CPSC 424 | HW1

kd538 - Keyang Dong

Exercise 1

Code

```
// Makefile
IC = icc
IFLAGS1 = -g -O0 -fno-alias -std=c99
IFLAGS2 = -g -O1 -fno-alias -std=c99
IFLAGS3 = -g -O3 -no-vec -no-simd -fno-alias -std=c99
IFLAGS4 = -g -O3 -xHost -fno-alias -std=c99

pi_1 pi_2 pi_3 pi_4: pi.c timing.o
    ${IC} ${IFLAGS1} -o pi_1 pi.c timing.o
    ${IC} ${IFLAGS2} -o pi_2 pi.c timing.o
    ${IC} ${IFLAGS3} -o pi_3 pi.c timing.o
    ${IC} ${IFLAGS4} -o pi_4 pi.c timing.o

// pi.c
#include <stdio.h>
#include <math.h>
#define N 1000000000
extern void timing(double* wcTime, double* cpuTime);

double approx_pi() {
    double sum = 0., slice = 1.0 / N, x;
    int i;
    x = slice / 2;
    for (i = 1; i < N; i++) {  // each cycle contains 6 FP operations
        sum += slice / (1.0 + x * x);
        x += slice;
    }
    return 4.0 * sum; // total FP operations: 4N
}

int main() {
    double t1, t2, cpu_t, pi;
    timing(&t1, &cpu_t);
    pi = approx_pi();
    printf("Pi=%lf, sin(Pi)=%lf\n", pi, sin(pi));
    timing(&t2, &cpu_t);
    printf("Time spent: %lf, ", t2 - t1);
    printf("MFlops/s = %lf\n", (4 * (N / 1000000)) / (t2 - t1));
    return 0;
}
```

Execution

```
[kd538@compute-32-13 hw1-1]$ make
icc -g -O0 -fno-alias -std=c99 -o pi_1 pi.c timing.o
icc -g -O1 -fno-alias -std=c99 -o pi_2 pi.c timing.o
icc -g -O3 -no-vec -no-simd -fno-alias -std=c99 -o pi_3 pi.c timing.o
icc -g -O3 -xHost -fno-alias -std=c99 -o pi_4 pi.c timing.o
[kd538@compute-32-14 hw1-1]$ ./pi_1
Pi=3.141593, sin(Pi)=0.000000
Time spent: 7.186978, MFlops/s = 556.562171
[kd538@compute-32-14 hw1-1]$ ./pi_2
Pi=3.141593, sin(Pi)=0.000000
Time spent: 7.186214, MFlops/s = 556.621333
[kd538@compute-32-14 hw1-1]$ ./pi_3
Pi=3.141593, sin(Pi)=0.000000
Time spent: 7.186769, MFlops/s = 556.578345
[kd538@compute-32-14 hw1-1]$ ./pi_4
Pi=3.141593, sin(Pi)=0.000000
Time spent: 3.592844, MFlops/s = 1113.324149
```

Explain the results

As Xeon is a processor with 4 cores and supports at most 8 threads, it can support some sorts of parallel computing based optimization. In the code piece that approximates Pi, the increment of x and the summing up of result could be overlapped a bit: Xeon don't have to wait several dozens of cycles (6 FP operations in each iteration) for the area value of a new slice of rectangle; instead, it can assign consecutive loaded FP operations to different cores, move on to increment x, and gather the previous results later to sum up.

Also, as Xeon is also consist of an instruction pipeline, with optimization options set before compilation, a FP instruction can even prefetch a new x value that has been generated from ALU, but not yet written back to register. This brings more speed up.

Estimate Division Latency in CPU Cycles

According to Page.27 on Slide 02, obtaining per results from floating point ADD and MUL takes about 1 cycles, while DIV takes about 44 cycles, a lot more than other arithmatic operations. So by estimation we can simply ignore ADD and SUB's execution time. As there are N DIV operations in total, using the fourth optimization option, the running time is T=7.2s, we have an approximated division latency of `cycles = (T/N) * freq = 7.2ns * 2.8GHz = 20.16`.

Exercise 2

Code

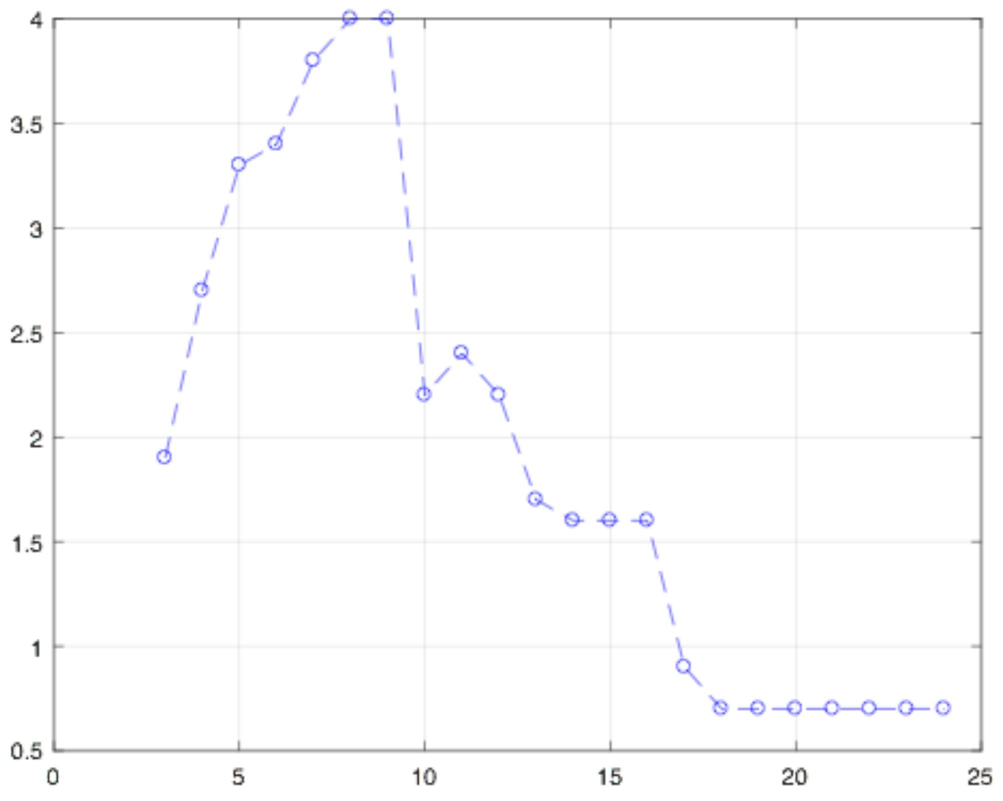
```
// Makefile
```

```
IC = icc
IFLAGS = -g -O3 -xHost -fno-alias -std=c99

triad: triad.c dummy.c timing.o
    ${IC} ${IFLAGS} -o triad triad.c dummy.c timing.o

// dummy.c
void dummy() { int x = 0; }

// triad.c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

extern void timing(double* wcTime, double* cpuTime);
extern void dummy();

void kernel(int N, double a[], double b[], double c[], double d[]) {
    int i = 0;
    for (; i < N; i++) {
        a[i] = b[i] + c[i] * d[i];
    }
}

double* init(int N) {
    double* arr = (double *) malloc(sizeof(double) * N);
    double drand_max = 100.0 / (double) RAND_MAX;
    int i = 0;
    for (; i < N; i++) {
        arr[i] = drand_max * (double) rand();
    }
    return arr;
}

int main() {
    int k, r, repeat;
    long N;
    double wcs, wce, ct, runtime;
    double *a, *b, *c, *d;

    for (k = 3; k <= 24; k++) {
        N = floor(pow(2.1, k));
        a = init(N), b = init(N), c = init(N), d = init(N);
        runtime = 0.;
        repeat = 1;
        while (runtime < 1.0) {
            timing(&wcs, &ct);
            for (r = 0; r < repeat; r++) {
                kernel(N, a, b, c, d);
                if (a[N>>1] < 0.) { dummy(); }  // fool the compiler
            }
            timing(&wce, &ct);
            runtime = wce - wcs;
            repeat *= 2;
        }
        repeat /= 2;
        // 2 * N * repeat FP operations in total
        printf("N=%ld, log(N) = %d, MFlops/s = %2.1lf G, repeat=%d, runtime=%lf\n"
, N, k, (2 * N * repeat/pow(10, 9)) / runtime, repeat, runtime);

    }
    return 0;
}
```

Execution

```
[kd538@compute-32-14 hw1-2]$ make
icc -g -O3 -xHost -fno-alias -std=c99 -o triad triad.c dummy.c timing.o
[kd538@compute-32-14 hw1-2]$ ./triad
N=9, log(N) = 3, MFlops/s = 1.9 G, repeat=134217728, runtime=1.271246
N=19, log(N) = 4, MFlops/s = 2.7 G, repeat=134217728, runtime=1.906999
N=40, log(N) = 5, MFlops/s = 3.3 G, repeat=67108864, runtime=1.622013
N=85, log(N) = 6, MFlops/s = 3.4 G, repeat=33554432, runtime=1.698615
N=180, log(N) = 7, MFlops/s = 3.8 G, repeat=16777216, runtime=1.580828
N=378, log(N) = 8, MFlops/s = 4.0 G, repeat=8388608, runtime=1.589102
N=794, log(N) = 9, MFlops/s = 4.0 G, repeat=4194304, runtime=1.678058
N=1667, log(N) = 10, MFlops/s = 2.2 G, repeat=1048576, runtime=1.585125
N=3502, log(N) = 11, MFlops/s = 2.4 G, repeat=524288, runtime=1.533458
N=7355, log(N) = 12, MFlops/s = 2.2 G, repeat=262144, runtime=1.771631
N=15447, log(N) = 13, MFlops/s = 1.7 G, repeat=65536, runtime=1.212527
N=32439, log(N) = 14, MFlops/s = 1.6 G, repeat=32768, runtime=1.311171
N=68122, log(N) = 15, MFlops/s = 1.6 G, repeat=16384, runtime=1.381107
N=143056, log(N) = 16, MFlops/s = 1.6 G, repeat=8192, runtime=1.465461
N=300419, log(N) = 17, MFlops/s = 0.9 G, repeat=2048, runtime=1.363523
N=630880, log(N) = 18, MFlops/s = 0.7 G, repeat=1024, runtime=1.960570
N=1324849, log(N) = 19, MFlops/s = 0.7 G, repeat=256, runtime=1.013237
N=2782184, log(N) = 20, MFlops/s = 0.7 G, repeat=128, runtime=1.064113
N=5842587, log(N) = 21, MFlops/s = 0.7 G, repeat=64, runtime=1.128019
N=12269432, log(N) = 22, MFlops/s = 0.7 G, repeat=32, runtime=1.174433
N=25765808, log(N) = 23, MFlops/s = 0.7 G, repeat=16, runtime=1.224998
N=54108198, log(N) = 24, MFlops/s = 0.7 G, repeat=8, runtime=1.296434
```

Plot



Explanation

We can see some obvious downhill slopes in the plotted curve, a major one from k=9-10; as the total size of four arrays involved in major part of computation occupies `N*8*4`, where 8 is the number of bytes a word (also size of a double precision floating point number) holds, and the number range from 25-53KB, indicating it could be caused by the bandwidth of cache read/write. The cache could have a cache line (or block) size of 32KB, and when one read cannot fit the whole required arrays into the cache, the performance significantly decreases.

Meanwhile, as the last slope ranging from k=13 to 17 suggests, indicates that the size of cache may be close to 8MB, while more cache misses emerge when read from cache lines, as the size of arrays getting larger. After exceeding this bound, the performance will be correspondingly steady around the worse case, there are always some ratio of cache misses happening.