1 Cooperation test

1.1 Introduction

The size of data that must be analyzed keeps increasing year after year and the prize for DRAM are not getting cheaper. NVDIMM offer a lot of storage at a cheaper prize. This opens the opportunity to save money by offloading some of the data to the NVDIMM where the data will be analyzed the same way as the data on the DRAM. The downside to this strategy is that NVDIMM is slower than DRAM so the question is how much data can be offloaded to NVDIMM. If the user offload too much data to NVDIMM then the threads working on analyzing the data on DRAM will be idle while waiting for NVDIMM threads to complete.

The goal of this benchmark is to find a formula that will make it possible to estimate how many threads should be delegated to work on the data located on NVDIMM. The methodology must also be able to calculate how much of the data on DRAM should be offloaded to NVDIMM

There is two version of the code that is being used. They have been programmed slightly different in the way read data. This is to explore which of them is the fastest.

1.2 Formula

In order to use the formula one must measure the NVDIMM and DRAM speed using a benchmark made in a previous chapter where speed was measured when data was transferred from DRAM-DRAM and NVDIMM-NVDIMM simultaneously with different amount threads allocated to the different processes. By using using the speed from the benchmark in the formula below along with the size of the data that will be used in the calculation the user can easily calculate how much data can be transferred to NVDIMM without losing time. The formula only decide how much data can be allocated to NVDIMM with a certain amount of threads. This means that the user must probably use the formula several times where the number of NVDIMM threads varies from one to five in order to find the best combination of threads and data allocated to the NVDIMM process.

Formula

$$\frac{Total\_data - nvdimm\_data}{dram\_speed} = \frac{nvdimm\_data}{nvdimm\_speed}$$

$$nvdimm\_data = \frac{nvdimm\_speed * Total\_data}{nvdimm\_speed + dram\_speed}$$

Calculation
I have created a benchmark that will test this formula to see if it is accurate. This benchmark has an two dimensional array filled with data. The benchmark start at element (1,1) of the array where it sum ups all of its eight neighbors and then takes the average. The result is stored in the same position in another two dimensional array. The benchmark does this for every element between (1,1) and (m-2,n-2). The benchmark repeats this process ten times and after each time the benchmark will swap both the DRAM arrays and NVDIMM array. The time is measured at the beginning of the process and at the end, this time is called total_time in the code. Each thread will also measure the time they takes to complete their own tasks, in the code this is called individual_time.

In listing 1 is an example of the serial code of the benchmark. The array has not been divided into two. The thread will will repeat the while-loop on line 1 k_length amount of time. It will start the time measurement at line 2. From line 3-10 the thread will pass through the the 2D-array row by row starting from element (1,1) and end at element (m-2,n-2). At each element the thread will calculate the average of all the elements neighbors, eight neighbors in total. The thread will then stop the time measurement at line 11 and swap the arrays at line 12-14. It will also increase k by one and start over if k is still lower than k_length.

Listing 1: Serial code of the test

```
1  while(k<K_length){
2    total_time[k] = mysecond();
3    for( i=1; i<m-2; i++){
4      for( j=1; j<n-2; j++){
5        temp = A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+
6                A[i][j-1]    +      A[i][j+1]+
7                A[i+1][j-1] + A[i+1][j] + A[i+1][j+1];
8        B[i][j] = temp/8;
9      }
10   }
11   total_time[k] = mysecond() - total_time[k];
12   temp_array = A;
```

```
13      A = B;
14      B = temp_array;
15      k++;
16  }
```

1.3 Prediction

Threads	DRAM speed	DRAM Size	DRAM pred.time	NVDIMM speed	NVDIMM Size	NVDIMM pred.time
1	61042.84	15620.75	0.2559	3038.16	379.48	0.1249
2	57761.80	15244.31	0.2639	6027.24	755.92	0.1254
3	53767.10	14828.91	0.2758	9223.43	1171.32	0.1270
4	51257.79	14418.57	0.2813	12630.42	1581.66	0.1252
5	47770.89	14068.95	0.2945	15202.48	1931.28	0.1270

Table 1: Prediction

Threads	DRAM speed	DRAM Size	DRAM pred.time	NVDIMM speed	NVDIMM Size	NVDIMM pred.time
1	61042.84	156206.40	3.8384	3038.16	3793.60	1.8730
2	57761.80	152441.60	3.9587	6027.24	7558.40	1.8811
3	53767.10	148286.40	4.1369	9223.43	11713.60	1.9050
4	51257.79	144184.00	4.2194	12630.42	15816.00	1.8783
5	47770.89	140686.40	4.4175	15202.48	19313.60	1.9056

Table 2: Prediction

1.4 First version

There are two groups of threads that works in parallel in this program. The first group of threads works on the part of the data that is stored on DRAM and the other works on the data stored on NVDIMM. One thread in each group works on data that borders with the other group. In the DRAM group that is the thread with the highest thread_id. Each of the elements in the last row of data will have three neighbors that exist on the NVDIMM side. This means that the thread must access the NVDIMM in order to get the data. The NVDIMM thread with the lowest thread_id also have elements in the first row of data that have

three neighbours that exist in DRAM that must be accessed by the thread directly.

Explaination of code:
The code below only shows the calculation process, it does not the rest of the code. Allocation of memory have been done by all the threads, as a result the data have been spread across all the memory channels. The data is a 2d array where the rows of data on DRAM will be divided equally between the DRAM threads, the rows of data on NVDIMM will also be divided equally between the NVDIMM threads. The variable slice_start hold the index of the row where the tread must start at and slice_end holds the index of the row the thread must stop at. Array A and B are DRAM array and array C and D are NVDIMM arrays. The average found by adding together eight neighbors in A will be placed in same position in B. The same is true for C and D.

The process are repeated K_length amount of time, usually ten times in my tests. One way of knowing if the test result are correct is to run the test several time and look for consistency. The code measures the time taken to complete one iteration of calculation, this is done in the beginning of the code at line 5 and at the end at line 84 by a single thread. All the threads then get divided into the DRAM and NVDIMM at line 8. If the thread_id of a thread is less than the dram_threads is it will do calculation on the data in DRAM, the rest will fail the if test and move on to the else bracket at line 42. Dram_threads is the total amount of threads that will be working on data in DRAM.

At line 11 the thread with the highest thread_id will pass the if test and the rest will move on to line 30. The thread with highest thread_id will then measure time at line 12 and end the measurement in line 29, this is the start and the end of the bracket. The thread will then enter a double for-loop at line 13-20 that will go through elements from position (slice_start,1) until (slice_end-1,n-1), this leaves out the last row assigned to the tread, that row will be dealt with later. At each element the for-loop it will add all of its eight neighbors together at line 15-17 and divide by eight at line 18. The thread will then enter a new for-loop at line 23, this for-loop will calculate average of the last row on DRAM. Elements of this row have three neighbors that exist in NVDIMM. The thread will access the NVDIMM directly when adding the eight neighbors at line 24-26. Data on NVDIMM are accessed by the thread at line 26, the thread is using a library developed for this purpose.

For all the other DRAM threads that jumped to line 30 will start

by taking time measurement at the beginning and at the end of the bracket at line 31 and 40. The code from line 32-39 is identical to line 13-20 describe before.

The group of NVDIMM enters the else bracket at line 42 where the thread with the lowest thread_id will pass the if-sentence at line 44, the rest will move on to the else bracket at line 62. The thread will then measure time at line 45 and end the measurement in line 61. It will then enter a for-loop at line 47 and will begin calculating the average of the neighbors of the elements in the first row. The first row have three of its eight neighbors in the row above and they exist in the DRAM. Once done the thread will move on to a new for-loop at line 53. This for-loop will go through the rest of the portion of data the thread have been given and calculate the average of each elements neighbors.

The rest of the NVDIMM threads will move into the else bracket at line 62. The code here is very similar to the code at 31-40 that has been described at a previous paragraph. The only difference is that the code at line 66-68 where the code reads from NVDIMM instead DRAM.

All the threads will wait a barrier at line 75 until all threads are done. After that on thread will enter a single bracket where array A and B will swap places, array C and D will also swap places. The time it took for this one iteration will be registered at line 84. After this the code will move back to line 1.

Listing 2: First version

```
1  while(k<K_length){
2    #pragma omp barrier
3    #pragma omp single
4    {
5      total_time[k] = mysecond();
6    }
7    //Divides threads into DRAM threads and NVDIMM threads.
8    if( thread_id < dram_threads ){
9
10     //for the thread bordering on NVDIMM thread.
11     if( thread_id==(dram_threads-1) ){
12       individual_time[k][thread_id] = mysecond();
13       for( i=slice_start; i<slice_end-1; i++){
14         for( j=1; j<nMinusOne; j++){
15           temp = A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+
16                A[i][j-1]    +      A[i][j+1]+
17                 A[i+1][j-1] + A[i+1][j] + A[i+1][j+1];
18           B[i][j] = temp*inverseEigth;
```

```
19            }
20          }
21
22          i = slice_end-1;
23          for( j=1; j<nMinusOne; j++){
24            temp = A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+
25                 A[i][j-1]    +      A[i][j+1]+
26                 D_RO(C)[i*n+j] + D_RO(C)[i*n+j] +
                      D_RO(C)[i*n+j];
27            B[i][j] = temp*inverseEigth;
28            }
29          individual_time[k][thread_id] = mysecond() -
               individual_time[k][thread_id];
30        }else{
31          individual_time[k][thread_id] = mysecond();
32          for( i=slice_start; i<slice_end; i++){
33            for( j=1; j<nMinusOne; j++){
34              temp = A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+
35                   A[i][j-1]    +      A[i][j+1]+
36                   A[i+1][j-1] + A[i+1][j] + A[i+1][j+1];
37              B[i][j] = temp*inverseEigth;
38            }
39          }
40          individual_time[k][thread_id] = mysecond() -
               individual_time[k][thread_id];
41        }
42      }else{
43        //for the thread bordering on DRAM thread.
44        if( thread_id==dram_threads ){
45          individual_time[k][thread_id] = mysecond();
46          i=0;
47          for( j=1; j<nMinusOne; j++){
48            temp =
                 A[dram_part-1][j-1]+A[dram_part-1][j]+A[dram_part-1][j+1]+
49                D_RO(C)[i*n+(j-1)]     +
                     D_RO(C)[i*n+(j+1)]+
50                D_RO(C)[(i+1)*n+(j-1)] + D_RO(C)[(i+1)*n+j]
                     + D_RO(C)[(i+1)*n+(j+1)];
51            D_RW(D)[i*n+j] = temp*inverseEigth;
52          }
53          for( i=slice_start+1; i<slice_end-1; i++){
54            for( j=1; j<nMinusOne; j++){
55              temp = D_RO(C)[(i-1)*n+(j-1)] +
```

```
                D_RO(C)[(i-1)*n+j] + D_RO(C)[(i-1)*n+(j+1)]+
56                D_RO(C)[i*n+(j-1)]     +
                     D_RO(C)[i*n+(j+1)]+
57                D_RO(C)[(i+1)*n+(j+1)] +
                     D_RO(C)[(i+1)*n+j] +
                     D_RO(C)[(i+1)*n+(j+1)];
58            D_RW(D)[i*n+j] = temp*inverseEigth;
59          }
60        }
61        individual_time[k][thread_id] = mysecond() -
             individual_time[k][thread_id];
62      }else{
63        individual_time[k][thread_id] = mysecond();
64        for( i=slice_start; i<slice_end; i++){
65          for( j=1; j<nMinusOne; j++){
66            temp = D_RO(C)[(i-1)*n+(j-1)] +
                D_RO(C)[(i-1)*n+j] + D_RO(C)[(i-1)*n+(j+1)]+
67                D_RO(C)[i*n+(j-1)]     +
                     D_RO(C)[i*n+(j+1)]+
68                D_RO(C)[(i+1)*n+(j-1)] +
                     D_RO(C)[(i+1)*n+j] +
                     D_RO(C)[(i+1)*n+(j+1)];
69              D_RW(D)[i*n+j] = temp*inverseEigth;
70          }
71        }
72        individual_time[k][thread_id] = mysecond() -
             individual_time[k][thread_id];
73      }
74    }
75    #pragma omp barrier
76    #pragma omp single
77    {
78      tempArray = B;
79       B=A;
80       A=tempArray;
81      temp_nvdimm = C;
82      C = D;
83      D = temp_nvdimm;
84      total_time[k] = mysecond() - total_time[k];
85      k++;
86    }
87    #pragma omp barrier
88  }//End of while
```

Table 1 shows the calculation of how much data must be allocated to NVDIMM in order for the DRAM and NVDIMM to complete their tasks simultaneously. M in row one in table shows how many rows the 2D-array has. The n in row two shows how many elements each row has. The total MB in row three is calculated in following manner, m*n/(8*1000000). Row five is the beginning of five column. First column shows how many threads are used to calculate data on NVDIMM. Second and third column is the DRAM and NVDIMM speed in MB per second. The speeds comes from a benchmark described in a previous chapter where data is transferred from DRAM-DRAM and NVDIMM-NVDIMM simultaneously. Column four uses the formula described in the beginning of the chapter, the numbers are in MB. The last column converts the result in fourth column into number of rows of the 2D-array that will be placed on NVDIMM.

m	100000			
n	100000			
total MB	80,000			
speed				
Nvm-thr	dram	nvm	nvmpart	rows
1	63003.00	3036.04	3677.88	2299
2	60302.95	6032.18	7274.80	4547
3	58967.56	8404.24	9979.54	6237
4	54617.41	10551.97	12953.28	8096
5	55545.81	13330.65	15483.55	9677
6	52102.34	18159.78	20676.61	12923

Table 3: First version, distribution

Each row in table two is a result of one test. In table two the first two column shows m and n. Third column shows how many rows are assigned to NVDIMM. Fourth and fifth column shows the number of DRAM and NVDIMM threads the test will have. Column six shows the average speed of all the DRAM threads in the test. The test is repeated ten times so if eleven DRAM threads then there are 99 DRAM threads that will be taken average of. The first test is excluded because the times are way higher then all the tests that comes after. The two next column is the dram minimum and dram maximum. Dram minimum is found by first finding the fastest DRAM thread in each of the nine tests and then take the average of them. Dram maximum is calculated the same way as dram minimum, The only difference is that this is for the slowest speed. Column nine, ten and eleven shows nvdimm average, nvdimm minimum and nvdimm maximum. These column are

similar to dram average, dram minimum and dram maximum. The different is that the times is for the nvdimm threads. The last three column is the total average, total minimum and total maximum. The total time is only measured once for each test so total minimum and total maximum shows the fastest and slowest test. Total average is the average of all the tests except the first test.

M	N	NVDIMM rows	DRAM threads	NVDIMM threads	DRAM time	NVDIMM time	Total time
31623	31623	750	15	1	0.2723	0.3468	0.3479
31623	31623	1494	14	2	0.2711	0.3463	0.3492
31623	31623	2315	13	3	0.3091	0.4189	0.5873
31623	31623	3126	12	4	0.2826	0.3599	0.3612
31623	31623	3817	11	5	0.2912	0.3552	0.3564

Table 4: First version, result

M	N	NVDIMM rows	DRAM threads	NVDIMM threads	DRAM time	NVDIMM time	Total time
100000	100000	2371	15	1	3.7423	4.4166	6.1124
100000	100000	4724	14	2	4.0061	5.0799	7.1181
100000	100000	7321	13	3	4.0294	7.3525	7.8412
100000	100000	9885	12	4	4.0449	6.2484	7.3231
100000	100000	12071	11	5	4.0556	7.7680	7.7681

Table 5: First version, result

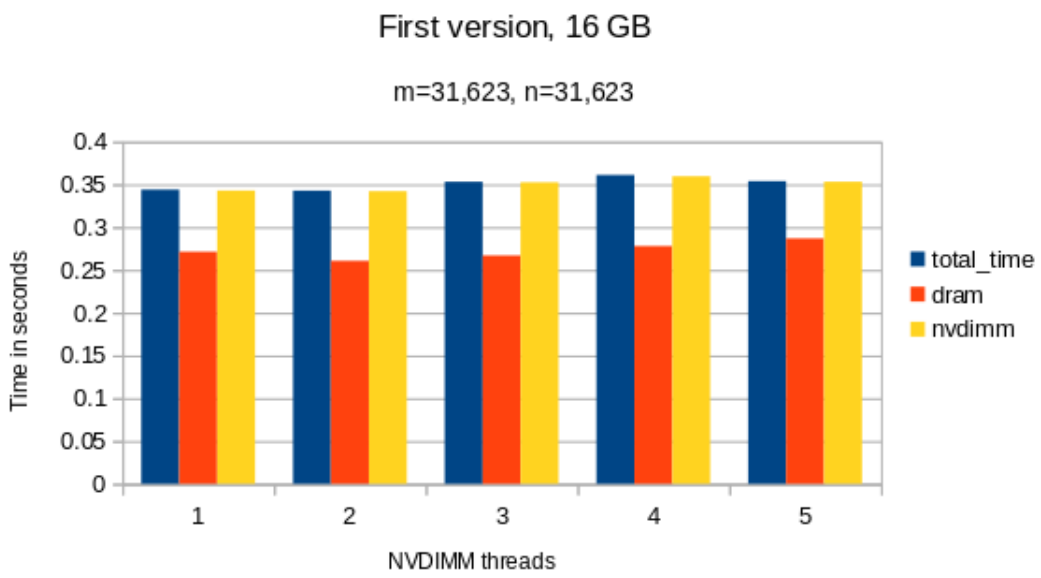

Figure 1: First version, result

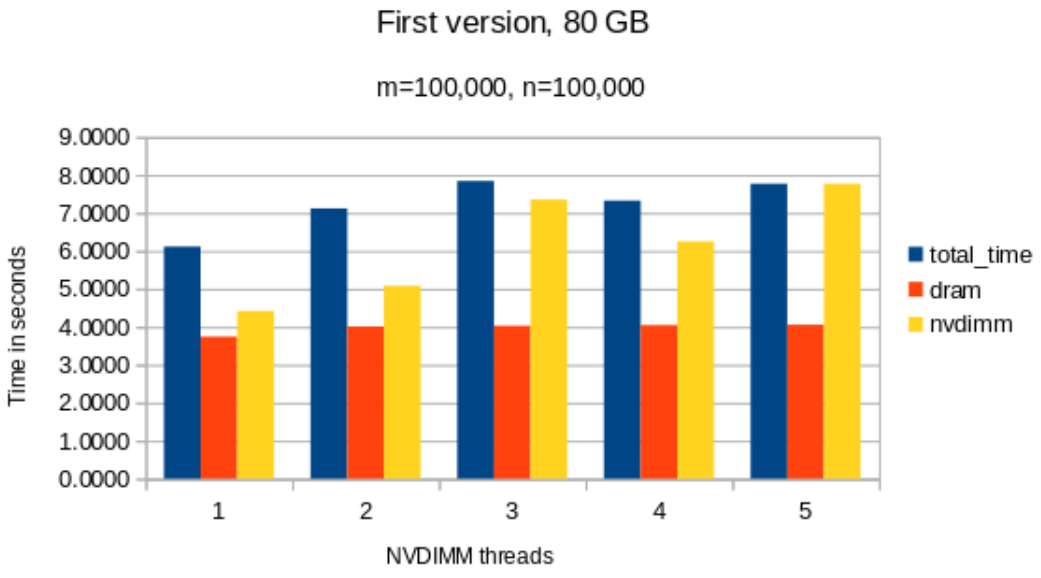

Figure 2: First version, result

The formula have been tested when the total array size are 8 GB, 16 GB, 32 GB, 64 GB and 80 GB. When the workload are 8 GB and 16 GB the workload between DRAM and NVDIMM is almost perfectly balanced. When the workload is increased to 32 GB the NVDIMM time starts to become slower than DRAM. The test with six NVDIMM threads have become a lot slower. When the workload is increased to 64 GB and 80 GB the NVDIMM time gets even slower compared to the DRAM time.

1.5 Second version

Same as the first version there are two groups of threads that works in parallel in this program. The first group of threads works on the part of the data that is stored on DRAM and the other works on the data stored on NVDIMM. In this version the two threads that has a row of elements with neighbours in the other type of memory will not directly access this data. Instead the two arrays will have their own ghost array on their memory that they will access instead of fetching data from the other side.

The while loop in line 1 will repeat itself for K_length amount of time usually ten times, this is decided by the user of the benchmark. All the threads will wait for all the other threads to arrive at line 2 before one of the threads will start the time measurement in line 5. When the threads arrives at line 8 the all the DRAM threads will pass the if-test and all the NVDIMM threads will move on to line 19. The DRAM threads will first start the time measurement in line 9 and will then start to calculate the average for every elements in the part of the array that have been allocated to the each of the threads, this will happen from line 10-17. The threads will then stop the time measurement in line 18.

The NVDIMM threads will enter the else bracket at line 19. The threads will then start time measurement at line 20. From line 21-28 the threads will calculate average on the elements that have been assigned to each thread. At line 29 the threads will stop the time measurements.

The DRAM and NVDIMM threads will then encounter a barrier at line 31 where they will wait for all the other threads to finish. All threads will then update the ghost arrays at line 34 and 35. At line 34 the threads are copying the second row in the NVDIMM array into the last row in the DRAM array. In line 35 they are copying the second last row in the DRAM array into the first row in the NVDIMM array.

Once all the threads are done copying their parts of the rows, one thread will enter the single at line 37. This thread will swap the DRAM arrays and then the NVDIMM array. The thread will then stop the time measurement that was at the beginning of the while-loop at line 47. At line 48 the k variable gets increased by one. The threads will then return to line 1.

Listing 3: Second version

```
while(k<K_length){
  #pragma omp barrier
  #pragma omp single
  {
    total_time[k] = mysecond();
  }
  //Divides threads into DRAM threads and NVDIMM threads.
  if( thread_id < dram_threads ){
    individual_time[k][thread_id] = mysecond();
    for( i=slice_start; i<slice_end; i++){
      for( j=1; j<nMinusOne; j++){
        temp = A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+
             A[i][j-1]   +     A[i][j+1]+
             A[i+1][j-1] + A[i+1][j] + A[i+1][j+1];
        B[i][j] = temp*inverseEigth;
      }
    }
    individual_time[k][thread_id] = mysecond() -
       individual_time[k][thread_id];
  }else{
  individual_time[k][thread_id] = mysecond();
    for( i=slice_start; i<slice_end; i++){
      for( j=1; j<nMinusOne; j++){
        temp = D_RO(C)[(i-1)*n+(j-1)] +
           D_RO(C)[(i-1)*n+j] + D_RO(C)[(i-1)*n+(j+1)]+
            D_RO(C)[i*n+(j-1)]     +
               D_RO(C)[i*n+(j+1)]+
            D_RO(C)[(i+1)*n+(j-1)] + D_RO(C)[(i+1)*n+j]
               + D_RO(C)[(i+1)*n+(j+1)];
        D_RW(D)[i*n+j] = temp*inverseEigth;
      }
     }
    individual_time[k][thread_id] = mysecond() -
       individual_time[k][thread_id];
  }
  #pragma omp barrier
  #pragma omp for
  for(a=0; a<n; a++){
    A[dram_part_Minus_One][a] = D_RO(C)[n+a];
    D_RW(C)[a] = A[dram_part_Minus_Two][a];
  }
  #pragma omp single
  {
```

```
      tempArray = B;
      B=A;
      A=tempArray;

      temp_nvdimm = C;
      C = D;
      D = temp_nvdimm;

      total_time[k] = mysecond() - total_time[k];
      k++;
    }
    #pragma omp barrier
}//End of while
```

M	N	NVDIMM rows	DRAM threads	NVDIMM threads	DRAM time	NVDIMM time	Total time
31623	31623	750	15	1	0.2716	0.3145	0.3181
31623	31623	1494	14	2	0.2739	0.3276	0.3281
31623	31623	2315	13	3	0.2792	0.3292	0.3331
31623	31623	3126	12	4	0.2824	0.3359	0.3466
31623	31623	3817	11	5	0.3365	0.5129	0.5844

Table 6: second version, result

M	N	NVDIMM rows	DRAM threads	NVDIMM threads	DRAM time	NVDIMM time	Total time
100000	100000	2371	15	1	4.1319	4.8735	4.9889
100000	100000	4724	14	2	4.0864	5.0859	6.0841
100000	100000	7321	13	3	4.0079	5.5318	6.6954
100000	100000	9885	12	4	4.1450	5.7829	6.5336
100000	100000	12071	11	5	4.3569	6.0666	6.7233

Table 7: second version, result

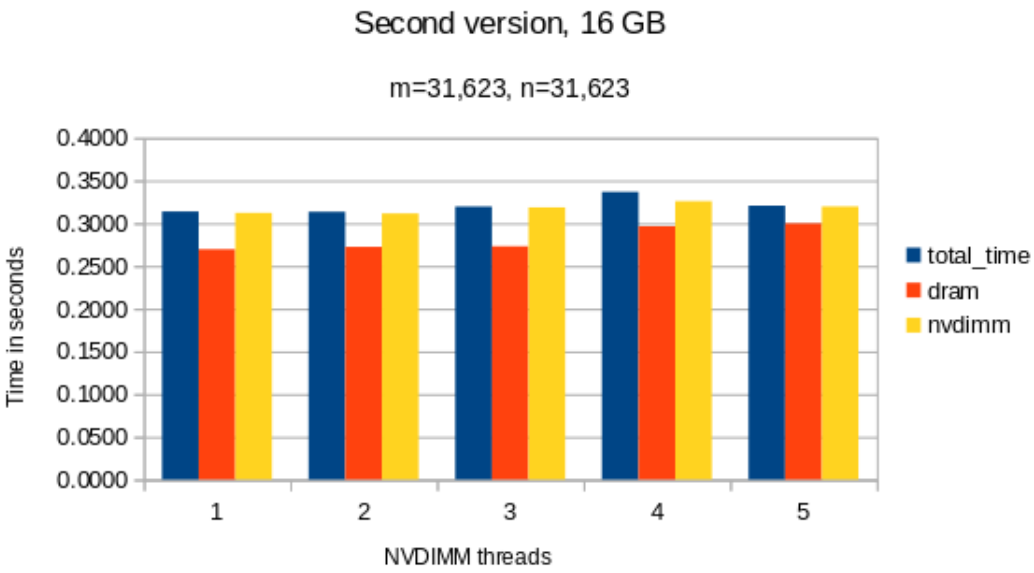

Figure 3: second version, result

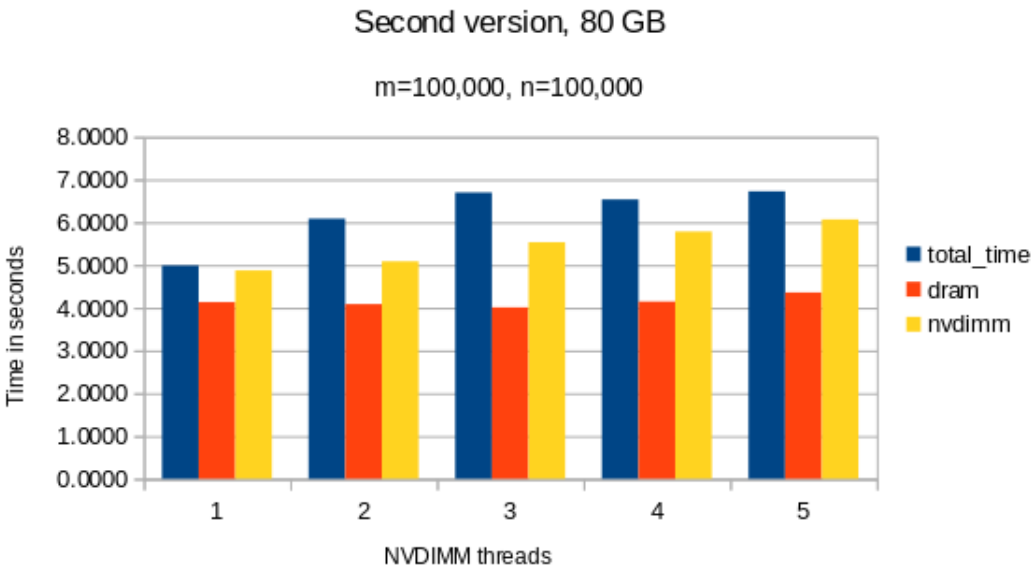

Figure 4: second version, result

The formula have been tested when the total array size are 8 GB, 16 GB, 32 GB, 64 GB and 80 GB. When the array size is 8 GB, 16 GB and 32 GB the workload between DRAM and NVDIMM is almost perfectly balanced. When the array size is increased to 64 GB and 80 GB the NVDIMM becomes slower and slower.

1.5.1 Comparisons

This