Project 2

|  |  |  |  |
| --- | --- | --- | --- |
| Flag | N | Time | Speed-Up |
| -O0 | 1024 | 12986686 | N/A |
| -O3 | 1024 | 11762395 | 1.10408517993 |

|  |  |  |  |
| --- | --- | --- | --- |
| Flag | N | Time | Speed-Up |
| -O3 | 16 | 6 | N/A |
| -O3 | 32 | 30 | 5 |
| -O3 | 64 | 260 | 8.66666666667 |
| -O3 | 128 | 2212 | 8.50769230769 |
| -O3 | 256 | 19528 | 8.82820976492 |
| -O3 | 512 | 1312502 | 67.211286358 |
| -O3 | 1024 | 11762395 | 8.96181110581 |
| -O3 | 2048 | 113681595 | 9.66483399002 |
| -O3 | 4096 | 1085812491 | 9.55134814039 |

From array size N 256 to 512 and later, the time increases sharply. The reason for that is because the cache can’t hold that many blocks once it makes prediction. It simply means that the capacity miss increases once the array size increases. For example, when the cache tries to predict, there are 20 blocks out there to be fetched for one row of the array. However, the cache only accepts first 10 blocks for one prediction. It costs one more miss prediction to get the rest of the 10 blocks. The more of blocks at one row of the array, the worse prediction it is with a small cache. That is why, the time increases dramatically.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Flag | N | Block Factor | Time | Speed-Up |
| -O3 | 16 | 8 | 10 | N/A |
| -O3 | 32 | 8 | 85 | 8.5 |
| -O3 | 64 | 8 | 564 | 6.63529411765 |
| -O3 | 128 | 8 | 4347 | 7.70744680851 |
| -O3 | 256 | 8 | 34610 | 7.96181274442 |
| -O3 | 512 | 8 | 299834 | 8.66321872291 |
| -03 | 1024 | 8 | 2543662 | 8.48356757406 |
| -O3 | 2048 | 8 | 20585895 | 8.09301510971 |
| -O3 | 4098 | 8 | 195196393 | 9.48204549766 |

When the size of the array is 256 or less, using the Dense Multiplication method is faster than with Blocking method. The reason for that is because the Dense Multiplication does not use the “for loops” as much as the Blocking method and have not reached the worst case of capacity miss yet. However, the average case of the Blocking method is the worst case of the Dense Multiplication. That is why the Dense Multiplication method is much faster than the Blocking mothed when the array size is small. When the size of the array gets larger, the Dense Multiplication is toward its worst case. On the other side, the Blocking method starts to decrease the capacity miss. That is why once the size of the array gets larger, the Blocking mothed runs faster than the Dense Multiplication.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Flag | N | Block Factor | Time | Speed-Up |
| -O3 | 2048 | 4 | 31749986 | N/A |
| -O3 | 2048 | 8 | 20845827 | 0.65656176982 |
| -O3 | 2048 | 16 | 18844296 | 0.90398409235 |
| -O3 | 2048 | 32 | 19151014 | 1.01627643718 |
| -O3 | 2048 | 64 | 21165623 | 1.10519594419 |
| -O3 | 2048 | 128 | 20824115 | 0.9838649682 |
| -O3 | 2048 | 256 | 21029235 | 1.00985011848 |
| -O3 | 2048 | 512 | 51277369 | 2.43838489607 |

The optimal factor of the block is 16 because that uses the smallest amount of time to complete the program. However, by looking at the time from the value of factor 16 to 256, they do not vary a lot from each other. That means using those numbers of blocking factor generates cache friendly code. Some of them even do not exceed the size of the cache block. Even if any of them exceed the size of the block, it is still a small amount of data. Using those values can maximize the performance of the program.

1. The performance of the grogram can significantly be affected by how to access the cache. If it is not cache friendly, such as the Dense Multiplication, there will be a lot of capacity misses, and negatively affecting the performance of the grogram. However, using the blocking method, and finding the correct blocking factor, the number of the capacity misses can reduce. The whole grogram can run much faster with blocking method than without blocking mothed.