High Performance Computing
Homework 1

Understanding the Programming Platform

Report

Peiyun Hu, 2010011297

Department of Computer Science & Technology

Sep. 22, 2012

1 Environment

1.1 Hardware

1.1.1 CPU

Model Name: Intel(R) Core(TM)2 Quad CPU @ 2.93GHz
CPU MHz: 1600.000
Cores: 4
But actually, only one core is involved in this task. What is more, the '1600 MHz' means that at which the processor is running right now, and '2.93 GHz' is the maximum CPU Speed.

1.1.2 Memory

Total Memory: 4055940 KB

1.1.3 Cache

Cache Size: 4096 KB

1.2 Software

1.2.1 OS

Linux version 2.6.38-8-generic

1.2.2 Compiler

gcc version 4.5.2 (Ubuntu/Linaro 4.5.2-8ubuntu4)

1.2.3 Compiler Options

Please refer to makefile.

1.2.4 Timing Method

As for the timing method, only meaningful calculation time are counted, which means that time spent in initialization and output are excluded from the final time.

Additionally, there are minute differences under different conditions. Firstly, in the vector-to-vector($\vec{a}, \vec{b}$) computation, one float operations are involved in each pair of elements, which is a multiplication of $a_i \times b_i$. Secondly, in the matrix-to-vector calculation, two float operations need to be counted in each pair, which is a multiplication and an addition. Besides, matrix-to-matrix condition is similar with the m2v's condition.

Consequently, assuming that the problem number of rows and columns are both n, we could infer that:

$$\text{FLOPS}_{v2v} = \frac{n}{t} \tag{1}$$

$$\text{FLOPS}_{m2v} = \frac{2n^2}{t} \tag{2}$$

$$\text{FLOPS}_{m2m} = \frac{2n^3}{t} \tag{3}$$

2 Performance

2.1 Theoretical Peak Performance

As is shown in 1.1.1, it is known that A Core 2 Quad has 4 cores. HPC world uses the following formulae for node theoretical peak performance:

Node performance in GFLOPS = (CPU speed in GHz) x (number of CPU cores) x (CPU instruction per cycle) x (number of CPUs per node). [2]

In this sequential task, the node performance is as below:

$$\text{Performance} = 2.93\text{GHz} \times 1 \times 4 \times 1 = 11.72\text{GHz} \tag{4}$$

An important note about CPU instruction per cycle is 4, as is shown in [1].

Moreover, considering there are two indicators of cpu speed, we choose the higher one, because the cpu is smart enough to gear up when more calculation is needed. So the indicator '2.93 GHz' is more appropriate.

2.2 Experiment Results

2.2.1 Vector-Vector

The efficiency in varied size of dataset is shown as Figure 1.As we can see from the chart, in small dataset, the performance of float rivals that of double. Because the dataset is rather small that the time of timing is not short enough to be ignored, even timing possessed most of time consumed.

When dataset gets bigger, float is almost twice faster than double, owing to their length.

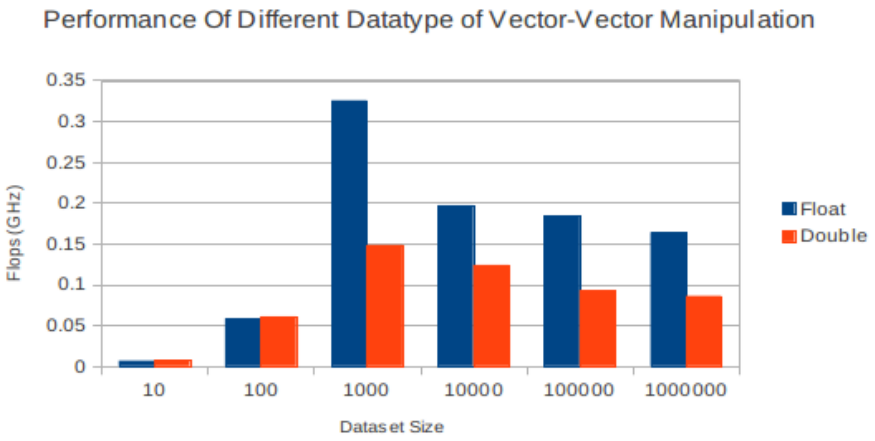

Figure 1: Performance of Vector-Vector Manipulation With Different Implements

2.2.2 Matrix-Vector

The efficiency in varied size of dataset is shown as Figure 2. As shown in the chart, the efficiency of float calculation is rather stable. And the double is increasing gradually, but stay lower than float operation.

2.2.3 Matrix-Matrix

The efficiency in varied size of dataset is shown as Figure 3. Two methods of multiplication are involved.

The first one is simple and intuitive. After reading in the matrix, do the multiplicaiton directly like below:

```
1 for(i=0; i<a_rows; i++)
2     for(j=0; j<b_cols; j++)
3         for(k=0; k<a_cols; k++)
```

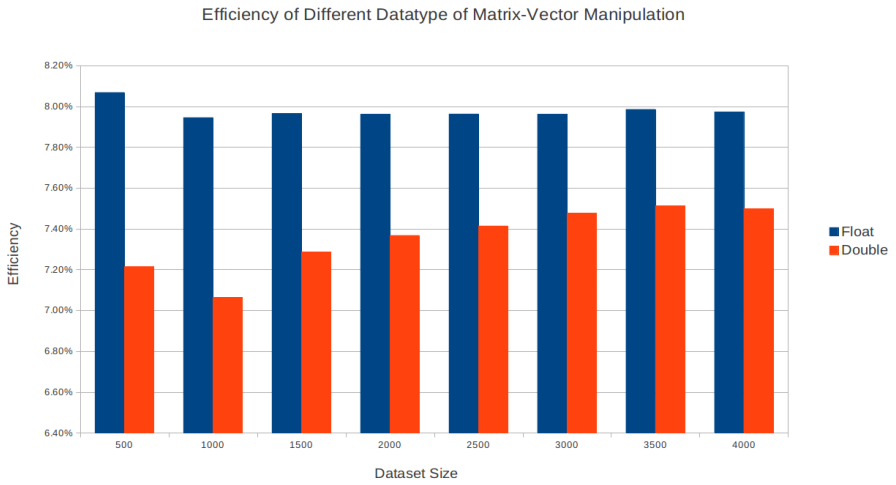

Figure 2: Efficiency of Matrix-Vector Manipulation With Different Implements

```
4                 res[i*a_cols+j] += v1[i*a_cols+k] * v2[k*b_cols+j];
```

But when considering taking advantage of cache, we can make it better by preprocessing the second matrix v_2. By transposing v_2, we can achieve a better hit rate when accessing v_2, owing to the linear arrangement of the array in the physical storage. And the code is a little different from above:

```
1 for(i=0; i<a_rows; i++)
2     for(j=0; j<b_cols; j++)
3         for(k=0; k<a_cols; k++)
4             res[i][j] += v1[i*a_cols+k] * v2[j*a_cols + k];
```

We can notice that the index of each array now increment by 1.And, we got almost 4 times faster than that of former method as is shown in 3.

What is more **interesting** is, in Figure 3, the float implement of first method is nearly on the same level with the second method. To interprete this, we should review 1.1.3. The cache can hold:

$$\frac{4096 * 1024}{500^2 * 2 * 4} = 2.097 \tag{5}$$

32-bit single-precision floating numbers. That is to say, without the optimization of memory access, CPU could still reach a high hit rate.

But cache is not big enough to store double-precision floating matrixs of this size at one time. That is why we see a sharp contrast between performance of float and performance of double, in the first method.

3 Further Discussion

Besides, noticing that SSE, which was firstly introduced in Pentium III, added eight new-bit registers known as XMM0 through XMM7, the cpu could handle 4 32-bit float at one time. Owing to SSE2 introduced later since Pentium Series, the usage of XMM is expanded to 64-bit double-precision mode. Additionally, the AMD64(originally called x86-64) added further eight registers from XMM8 to XMM15.

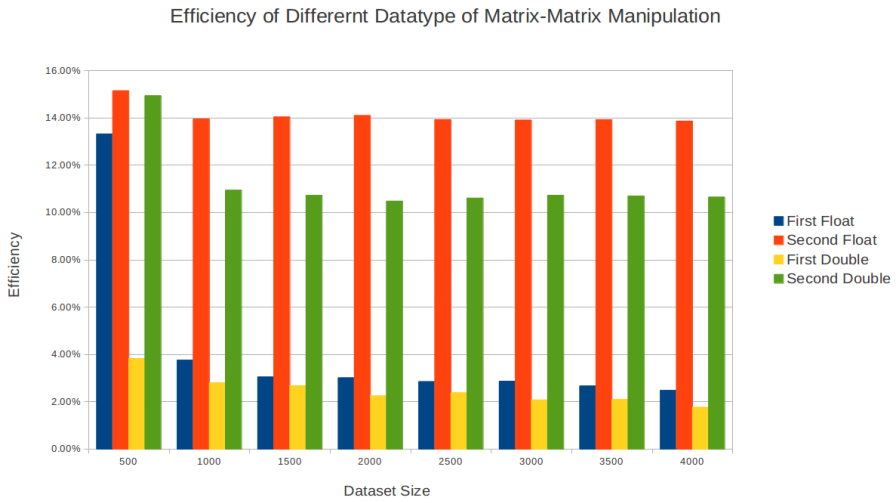

Figure 3: Efficiency of Matrix-Matrix Manipulation With Different Implements

So with SSE programmed inline, we could further enhance the performance. However, it is a little embarassing in this task, for I did not find a easy way to mutiply using SSE.

References

[1] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, November 2009. 2.1 Intel Core Microachitecture and Enhanced Intel Core Microachitecture, Intel Wide Dynamic Execution, Page 30.

[2] NOVATTE. How to calculate theoretical peak performance of a cpu-based hpc system. http://novatte.com/blog/2012/03/how-to-calculate-theoretical-peak-performance-of-a-cpu-based-hpc-system/.

Appendix

A Implement of Vector-Vector

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5
6  #define start_time clock_gettime(CLOCK_MONOTONIC, &start);
7  #define end_time clock_gettime(CLOCK_MONOTONIC, &finish);
8
9  struct timespec start, finish;
10
11 int main(int argc, char *argv[]){
12     int output_switch;
13     char *input_file_name;
14     char *output_file_name;
```

```
15      FILE* pfile;
16
17      /* resolve the input arguments */
18      if (argc < 2){
19          printf("Error! the number of arguments is less(%d)!\n",
                argc);
20          system("pause");
21          return 0;
22      }
23      input_file_name = argv[1]; // get input file name
24
25      output_switch = 0;      // output results or not
26      if (argc > 2) {
27          output_switch = 1;
28          output_file_name = argv[2];
29      }
30
31      if ((pfile = fopen(input_file_name, "rb")) == NULL){
32          printf("file: %s can not be opened \n", input_file_name);
33          system("pause");
34          return 0;
35      }
36
37      int size;
38      float *v1, *v2, *res;
39
40  // set size
41      fread((void*)(&size), sizeof(int), 1, pfile);
42
43  // alloc
44      v1 = (float*)malloc(size * sizeof(float));
45      v2 = (float*)malloc(size * sizeof(float));
46      res = (float*)malloc(size * sizeof(float));
47
48  // read in data
49      fread((void*)v1, sizeof(float), size, pfile);
50      fread((void*)v2, sizeof(float), size, pfile);
51
52  // set start
53      start_time
54
55  // calculate
56      int i;
57      for(i=0; i<size; i++)
58          res[i] = v1[i] * v2[i];
59
60  // set end
61      end_time
62      printf("%d %.16lf\n", size, finish.tv_sec-start.tv_sec + (
            double)(finish.tv_nsec - start.tv_nsec) / 1000000000.0);
63
64  #ifndef SILENT
65      for (i = 0; i < a_rows; i ++){
66          printf("%f ", v1[i]);
67      }
68      printf("\n");
```

```
#endif

    /* output results */
    if (output_switch){
        if ((pfile = fopen(output_file_name, "wb")) == NULL){
            printf("file: %s can not be opened \n",
                output_file_name);
            system("pause");
            return 0;
        }
        fwrite(res, sizeof(float), size, pfile);
        fclose(pfile);
    }
}
```

B Implement of Matrix-Vector

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <memory.h>

#define start_time clock_gettime(CLOCK_MONOTONIC, &start);
#define end_time clock_gettime(CLOCK_MONOTONIC, &finish);

struct timespec start, finish;

int main(int argc, char *argv[]){
    int output_switch;
    char *input_file_name;
    char *output_file_name;
    FILE* pfile;

    /* resolve the input arguments */
    if (argc < 2){
        printf("Error! the number of arguments is less(%d)!\n",
            argc);
        system("pause");
        return 0;
    }
    input_file_name = argv[1]; // get input file name

    output_switch = 0;     // output results or not
    if (argc > 2) {
        output_switch = 1;
        output_file_name = argv[2];
    }

    if ((pfile = fopen(input_file_name, "rb")) == NULL){
        printf("file: %s can not be opened \n", input_file_name);
        system("pause");
        return 0;
```

```
117     }
118
119     int a_rows, a_cols, b_cols;
120     float *v1, *v2, *res;
121
122 // set size
123     fread((void*)(&a_rows), sizeof(int), 1, pfile);
124     fread((void*)(&a_cols), sizeof(int), 1, pfile);
125
126 // alloc
127     v1 = (float*)malloc(a_rows*a_cols * sizeof(float));
128
129     v2 = (float*)malloc(a_cols * sizeof(float));
130     res = (float*)malloc(a_cols * sizeof(float));
131
132 // read in data
133     fread((void*)v1, sizeof(float), a_rows*a_cols, pfile);
134     fread((void*)v2, sizeof(float), a_cols, pfile);
135 // memset
136     memset(res, 0.0, a_rows * sizeof(float));
137
138 // set start
139     start_time
140
141 // calculate
142     int i, j;
143     for(i=0; i<a_rows; i++)
144             for(j=0; j<a_cols; j++)
145                     res[i] += v1[i*a_cols+j] * v2[j];
146
147 // set end
148     end_time
149     printf("%d %d %.16lf\n", a_rows, a_cols, finish.tv_sec-start.
        tv_sec + (float)(finish.tv_nsec - start.tv_nsec) /
        1000000000.0);
150
151     printf("a: %d %d \n", a_rows, a_cols);
152 #ifndef SILENT
153     for (i = 0; i < a_rows; i ++){
154         for (j = 0; j < a_cols; j ++)
155             printf("%f ", v1[i*a_cols+j]);
156             printf("\n");
157     }
158 #endif
159
160     /* output results */
161     if (output_switch){
162         if ((pfile = fopen(output_file_name, "wb")) == NULL){
163             printf("file: %s can not be opened \n",
                    output_file_name);
164             system("pause");
165             return 0;
166         }
167         fwrite(res, sizeof(float), a_cols*b_cols, pfile);
168         fclose(pfile);
169     }
```

```
}
```

C Implement of Matrix-Matrix

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <memory.h>

#define start_time clock_gettime(CLOCK_MONOTONIC, &start);
#define end_time clock_gettime(CLOCK_MONOTONIC, &finish);

struct timespec start, finish;

int main(int argc, char *argv[]){
    int output_switch;
    char *input_file_name;
    char *output_file_name;
    FILE* pfile;

    /* resolve the input arguments */
    if (argc < 2){
        printf("Error! the number of arguments is less(%d)!\n",
           argc);
        system("pause");
        return 0;
    }
    input_file_name = argv[1]; // get input file name

    output_switch = 0;     // output results or not
    if (argc > 2) {
        output_switch = 1;
        output_file_name = argv[2];
    }

    if ((pfile = fopen(input_file_name, "rb")) == NULL){
        printf("file: %s can not be opened \n", input_file_name);
        system("pause");
        return 0;
    }

    float *v1, *v2, *res;
    int a_rows, a_cols, b_cols;

// set size
    fread((void*)(&a_rows), sizeof(int), 1, pfile);
    fread((void*)(&a_cols), sizeof(int), 1, pfile);
    fread((void*)(&b_cols), sizeof(int), 1, pfile);
// alloc
    v1 = (float*)malloc(a_rows* a_cols * sizeof(float));
    v2 = (float*)malloc(a_cols* b_cols* sizeof(float));
```

```
219     res = (float*)malloc(a_rows*b_cols* sizeof(float));
220
221 // read in data
222     fread((void*)v1, sizeof(float), a_rows*a_cols, pfile);
223     fread((void*)v2, sizeof(float), a_cols*b_cols, pfile);
224
225     fclose(pfile);
226
227 // memset
228     memset(res, 0.0, a_rows * b_cols * sizeof(float));
229 // set start
230     start_time
231
232 // calculate
233     int i, j, k;
234     for(i=0; i<a_rows; i++)
235         for(j=0; j<b_cols; j++)
236             for(k=0; k<a_cols; k++)
237                 res[i*a_cols+j] += v1[i*a_cols+k] * v2[k*b_cols+j
                        ];
238
239 // set end
240     end_time
241     printf("%d %d %d %.16lf\n", a_rows, a_cols, b_cols, finish.
           tv_sec-start.tv_sec + (double)(finish.tv_nsec - start.
           tv_nsec) / 1000000000.0);
242
243     printf("a: %d %d \n", a_rows, a_cols);
244     printf("b: %d %d \n", a_cols, b_cols);
245     printf("c: %d %d \n", a_rows, b_cols);
246 #ifndef SILENT
247     for (i = 0; i < a_rows; i ++){
248         for (j = 0; j < a_cols; j ++)
249             printf("%f ", v1[i*a_cols+j]);
250             printf("\n");
251     }
252 #endif
253
254     /* output results */
255     if (output_switch){
256         if ((pfile = fopen(output_file_name, "wb")) == NULL){
257             printf("file: %s can not be opened \n",
                        output_file_name);
258             system("pause");
259             return 0;
260         }
261         fwrite(res, sizeof(float), a_cols*b_cols, pfile);
262         fclose(pfile);
263     }
264 }
```