Compiling for Performance on hp OpenVMS I64

Doug Gordon

Original Presentation by Bill Noyce

European Technical Update Days, 2005

Compilers discussed

- C, Fortran, [COBOL, Pascal, BASIC]
  - Share GEM optimizer & code generator
  - Much in common with Alpha compilers
- C++
  - Different optimizer & code generator
  - Well-tuned for Itanium

Performance Topics

- Alignment
- Memory
- Floating Point
- Optimization Levels

Alignment

- Use natural alignment whenever possible
- Unaligned data is handled in software
  - When compiler knows, ugly code adds a few cycles
  - When unexpected, takes an expensive trap
  - These traps are 5x more expensive than on Alpha

Alignment

- Use:
  - Fortran /align=(…) /warn=alignment
  - C & C++ #pragma member_alignment
  - C & C++ __unaligned attribute where needed
  - COBOL /align & *DC SET ALIGNMENT

- Avoid:
  - C & C++   #pragma nomember_alignment
  - C & C++   #pragma pack
  - Fortran SEQUENCE attribute

Memory

- CPU speed advancing faster than memory
- Big caches can help
- Design algorithms for cache locality
- Allow compiler to schedule loads early
- Avoid apparent aliasing

Memory - Aliasing

- SAXMAIN.FOR

```
real a(1000)
real b(1000,1000)
real c(1000,1000)
do j=1,1000
    do i=1,1000
        call saxpy(1000,
              a(j), b(1,i), c(1,j))
    enddo
enddo
end
```

- SAXC.C

```
void saxpy( int *np,
       float *ap,
       float *x, float *y)
{
    int i;
    for (i=0; i<*np; i++)
        y[i] = y[i] + *ap * x[i];
}
```

Memory - Aliasing

- Store to y[i] might affect *ap or x[i+1]
- Compiled code completes one iteration before starting the next
- 2 billion FLOPs in 10 secs = 200 MFLOPS
- Idiomatic C makes no difference:

```
for (i=0; i<*np; i++)  *y++ += *ap * *x++;
```

- /noansi_alias is even worse (alias *np):

  2 billion FLOPS in 12 secs = 170 MFLOPS

Memory - Aliasing

- Eliminate aliasing with *ap:

```
float t = *ap;
for (i=0; i<*np; i++)
      y[i] += t * x[i];
```

- 2 billion FLOPs in 3 secs = 670 MFLOPS
- Compiler produced two versions of loop, with test for alias between x & y
- Guarded loop gets unrolled and scheduled
- Guarded loop not eligible for software pipelining

Memory - Aliasing

- Rewrite in Fortran to remove all aliasing

```
subroutine saxpy(n,a,x,y)
integer n, i
real a, x(n), y(n)
do i=1,n
        y(i) = y(i) + a*x(i)
enddo
end
```

- 2 billion FLOPs in 2 secs = 1000 MFLOPS
- Loop is pipelined with no checks needed

Memory - Aliasing

- Use Itanium features (speculative load)
  - add extern “C” & compile with C++
- 2 billion FLOPS in 8 secs = 250 MFLOPS

- Eliminate alias with *np:

  int n = *np; for (i=0; i<n; …)
- Loop is pipelined, and checks inserted
- 2 billion FLOPs in 2.2 secs = 900 MFLOPS

- Add /assume=noaccuracy_sensitive
- 2 billion FLOPS in 1.9 secs = 1100 MFLOPS

Floating Point

- Use native IEEE floating-point formats
- Same precision & essentially same range as VAX F & G formats
- VAX formats (F, D, G) are emulated in software by converting to/from IEEE
  - Performance cost up to 5x
- IEEE formats also work well on Alpha

Floating Point

- If files must use VAX formats, convert on input & output
- In Fortran, CONVERT= makes it easy
- Otherwise, CVT$ routines can be used

Floating Point

- IEEE formats can support new semantics:
  - Gradual underflow (denorms)
  - Infinity and NaN instead of traps
- Selected by main program’s compilation:
  - /IEEE_mode = FAST
  - /IEEE_mode = UNDERFLOW_TO_ZERO
  - /IEEE_mode = DENORM_RESULT
- Producing or using a denorm can be slow
  - Traps to “software assistance” handler
  - Can avoid by choosing flush-to-zero semantics

Floating Point

- One-at-a-time math

  x = a*b + c*d

  1. multiply a*b (& round)
  2. multiply c*d (& round)
  3. add the products

- Fused mul-add

  x = a*b + c*d

  1. multiply a*b (& round)
  2. multiply c*d & add
     (round only at end)

- These produce slightly different results
- Fused version is often more accurate, but less predictable
- Fused version runs faster

Floating Point

- /assume=noaccuracy_sensitive enables transformations that can change results
  - – Fused mul-add
  - – Replace divide with multiply by inverse
  - – Tree height reduction
- Some apps are “sensitive” to any change
  - – Therefore, these are disabled by default
- Poor abbreviation: assume=noaccuracy
  - – Doesn’t mean what this sounds like

Optimization Levels

- OpenVMS compilers default to high optimization
- You may reduce opt level for debugging
- /opt=level= (for GEM compilers)
  - 0: very naïve code, no optimization at all (= /noopt)
  - 1: simple peephole optimizations
  - 2: traditional opts: CSE, hoist, strength
  - 3: adds loop unrolling
  - 4: adds inlining & software pipelining (default)
  - 5: adds loop interchange & blocking, may help or hurt

Optimization Levels

- Default (high) level is designed to be safe for standard-conforming programs
- Additional transformations via switches:
  - /assume=noaccuracy_sensitive
  - /assume=nopointers_to_globals
  - /assume=nomath_errno
- More /assume= switches available for programs that break the language standard’s rules
- “Optimizer bugs” are usually user errors
- If it is our bug, we want to fix it

Performance Topics

- Alignment
- Memory
- Floating Point
- Optimization Levels

Questions?