RESEARCH • TECHNOLOGY • INNOVATION

Wednesday 14th October 2015

Performance Analysis & Optimization.

- Ultimate goal:

  Answer questions: 1) How fast does my code run?
  → Performance measurement.

  2) How fast can my code run?
  → Performance analysis/modeling.

  3) How can we achieve 2)
  → Optimization.

- Sessions.

  Wed. 09:00 - 10:00 This lecture.
  10:30 - 12:00 Excercises.

  Thu. 08:30 - 10:30 Cont. Excercises from Wed.
  Interleaved lecture/excercise.

20 Konstantinou Kavafi Street, 2121 Aglantzia, P.O. Box 27456, 1645 Nicosia, Cyprus
Tel.: +357 22208600 • Fax: +357 22208625
www.cyi.ac.cy

RESEARCH • TECHNOLOGY • INNOVATION

1) How fast does my code run?

- Flop-rate
- Bandwidth
- Wall-time

→ Measure time from beginning to end.

time ./a.out.

- Depending on OS, 'time' gives user/system/cpu.
- No fine-grained info.
- Inconvenient - not portable.

→ Measure time using a system function.

- Need "wall time": not cpu-ticks, not program time.
- gettimeofday()

Returns time elapsed since epoch (Jan 1st, 1970 for most UnixSystems)

Returns two integers: tv_sec (seconds)

tv_usec (μseconds)

⇒ Seconds = $tv\_sec + tv\_usec \cdot 10^{-6}$

- Can wrap around my kernels for fine-grained info.
- Even though it returns μsec, don't assume error is $\mathcal{O}(\mu s)$.
- O.S. "jitters" ⇒ runtime fluctuates.
- Take a sufficient number of measurements
- E.g.

```
for i=1,...,Nouter
    t0 = Stop_watch()
    for j=1,...,Ninner
        my_kernel()
    end for
    t1 = Stop_watch()
    T[i] = (t1-t0)/Ninner;
end for.
```

Take average and error:

$$\bar{T} = \sum_i \frac{T_i}{Nouter} \qquad \sigma_T = \sqrt{\frac{(T_i-\bar{T})^2}{Nouter}}$$

: Increase Ninner until.

$\frac{\sigma_T}{\bar{T}}$ acceptable. (e.g. 5%)

20 Konstantinou Kavafi Street, 2121 Aglantzia, P.O. Box 27456, 1645 Nicosia, Cyprus
Tel.: +357 22208600 • Fax: +357 22208625
www.cyi.ac.cy

RESEARCH • TECHNOLOGY • INNOVATION

- Measure performance
  - Susstained Floating-point operations per second
    —— // —— Bytes-read/written per second.

  - Assumptions for our kernels:
    - Fixed number of FP ops per kernel call
    - —— // —— Bytes of I/O per kernel call

    $\Rightarrow \beta_{FP} = \frac{N_{FP}}{\bar{T}}$: susstained Floating point operations.

    $\beta_{IO} = \frac{N_{IO}}{\bar{T}}$: susstained bandwidth.

    N_{FP}: Count floating point operations explicitely.

    +, -, * : 1 FP operation.

    /, sqrt(), etc : depend on architecture $\gg$ 2 Flops.

    N_{IO}: Count bytes of I/O explicitly.

    - Assume that for every kernel call, each byte read is never re-read

      "maximum data re-use"

    - Count I/O of $\mathcal{O}(N)$ and above, where N is the problem size.

  - Example:

    Kernel: $Y_j^{ab} = \sum_{c=0}^{N-1} A^{ac} X_j^{cb}$: Array of matrices-times-array.

    $j = 1, \ldots, L$, $L \gg N$.

    e.g. $N=2$: $Y_j^{ab} = A^{a0} \cdot X_j^{0b} + A^{a1} \cdot X_j^{1b}$

20 Konstantinou Kavafi Street, 2121 Aglantzia, P.O. Box 27456, 1645 Nicosia, Cyprus
Tel.: +357 22208600 • Fax: +357 22208625
www.cyi.ac.cy

RESEARCH • TECHNOLOGY • INNOVATION

$$Y_i^{ab} = \underbrace{A^{ab} \cdot X_i^{ob} + A^{a1} \cdot X_i^{1b}}_{\text{3 terms}}$$

F·P: $3 \cdot 2 \cdot 2 \cdot L \simeq 12 \cdot L$

I/O: $\underbrace{(2 \cdot 2 + 2 \cdot 2)}_{X \quad Y} \cdot L + \underbrace{22}_{A} \simeq 8\,L$ + low order terms (L).

$= 8 \cdot 8 \cdot L = 64\,L$ Bytes.

$\rightarrow L = 2^{22} = 4M$, $T = 0.1$ sec. $\Rightarrow \beta_{FP} = \frac{2^{22} \cdot 12}{0.1\,\text{sec}} = 120\,2^{22}$ Flop/sec $= 0{,}5$ GFlops/sec.

$\Rightarrow \beta_{IO} = \frac{2^{22} \cdot 64}{0.1\,\text{sec}} = 64 \cdot 2^{22}$ Bytes/sec.

$\simeq 2{,}7$ GByte/sec.

Note: 1) Such kernels have a fixed number of Flops/byte I/O independent of "L"

"Arithmetic or Computational Intensity": e.g. here: $\frac{12L}{64L} \simeq 0.1875 \frac{\text{Flop}}{\text{Byte}}$

$I_k = \beta_{FP}/\beta_{IO} = N_{FP}/N_{IO}$ = independant of problem size.

2) Assumed max. data reuse.

Re-using the elements of X_i did not as extra I/o

$$Y_i^{00} = A^{00} X_i^{00} + A^{01} X_i^{10}$$
$$Y_i^{01} = A^{00} X_i^{01} + A^{01} X_i^{11}$$
$$Y_i^{10} = A^{10} X_i^{00} + A^{11} X_i^{10}$$
$$Y_i^{11} = A^{10} X_i^{01} + A^{11} X_i^{11}$$

Remark: Other ways to count performance: Hardware counters (PAPI)

→ For codes with non-deterministic FP/IO counts.

RESEARCH • TECHNOLOGY • INNOVATION

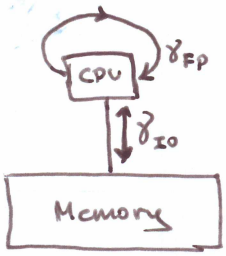
2) How fast can my code run?

Depends on:

i) Machine/Hardware
- Peak: FP, Bw

ii) Kernel.
- Arithmetic intensity.

Assume a simple model for the architecture:



γ_{FP} : Rate at which processor consumes FP ops.

$\gamma_{I/O}$: Rate at which bytes of I/O move between memory–CPU.

Peak F.P.:
- Clock rate of CPU (GHz)
- Flop/cycle of CPU (vector-length).

Vector length:

- Modern CPUs allow processing multiple data at the same time:
  Single Instruction Multiple Data: SIMD.
- The number of elements processed depends on the vector length of the architecture.

i) SSE(2,3,4): 128 bits (Westmere/Sandybridge) ii) AVX: 256 bits iii) AVX-512 KNC: 512 bits (XeonPhi)

E.g. SSE, Single precision; axpy operation.

for $i=0,\dots,L-1$
$y_i = a \cdot x_i + y_i$

SP: 4 bytes
=> 4 elements per 128 bit vector

=> for $i=0,4,8,\dots,L$.
$y_i = a x_i + y_i$
$y_{i+1} = a x_{i+1} + y_{i+1}$
$y_{i+2} = a x_{i+2} + y_{i+2}$
$y_{i+3} = a x_{i+3} + y_{i+3}$

20 Konstantinou Kavafi Street, 2121 Aglantzia, P.O. Box 27456, 1645 Nicosia, Cyprus
Tel.: +357 22208600 • Fax: +357 22208625
www.cyi.ac.cy

RESEARCH • TECHNOLOGY • INNOVATION

Need : vector y : $vy_i = \{y_i, y_{i+1}, y_{i+2}, y_{i+3}\}$

vector x : $vx_i = \{x_i, x_{i+1}, x_{i+2}, x_{i+3}\}$

vector a : $va = \{a, a, a, a\}$

Pseudo-code

```
va = vload1(a)
for i=0,4,8,...,L.
  vy = vload(y_i)
  vx = vload(x_i)
  vy = va * vx + vy
  vstore(y_i, vy)
```

- The number of vector-ops per cycle.
  - 0.5, 1, 2 | 0.5: 1 vector op every other cycle (old).
    1: 1 ——"—— every cycle.
    2: 1 multiply + 1 add op every cycle (madd op).

Example: SSE4 architeture, 1 madd per cycle, F=2 GHz.

Peak: 4 Flop (vector length) × 2 (*,+) × 2 GHz = 16 GFlop/sec. peak

γ_{FP}^{Peak}

γ_{Io} : More difficult to figure out.

- Read of from specs of vendor.
- Depends on RAM channels populated: for HPC systems, assume fully populated channels.

So: How fast can my kernel run?

$$T_{FP} = \frac{N_{FP}}{\gamma_{FP}}, \quad T_{I/O} = N_{Io}/\gamma_{Io} \Big\} \quad T = \max[T_{FP}, T_{Io}]$$

Assumes: I/O ops can be issued in parallel with FP ops.

How far is T measured from theoretical $T = \max[T_{FP}, T_{Io}]$?

Remarks: - Some systems may not yield γ_{Io}^{Peak}. - Maximum data reuse must apply.
- Assumption of fixed ops in γ_{FP}^{Peak}

20 Konstantinou Kavafi Street, 2121 Aglantzia, P.O. Box 27456, 1645 Nicosia, Cyprus
Tel.: +357 22208600 • Fax: +357 22208625
www.cyi.ac.cy

RESEARCH • TECHNOLOGY • INNOVATION

3) How to get to optimal performance

Remark: In the same way we defined "computational intensity" for a kernel, we can determine a similar ratio for the machine: $I_m = \frac{\gamma_{FP}}{\gamma_{I/O}}$.

a) $I_m > I_k \Rightarrow$ The kernel is said to be "bandwidth-bound" on this machine
(Most typical situation nowadays for lattice kernels.) i.e. at peak bandwidth, the F.P. units of the machine will be under-utilized.

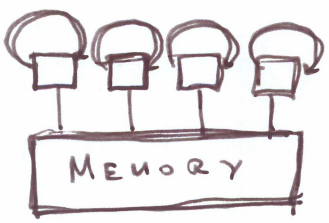
b) $I_m < I_k \Rightarrow$ The kernel is said to be "computational-bound" on this machine.
i.e. We can reach Peak F.P. at less than peak BW.

c) $I_m = I_k \Rightarrow$ The kernel is balanced. (Ideal situation!)

Depending on where kernel is (a, b, or c?), we optimize for better BW utilization, FP utilization or both.

Optimization Strategies covered here.

Multi-core system:



1) Effectively multiply γ_{FP}.
2) Closer to γ_{IO}.

1) OpenMP for multi-core processing.

2) Vectorization of kernels for better FP and I/o utilization.

Other optimizations: - Cache locality (spacial, temporal)
→ Not covered, but some indirect insight from exercises.
- Optimization for distributed memory systems (MPI, overlap comm.-comp.)
- Optimization for streaming processors (GPUs)

RESEARCH • TECHNOLOGY • INNOVATION

1) Open-MP for multi-processing.

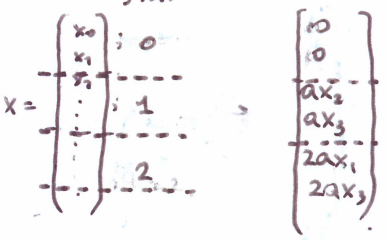
Add pragmas to your code:

```
for(i=0;i<N;i++)                    #pragma omp for
    x[i] = a*x[i];          ⟶      for(j=0;j<N;j++)
                                        x[j]=a*x[j].
```

} Split N over number of open-mp threads, which is controlled by OMP_NUM_THREADS variable.

```
double a = 2.0; ← local variable.
#pragma omp parallel.
{
   int ithr = omp_get_thread_num();
   double b = ithr*a;
#pragma omp for
   for(int j=0; j<N; j++)
        x[j] = b*x[j];
}.
```

ithr



Full API and pragma listing.

→ web. OpenMP.org.

Google: openmp reference.

2) Vectorisation

a) Bring kernel to vectorizable form.
b) Implement with intrinsics.

} Improve both FP AND BW performance

BW: Improved because of less shuffles, simpler load/stores

Remark:

If code is in vectorizable form (a), then implementing (b) may not be necessary - may be handled by compiler auto-vectorization.

RESEARCH • TECHNOLOGY • INNOVATION

Short vectorization how-to:

- Here for Intel intrinsics. Should work for gcc/icc.
- Similar for other vector types (e.g. Altivec)

Includes: <xmmintrins.h>
<e — n — >

Types: __m128 (128-bit floating point, holds 4)
__m128d (128-bit double, holds 2)

Functions:
_mm_load_ps (*float) : load from memory (s.p)
_mm_load_pd (*double) : load from memory (d.p).

_mm_load1_p(s/d) (*float / *double)

_mm_store_p(s/d) (single * / double *, __m128d / __m128)

_mm_mul_ps/d (__m128/d, __m128/d).
_mm_add_ps/d (--- ---)

Shuffle:

C = _mm_shuffle_ps(a, b, mask) mask: 8 bits.

$a = (a_3, a_2, a_1, a_0)$

$b = (b_3, b_2, b_1, b_0)$

mask = $(x_7|x_6|x_5|x_4|x_3|x_2|x_1|x_0)$

$C[0] = a[x_1 x_0]$, $C[1] = a[x_3 x_2]$.

$C[2] = b[x_5 x_4]$, $C[3] = b[x_7 x_6]$.

Example: a=[0,1,2,3], b=[4,5,6,7], mask=0b00011011

C[0] = a[3], C[1] = a[2], C[2] = b[1], C[3] = b[0], C = [3, 2, 5, 4].

More info: software.intel.com/sites/landingpage/IntrisicsGuide/

20 Konstantinou Kavafi Street, 2121 Aglantzia, P.O. Box 27456, 1645 Nicosia, Cyprus
Tel.: +357 22208600 • Fax: +357 22208625
www.cyi.ac.cy