cc/CC)**, N=0-3 [default N=2]
  - To shut off use **-O/-h thread0** or **-xomp (ftn)** or **-hnoomp**

- Autothreading is NOT on by default
  - **-hautothread** to turn on
  - Interacts with OpenMP directives

- If you do not want to use OpenMP and have OMP directives in the code, make sure to shut off OpenMP at compile time

CCE – GNU – Intel compilers

- More or less all optimizations and features provided by CCE are available in Intel and GNU compilers

  - GNU compiler serves a wide range of users & needs
    - Default compiler with Linux, some people only test with GNU
    - **GNU defaults are conservative** (e.g. -O1)
      - -O3 includes vectorization and most inlining
    - Performance users set additional options
  - Intel compiler is typically more aggressive in the optimizations
    - **Intel defaults are more aggressive** (e.g -O2), to give better performance “out-of-the-box”
      - Includes vectorization; some loop transformations such as unrolling; inlining within source file
    - Options to scale back optimizations for better floating-point reproducibility, easier debugging, etc.
    - Additional options for optimizations less sure to benefit all applications
  - **CCE is even more aggressive** in the optimizations by default
    - Better inlining and vectorization
    - Aggressive floating-point optimizations
    - OpenMP enabled by default

- GNU users probably have to specify higher optimisation levels

Cray, Intel and GNU compiler flags

Feature	Cray	Intel	GNU
Listing	-hlist=a	-opt-report3	-fdump-tree-all
Free format (ftn)	-f free	-free	-ffree-form
Vectorization	By default at -O1 and above	By default at -O2 and above	By default at -O3 or using -ftree-vectorize
Inter-Procedural Optimization	-hwp	-ipo	-flto (note: link-time optimization)
Floating-point optimizations	-hfpN, N=0...4	-fp-model [fast|fast=2|precise|except|strict]	-f[no-]fast-math or -funsafe-math-optimizations
Suggested Optimization	(default)	-O2 -xAVX	-O2 -mavx -ftree-vectorize -ffast-math -funroll-loops
Aggressive Optimization	-O3 -hfp3	-fast	-Ofast -mavx -funroll-loops
OpenMP recognition	(default)	-fopenmp	-fopenmp
Variables size (ftn)	-s real64 -s integer64	-real-size 64 -integer-size 64	-freal-4-real-8 -finteger-4-integer-8

Summary

- 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