Part 1.

Given code test\_align.c was compiled and ran.

In the given code the starting address is 140611251064896. To ensure the proper working of alignment, we need to check if the address is divisible by 8 bytes.. So, we increment the address by k such that k increments from 0 to Boundary Alignment. Whenever start address + current value of k is an address which is not divisible by 8, misalignment occurs.

Given file was modified by changing the type casting to long double so as to simulate two cache misses per fetch. The difference in performance between casting long and long double is illustrate below.

Part 2.

Test\_combine8.c was compiled and ran using float and +. The graph was plotted as given below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | C4 | C6\_5 | C8 | C8\_4 |
| CPE | 11.16 | 3.39 | 2.47 | 0.5 |

The vector results are classified by c8\_4 and c8.

C8\_4 has 4 vector accumulators which add 16 elements in parallel. This has a better performance than scalar unrolled with 5 accumulators. Also c6\_5 in the base vector code has a CPE of 3.33 which is better than the base scalar code (C4) which has a CPE of 11.19.

From the above graph we see that the best performance is given by vector unrolling with 2 accumulators. As the number of accumulators increase the performance reduces due to an increase in the number of operations.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Method | c4 | c6\_5 | c8 | c8\_2 | c8\_4 | c8\_8 |
| CPE | 10.33 | 3.39 | 2.78 | 0.24 | 5.44 | 2.8 |

Test\_dot.c was compiled and ran using float data type. The default functions and graphs and CPEs were calculated. New functions were written to perform vector unrolling by a factor 4.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Method | d4 | d5 | d6\_2 | d6\_5 | d8 | d8\_2 | d8\_4 |
| CPE | 9.64 | 5.11 | 4.7 | 2.96 | 3.47 | 0.94 | 1.05 |

From above graph and CPE table we can see that base vector dot product has a better performance over a scalar dot product because in vectorized code we are traversing 4 elements in one cycle and operating on all of them instead of operating on individual elements in scalar dot product.

Part 3:

1.

The performance of the intrinsic version is significantly better as SSE vectorizes the code and the data in the array is processed in segments or chunks of 128 bits. Hence SSE version has a much lower CPE (6.24) compared to the non SSE version (27.77).

2. As we can see, there wasn’t much difference in element wise addition and element wise multiplication. The code was optimized and tried on different array size and this was the optimal performance of the code. Both had comparable CPE 0.81(Addition) 0.86(Multiplication)

3.

A vectorized dot product was created using intrinsic. The graph was plotted as above. The CPE was calculated to be 2.39.

Comparing intrinsic dot product with the dot product from Part 2 shows that intrinsic sse has a better performance as shown in the graph. This is because in SSE intrinsic the data is aligned properly whereas in the vector version we have to manually take care of alignment.

4. Comparing SSE intrinsic with vector method, I came to a conclusion that intrinsic method has a better performance since SSE intrinsic programming takes care of alignment. The only disadvantage of intrinsic is that it evaluates only particular elements which are stored in an address register which is a multiple of 4,8. etc. which is not ideal.

Part 4.

The fastest version of Transpose was created by reusing the modified transpose\_rev function in test\_transpose.c as in Lab 1. Loop blocking was applied to the same function in another file called test\_transpose\_block.c. O2, O3 optimizations were applied to the above code.

SSE Transpose Intrinsic was used to implement transpose and transpose in blocking in test\_intrinsics.c under functions transpose4x4\_SSE and transpose\_block\_SSE4x4 respectively. O2, O3 optimizations were applied to the above code.

The values of lda and ldb are the width of the matrix. These need to be multiples of the block size. To find the values and allocate memory a round UP function and \_mm\_malloc function was used. Additionally \_MM\_TRANSPOSE4\_PS intrinsic macro was used whose arguments correspond rows of a 4x4 matrix. This defines our block size. I found the fastest way to transpose was to use intrinsic blocking with block size 16.

Given Below are the performance comparison of different transpose techniques (normal and intrinsic) with respect to their level of optimizations.

From above, it can be seen that irrespective of optimization type, the performance of intrinsic code is significantly better. The best version will be where block size is 16 with blocking using SSE intrinsic coding.