Performance Programming I
Exploiting the Power Processor

Larry Carter
Sean Peisert
Outline

• Exploiting the Power Processor (Monday)
  – Peak processor performance:
    • Is it attainable?
    • What can go wrong?
    • Tricks and pitfalls
  – Skills
    • Reading assembly code
    • Timing & profiling

• Lab

• Cache and TLB issues (Tuesday)
Approach

• Engineer’s method:
  - DO UNTIL (exhausted)
  - tweak something
  - IF (better) THEN accept change

• Scientific method:
  - DO UNTIL (enlightened)
  - make hypothesis
  - experiment
  - revise hypothesis
Power3’s power … and limits

- Eight pipelined functional units
  - 2 floating point
  - 2 load/store
  - 2 single-cycle integer
  - 1 multi-cycle integer
  - 1 branch

- Powerful operations
  - Fused multiply-add (FMA)
  - Load (or Store) update
  - Branch on count

- Launch 4 ops per cycle
- Can’t launch 2 stores/cyc
- FMA pipe 3-4 cycles long
- Memory hierarchy (Tues)
Can its power be harnessed?

```c
for (j=0; j<n; j+=4) {
    p00 += a[j+0]*a[j+2];
    m00 -= a[j+0]*a[j+2];
    p01 += a[j+1]*a[j+3];
    m01 -= a[j+1]*a[j+3];
    p10 += a[j+0]*a[j+3];
    m10 -= a[j+0]*a[j+3];
    p11 += a[j+1]*a[j+2];
    m11 -= a[j+1]*a[j+2];
}
```

8 FMA’s
4 Loads

Runs at 4.6 cycles/iteration (= 772 MFLOP/S)
Can its power be harnessed (part II)

- 8 FMA, 4 Load - 1.15 cycle/load (previous slide)
- 8 FMA, 6 Load - 1.3 cycle/load
- 8 FMA, 8 Load - 1.2 cycle/load
- 4 Add, 4 Load - 1.1 cycle/load
- Shift, Add, Load, Store - 1.15 cycle/MemOp
- Load, Store - 1.1 cycle/MemOp

---

- I haven’t broken the 1 cycle/MemOp barrier!
- but I’ve only spent 2 days trying …maybe the AGEN unit is disabled ...
FLOP to MemOp ratio

• Most programs have at most one FMA per MemOp
  – Matrix-vector product: (K+1) loads, K fma’s
  – FFT butterfly: 8 MemOps, 10 floats (but 5 or 6 FMA)
  – DAXPY: 2 Loads, 1 Store, 1 FMA
  – DDOT: 2 Loads, 1 FMA

• A few have more (use ESSL!)
  – Matrix multiply (well-tuned): 2 FMA per load
  – Radix-8 FFT

• Performance is limited by Memory Operations!
The effect of pipeline latency

for (i=0; i<size; i++) {
    sum = a[i] + sum;
}

Next add can’t start until previous is finished (3 to 4 cycles later)

for (i=0; i<size; i+=4) {
    sum0 += a[i];
    sum1 += a[i+1];
    sum2 += a[i+2];
    sum3 += a[i+3];
}
sum = sum0+sum1+sum2+sum3;

May change answer due to different rounding.
What’s so great about Fortran??

DO I = 1, N
  A(I) = B(I)
ENDDO

CL.8:
L4A      gr0=b(gr5,4)
L4A      gr6=b(gr5,8)
L4A      gr7=b(gr5,12)
L4AU     gr8,gr5=b(gr5,16)
ST4A     a(gr4,8)=gr6
ST4A     a(gr4,4)=gr0
ST4A     a(gr4,12)=gr7
ST4U     gr4,a(gr4,16)=gr8
BCT      ctr=CL.8,

CL.6:
BCT      ctr=CL.8,

for (i=0; i<N; i++) {
  b[I] = a[i];
}

ST4U     gr4,(*int(gr4,4)=gr24
L4AU     gr24,gr3=(*int(gr3,4)
BCT      ctr=CL.6,
Fortran vs C - what’s going on??

• C prevents compiler from unrolling code
  – A feature, not a bug!
  – User may want \texttt{b[0]} and \texttt{a[1]} to be same location
  – tricky way to set \texttt{a[n]} = \ldots = \texttt{a[1]} = \texttt{a[0]}

• Most C compilers don’t try to prove non-aliasing
  – \texttt{a} and \texttt{b} were \texttt{malloc}-ed in this example

• Fortran doesn’t allow arrays to be aliased
  – Unless explicit, e.g. via \texttt{EQUIVALENCE}
Fortran vs. C - does it matter??

• Yes - Fortan code *should* perform better
  – My tests show both are about 1 cycle/MemOp
  – Fortran *should* be .5 cycle/MemOp

• No - you could get the “Fortran” object code from

```c
for (i=0; i<N; i+=4) {
   b0 = a[i];
   b1 = a[i+1];
   b2 = a[i+2];
   b3 = a[i+3];
   b[i] = b0;
   b[i+1] = b1;
   b[i+2] = b2;
   b[i+3] = b3;
}
```
Miscellany

• Excellent reference:

• Use ESSL and PESSL if appropriate

• MASS is much faster for intrinsic functions
  – But may differ in last bit from IEEE standard

• I’m carter@cs.ucsd.edu, www.cs.ucsd.edu/users/carter
Performance Programming II
Cache and TLB Issues

Larry Carter
Sean Peisert
Stride one memory access

![Graph showing cycles per load vs KB of memory accessed for L1 and L2 caches.](image-url)
Strided Memory Access

Program adds 4440 integers located at given stride
Strided Memory Access

Program adds 22200 integers located at given stride
Strided Memory Access

Square - 4,440 element sum, diamond - 22,200 element sum

Stride

Cycles per load

> 1 element/cacheline > 1 element/page 1 element/page

(as bad as it gets)

Stride 64

55 110
Decreasing MemOp to FLOP Ratio

for (i=1; i<N; i++)
    for (j=1; j<N; j++)
        b[i,j] = 0.25 *
            (a[i-1][j] + a[i+1][j]
            + a[i,j-1] + a[i][j-1]);

for (i=1; i<N-2; i+=3) {
    for (j=1; j<N; j++) {
        b[i+0][j] = ... ;
        b[i+1][j] = ... ;
        b[i+2][j] = ... ;
    }
}

for (i = i; i < N; i++) {
    ... ; /* Do last rows */