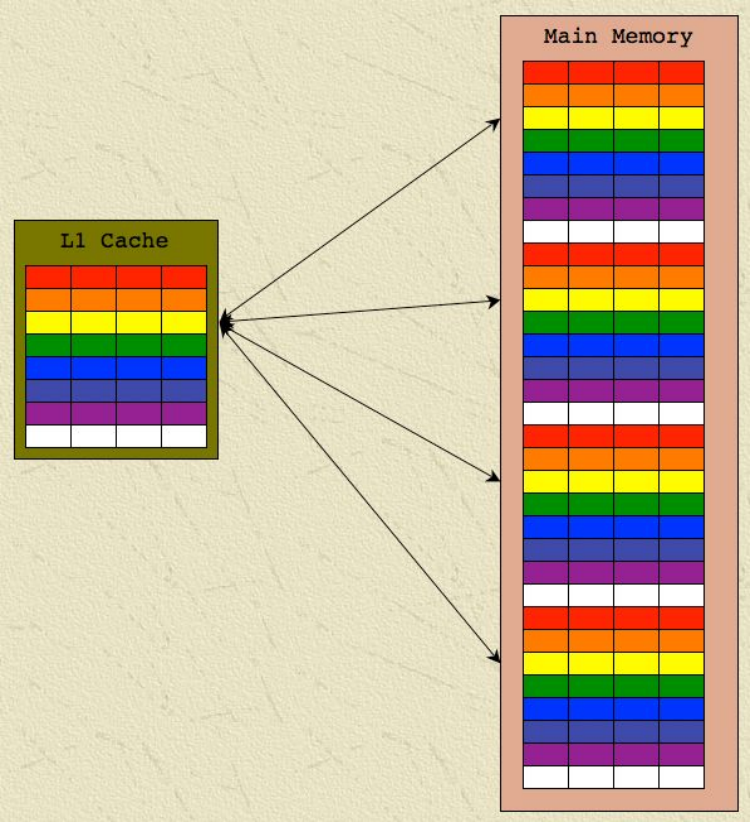
Optimization 1 for Matrix Multiplication: Transpose one input matrix

- Optimization goal: Increase cache hit rate
    - Hardware resources being optimized toward: cache
    - The specification of the hardware we are optimizing for: L1d 32 KB
- Optimization process:
    - Data considerations: To calculate one point value in the result matrix, we need one row value in a input matrix and one column value in another input matrix. Since the cache and memory is row-base. The access of column would make a lot of memory line changing and cache miss, so I transpose the second input matrix at the program begin. Then I can compute each point by two rows in input matrices.

- Optimization results:
    - Performance before optimization: 1081.5 ms (sum of all test cases)
    - Performance after optimization: 971.2 ms (sum of all test cases)

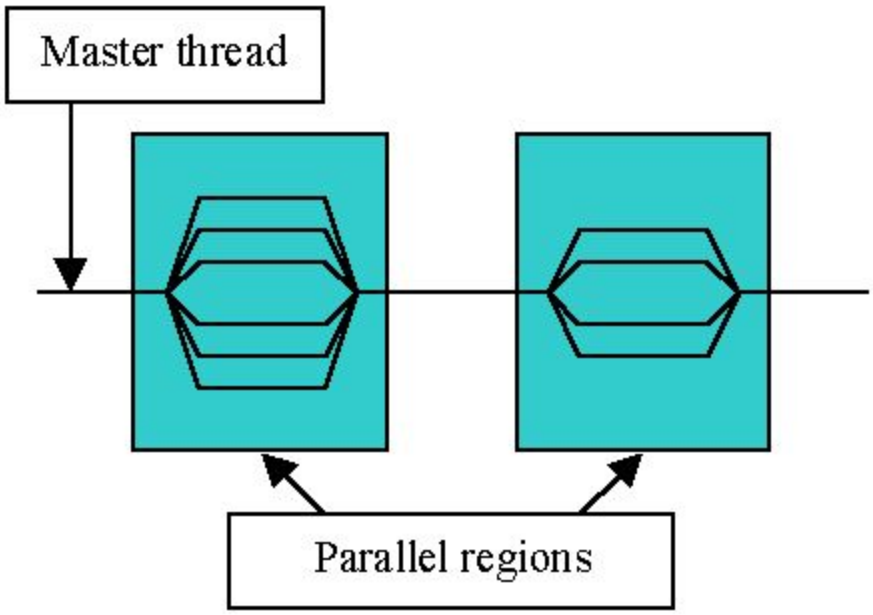


(ref: https://wiki.duke.edu/display/SCSC/Cache+Optimizations)

Optimization 2 for Matrix Multiplication: OMP parallel for

- Optimization goal: Parallelize the calculation of each point in result matrix
    - Hardware resources being optimized toward: Multicore/Multithread
    - The specification of the hardware we are optimizing: 8 threads
- Optimization process:
    - Parallelization considerations: each result point calculation is independent to other points'. So I can simply parallelize the for loop. However, the result matrix has to be shared, the speed up of parallelization would be limited by accumulation of each point in the result matrix. I use a private local variable, set up in OMP with clause "reduction(+:)" to accumulate the value and write back to the result matrix when accumulation is done.

- Optimization results:
    - Performance before optimization: 971.2 ms (sum of all test cases)
    - Performance after optimization: 267.6 ms (sum of all test cases)



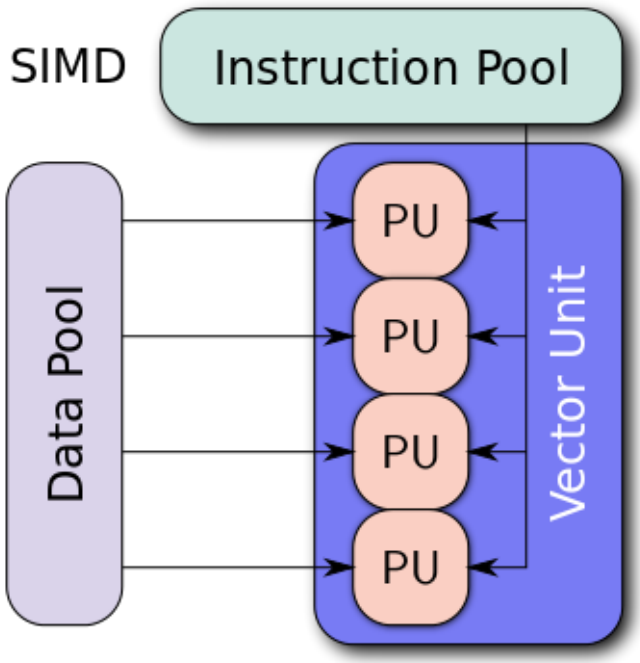
(ref: https://www.dartmouth.edu/~rc/classes/intro_openmp/print_pages.shtml)

Optimization 3 for Matrix Multiplication: SIMD

- Optimization goal: Speed up by utilizing SIMD
    - Hardware resources being optimized toward: SIMD
    - The specification of the hardware we are optimizing for: SSE3

- Optimization process:
    - Data considerations: SSE3 vector needs be fitted to 4 float type data units. In order to make SSE3 work, I expand every number of dimension to multiple of 4. And for expanded rows and columns, I insert zeros as elements to avoid affecting the product.
    - Parallelization considerations: I just load the consecutive 4 points in a row as an input of SSE3, for second input matrix, thanks to transpose, I can also simply load consecutive 4 points in a row as an input. Then use load, add, mul, store these SIMD instructions to implement the matrix multiplication.

- Optimization results:
    - Performance before optimization: 267.6 ms (sum of all test cases)
    - Performance after optimization: 153.2 ms (sum of all test cases)



(ref: https://en.wikipedia.org/wiki/SIMD)

Optimization 4 for K-means Clustering: K-means++ algorithm

- Optimization goal: Speed up by alternative algorithm K-means++
    - Hardware resources being optimized toward: None
    - The specification of the hardware we are optimizing: None
    - It's a better alternative algorithm for selecting initial clusters' centroids.

- Optimization process:
    - For K-means clustering, there are lots of approaches. One of efficient methods to reduce the regression times from update and assignment step is K-means++ algorithm. K-means++ first chooses one data point randomly as the first centroid for the first cluster, then calculates square of distances($D(x)^2$) of each point to the nearest centroid. And deciding the second centroid for the second cluster with probability proportional of each $D(x)^2$. Repeat the process until the final centroid is generated. K-means++ scatters the centroids as possible. It provides O(log k) competitive to the optimal K-means solutions.
    - Although K-means++ is helpful and I also implemented it successfully, the testbench can not allow the change for index number of clusters, and thus cause the test failure. As the result, I didn't apply this algorithm in this project, but I still remained the K-means++ function and function call as a comment in the codes for testbench update in the future.

- Optimization results:
    - Performance before optimization: 125.8 s (sum of all test cases)
    - Performance after optimization: 125.8 s (sum of all test cases)

K-means Algorithm:

1a. Take one center c_1, chosen uniformly at random from $\mathcal{X}$.
1b. Take a new center c_i, choosing $x \in \mathcal{X}$ with probability $\frac{D(x)^2}{\sum_{x \in \mathcal{X}} D(x)^2}$.
1c. Repeat Step 1b. until we have taken k centers altogether.
2-4. Proceed as with the standard `k-means` algorithm.

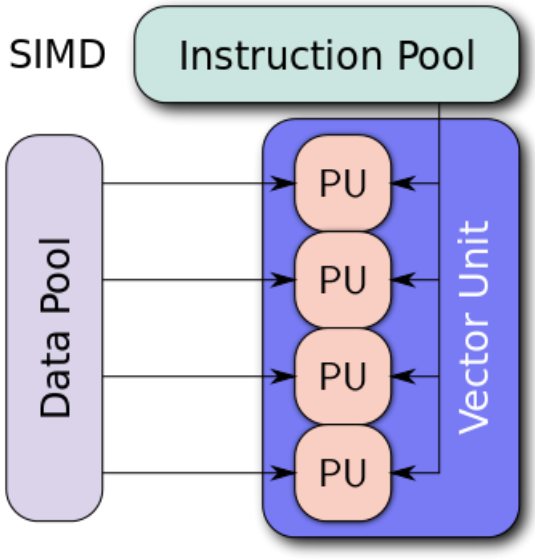
(ref: Arthur, D.; Vassilvitskii, S. (2007). "k-means++: the advantages of careful seeding". Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics Philadelphia, PA, USA. pp. 1027–1035.)

Optimization 5 for K-means Clustering: SIMD

- Optimization goal: Speed up by utilizing SIMD on calculating euclidean distance
    - Hardware resources being optimized toward: SIMD
    - The specification of the hardware we are optimizing: SSE3
    - After analysis of sequential k-means clustering, I found the euclidean distance calculation spends most of the runtime.

- Optimization process:
    - Data considerations: In order to fit 4 data units as SIMD's input at once, I expanded the number of coordinate to multiple of 4. Although there is another way that to check if the last is 4 data units or not, and fill zeros into the range out of number of coordinate, it would dramatically increase the branch selects when calculate the euclidean distance. It's why I choose expanding the number of coordinate. For additional coordinates, fill zeros to avoid affecting the results of distance.
    - Parallelization considerations: Simply put 4 consecutive float data units into SSE3 vector, and use load, sub, mul, add and store instructions to accumulate each 4 coordinates' result at once. Finally, sum them up to get the square of euclidean distance.

- Optimization results:
    - Performance before optimization: 125.8 s (sum of all test cases)
    - Performance after optimization: 16.1 s (sum of all test cases)



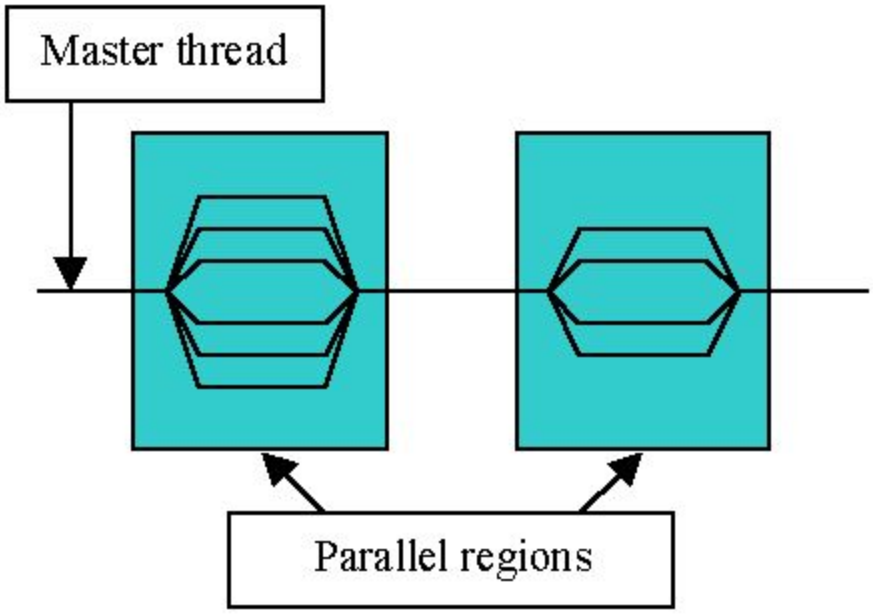
(ref: https://en.wikipedia.org/wiki/SIMD)

Optimization 6 for K-means Clustering: OMP parallel for

- Optimization goal: Utilize OMP to parallel for loops
    - Hardware resources being optimized toward: Multicore/Multithread
    - the specification of the hardware we are optimizing: 8 threads

- Optimization process:
    - Parallelization considerations: There are some for loops in the program can be done in parallel such as picking first data points as initial clusters' centroids and initializing the membership, so I just utilize #pragma omp parallel for to speed up them.

- Optimization results:
    - Performance before optimization: 125.8 s (sum of all test cases)
    - Performance after optimization: 125.3 s (sum of all test cases)
    - Because the calculation amount of picking first data points as initial centers and initializing membership is relatively much smaller than calculating euclidean distance, so the speed up of them is limited.



(ref: https://www.dartmouth.edu/~rc/classes/intro_openmp/print_pages.shtml)

Optimization 7 for K-means Clustering: Arithmetic optimization

- Optimization goal: Utilize Arithmetical to decrease using CPU resources
    - Hardware resources being optimized toward: CPU/Compiler
    - the specification of the hardware we are optimizing: Intel Core i7-4785T

- Optimization process:
    - Data considerations: Because division is an expensive operation. I would like to use other arithmetic operations to replace division. In replacing new clusters' centers, it's divided by cluster's size, I change it into multiply the inverse of cluster's size. Also in while loop condition, originally delta is divided by number of objects and then compare to the threshold, I make delta not being divided and let threshold multiply number of objects. These arithmetical optimization would saving the CPU resources efficiently.

- Optimization results:
    - Performance before optimization: 125.8 s (sum of all test cases)
    - Performance after optimization: 124.7 s (sum of all test cases)
    - Because the calculation amount of replacing new clusters' centers and checking while loop condition is relatively smaller than calculating euclidean distance, so the speed up of arithmetic optimization is limited.