Name : Mincheol Sung

Task 1.3

Input the number

With “-O2”, it costs 21507 cycles,

Without “-O2”, it costs 63109 cycles.

With “-O2”, without printing sum out, it costs 1379 cycles.

Without “-O2”, without printing sum out, it costs 63114 cycles.

With Hardcoding the number directly into the program,

With “-O2” and printing sum out, it costs 1582 cycles.

Without “-O2” and printing sum out, it costs 66707 cycles.

With “-O2”, without printing sum out, it costs 1440 cycles.

Without “-O2”, without printing sum out, it costs 66776 cycles.

Explain the discrepancy in the timings,

Without printing sum out, and hardcoding the number directly in to the program, it is the most accurate way of timing. Because, such i/o takes too much time compared to arithmetic computing.

Also “atoi(argv[1])” works as a keyboard i/o, so it takes too much time. “argv[1]” is an input by keyboard when the main starts, and “atoi()” is a function that changes char into int.

Task 2.3

Before requesting a value from memory, getTick() gets the ticks(CPU time) at that time, and after obtaining the value from memory, the benchmark calculate the difference between the ticks.

If array’s length is N with integer type, each integer is 4 bytes , so the array’s size is 4N bytes.

This is a plot showing the relationship between the average number of cycles to execute a step (vertical axis) and the size in bytes of the array(horizontal axis). The size is varied from 32KB to 8MB. When the array’s size is 8MB, there is a segmentation fault. (compiled with no option)

There is increasing curve. When the memory size is big, the cycles per on step increase. However, the number of cycles per one step changed a lot like 10, 15, 20. The reason is that with rand() function which makes random number doesn’t make numbers randomly. It means that there is a circulation among the random numbers, then it make the process just approaches few memories. For example, rand() makes 1,0,2,4,4,7,9,2,5,6, in this case i starts as 0. So, next step, i = array[0], so becomes 1. Next step, i = array[1] is 0, again i

becomes 0. There would be approaches between array[0] and array[1]. Because of this, the number of cycles per one step changes a lot.

There would be overheads in the executions, so we can’t be sure that a single step is actually completing in t/c cycles.

Task 3.4

(compiled –msse4 option only)

When the size of array is over 2MB, there occurs segmentation fault.

Between the size of array 256KB and 512KB, there is a sudden increase in bandwidth. It is because, even though the size of array increases, the cache volume is limited.

When the first memory approach happens, there are no values in the cache. It is called “cold miss”. After a few copy executions, the cache stores values, so there would be “cache hit”. It makes execution fast.

Because of the cache, the measurement of maximum bandwidth with inefficient array copying procedure is incorrect. It is because there may be cache miss or hit and limited cache size.

\_mm\_prefetch(char const\* p, int i) : this intrinsic fetches the line of date from memory that contains address p to a location in the cash hierarchy.

\_mm\_load\_si128(\_\_m128i const\* mem\_addr) : this loads 128-bits of integer date from memory into dst.mem\_addr.

\_mm\_stream\_si128(\_\_m128i\* mem\_addr, \_\_m128i a) : this stores 128-bits of integer data from a into memory using a non-temporal memory hint where mem\_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.

\_mm\_store\_si128(\_\_m128i\* mem\_addr, \_\_m128i a) : this stores 128-bits of integer date from a into memory where mem\_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.

Task 4

|  |  |  |
| --- | --- | --- |
|  | Flops | IPC |
| opt\_simd\_sgemm | 568593644 | 2.346597 |
| opt\_scalar1\_sgemm | 2331996861 | 2.598341 |
| opt\_scalar0\_sgemm | 1961191403 | 0.731835 |
| naïve\_sgemm | 2620392286 | 0.328642 |

(compiled with –msse4 and –O2 option)

To multiply two square matrices of size n, first multiply each rows and columns. It costs n^2 times, and then add each multiplied elements; it costs 2n-1 times.

Total is n^2 \* (2n -1) times. In addition, arithmetic intensity would be O(n^3).

|  |  |
| --- | --- |
|  | Total number of floating point operation / time |
| opt\_simd\_sgemm | 2146435072 |
| opt\_scalar1\_sgemm | 1073217536 |
| opt\_scalar0\_sgemm | 1073217536 |
| naïve\_sgemm | 214643507.2 |

Exceeding 1 in IPC, for example 2.3 from opt\_simd\_sgemm or 2.5 from opt\_scalar1\_sgemm, means that the processor is completing more than one instruction per cycle. This can be possible because of the pipeline or superscalar architectures. These architectures can execute several instructions at the same time in one cycle.

That is power of simd(Single Instruction Multiple Data). opt\_simd\_sgemm executes fewer instructions than opt\_scalar1\_sgemm and we can see with the IPC, instructions of opt\_simd\_sgemm can compute multiple data. So, a higher IPC isn’t always an indicator of an efficient program.