

1 introduction

In science computer simulation has become an important tool. Simulation are becoming more and more advanced which increase the amount of data that are being generated. This data gets stored on harddrive and loaded again when its time to analyze the data. By doing in-situ real-time analysis where the data gets analyzed immediately after being generated. By doing computer simulation this way it may be possible to save time and hardware resources.

2 Benchmarks

2.1 Hardware

The program have been tested on a server with the following hardware.

Motherboard: Supermicro X11DPU-Z+

CPU: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 32 core

DRAM: Samsung RDIMM, 2666 MT/s.

NVDIMM: Micron Technology NV-DIMM , 2933 MT/s

Both CPU have twelve memory slots each. Each CPU have six channels. There are one DRAM and one NVDIMM sharing one channel.

2.2 STREAM DRAM

The STREAM[**STREAM-c**] benchmark is a synthetic and simple benchmark that is designed to measure bandwidth in MB/s. This benchmark is seen as the standard for measuring memory bandwidth and has not been modified in any way after it was downloaded from the creators websites. The benchmark test memory bandwidth by running four different tests. The first one test is copy where the elements in one array is copied to another array. The second test is called scale where each element are multiplied with a constant and the result is placed in a second array, the index of the element in the first array and the result in the second array is the same. Third test is add where the elements from two different arrays with the same index are added together and place in a third array where the index is the same as in the two other arrays. Last test is the triad where the one array is multiplied with a constant then added together with a second array and then placed in a third array.

The benchmark run the test 32 times and only on one socket, every times it restart with one extra thread is added. The CPU has 16 cores and when the thread number surpass that number it starts using the hyper thread on the same core. The Linux program numactl is also used to manage the number of threads and what socket the benchmark is allowed to used. The result is as one would expect, adding more threads in beginning will give a big increase in transfer speed. But at thread 5 there gains in transfer speed will start to diminish and at thread 11 there will be very little increase in transfer speed when adding more threads.

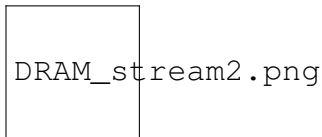


Figure 1: DRAM Stream

2.3 STREAM NVDIMM

The stream NVDIMM benchmark measure the memory speed of the NVDIMM. This benchmark is the same as the STREAM DRAM benchmark mention above. The different is that the memory type have been changed from DRAM to NVDIMM. The original code looks like this.

Listing 1: Description

```

1 #ifndef STREAM_TYPE
2 #define STREAM_TYPE double
3 #endif
4
5 static STREAM_TYPE a[STREAM_ARRAY_SIZE+OFFSET],
6                    b[STREAM_ARRAY_SIZE+OFFSET],
7                    c[STREAM_ARRAY_SIZE+OFFSET];

```

It have been changed into this. In addition the PMEMObjpool must be initiated in main method.

```

1 PMEMObjpool *pop;
2 POBJ_LAYOUT_BEGIN(array);
3 POBJ_LAYOUT_TOID(array, double);
4 POBJ_LAYOUT_END(array);

```

```

5  TOID(double) a;
6  TOID(double) b;
7  TOID(double) c;
8
9  int main()
10 {
11     const char path[] = "/mnt/pmem0-xfs/pool.obj";
12     pop = pmemobj_create(path, LAYOUT_NAME, 10737418240,
13                          0666);
14     if (pop == NULL)
15         pop = pmemobj_open(path, LAYOUT_NAME);
16
17     if (pop == NULL) {
18         perror(path);
19         exit(1);
20     }

```

NVDIMM_stream2.png

2.4 benchmark 3

Graphs and tables below show the speed of a certain amount of NVDIMM threads while the rest of the threads are from DRAM to DRAM. The test have been conducted by transfer data simultaneously from DRAM-DRAM and NVM-NVM. All the threads are transferring the values of one array to another, all the arrays have 100 million elements of type double. This transfer happens 5000 times and the graphs shows the average of the first 200 iterations. This is done to ensure that all the threads can't finish early and make the remaining threads faster. The sum graphs shows the sum bandwidth of DRAM and NVM. Average graphs shows the average bandwidth of DRAM and NVM.

2.4.1 The code

The benchmark have three different program, the code is mostly the same except for the part where NVDIMM threads from NVDIMM-NVDIMM, DRAM-NVDIMM or NVDIMM-DRAM.

The entire code is run in the main function except when it finds the current time, that is done in a different function. The code begins with creating a memory pool and reads the parameters from command line. The parameters are how many threads are using the NVDIMM, the total amount of threads used and how many time the test will repeat itself. The code will then enter parallel area where one thread will create a 2d array where all the threads will save the time it took to copy the array.

Every threads will then create their two arrays. The thread id will determine if both arrays will be DRAM array, or if one or two arrays will be NVDIMM array. Both arrays will be DRAM if the thread id is lower than the total amount of threads minus the number of NVDIMM threads. Thread ids equal or higher than that will either have one or both arrays stored in NVDIMM. All the arrays in all of the threads will be populated by random numbers. When a thread is done it will wait at a barrier until all the other threads are populating their threads.

Listing 2: Creation of DRAM and NVDIMM arrays

```

1  if(thread_id < totalThreads-nvmThreads){
2      //From DRAM to DRAM
3      drm_read_array =
4          (double*)malloc(ARRAY_LENGTH*sizeof(double));
5      drm_write_array =
6          (double*)malloc(ARRAY_LENGTH*sizeof(double));
7      #pragma omp critical
8      {
9          for(i=0;i<ARRAY_LENGTH;i++){
10             drm_read_array[i] =
11                 ((double)rand()/(double)(RAND_MAX));
12             drm_write_array[i] =
13                 ((double)rand()/(double)(RAND_MAX));
14         }
15     }
16 }
17 else if(thread_id >= totalThreads-nvmThreads){
18     //From NVDIMM to NVDIMM
19     POBJ_ALLOC(pop, &nvm_read_array, double, sizeof(double)
20         * ARRAY_LENGTH, NULL, NULL);
21     POBJ_ALLOC(pop, &nvm_write_array, double, sizeof(double)
22         * ARRAY_LENGTH, NULL, NULL);
23     #pragma omp critical
24     {

```

```

19     for(i=0; i<ARRAY_LENGTH; i++) {
20         D_RW(nvm_read_array)[i] =
            ((double) rand() / (double) (RAND_MAX));
21         D_RW(nvm_write_array)[i] =
            ((double) rand() / (double) (RAND_MAX));
22     }
23 }
24 }

```

Threads with DRAM arrays and threads with one or two NVDIMM array will split into their own part of the code with an if-sentence. All the threads will run the as many times as specified in the parameters and save the time each test takes in the 2d array that was created in the beginning. When they are done they will free up the memory and leave the parallel area.

Listing 3: Threads running their test.

```

1  if(thread_id < totalThreads-nvmThreads){
2      //From DRAM to DRAM
3      for(i=0; i<total_tests; i++){
4          //Time start
5          test_time[thread_id][i] = mysecond();
6          for(j=0; j<ARRAY_LENGTH; j++){
7              drm_write_array[j] = drm_read_array[j];
8          }
9          //Time stop.
10         test_time[thread_id][i] = mysecond() -
            test_time[thread_id][i];
11     }
12 }
13 else if(thread_id >= totalThreads-nvmThreads){
14     //From NVDIMM to NVDIMM
15     for(i=0; i<total_tests; i++){
16         //Time start
17         test_time[thread_id][i] = mysecond2();
18         for(j=0; j<ARRAY_LENGTH; j++){
19             D_RW(nvm_write_array)[j] = D_RO(nvm_read_array)[j];
20
21         //Time stop.
22         test_time[thread_id][i] = mysecond2() -
            test_time[thread_id][i];
23     }
24 }

```

The code will then print the entire 2d array where the time measurements are stored to the terminal. Each line represent all the test done by one thread. In the beginning of each line the code will add either DRAM if both arrays are stored on DRAM. Or NVM if one or both arrays are stored on NVDIMM. When the program is done printing it will exit.

2.4.2 NVM-NVM

The tables below shows the result of the benchmark where one group of threads transfer from DRAM-DRAM and another group from NVDIMM-NVDIMM. The test result in the tables are the transfer speed in MB/s. The first table shows the combined transfer speed of all threads that are copying from DRAM-DRAM and the second table shows the combined speed of all the threads that are copying from NVDIMM-NVDIMM.

The first line in the first table in figure 2 shows the combined transfer speed of the DRAM-DRAM copying where there are one thread copying from NVDIMM-NVDIMM. That means there are 15 threads that are copying from DRAM-DRAM. The first line in the second table shows the transfer speed of that one thread. The columns shows the the test numbers. The column with name the "1-20" shows the average transfer speed of the first 20 tests. The third table in figure 3 shows the combined transfer speed of all the 16 threads in the first two tables. There is also a graph where all three tables are being represented.

One of the hopes by using NVDIMM and DRAM simultaneously was that there would be an increase in the transfer speed. But by comparing copy on figure 1 with the sum on figure 4 one can see that there has been no increase in transfer speed. Both graphs shows a transfer speed on around 65000 MB/s.

Figure 2: NVM-NVM 1-100 iteration, 16 threads total, 3rd version

Figure 3: NVM-NVM 1-100 iteration, 16 threads total, 3rd version

Figure 4: NVM-NVM graph 1-20, 3rd version

2.4.3 NVM-DRAM

This benchmark is similar to the previous benchmark. The only difference is that some threads will transfer data from NVDIMM-DRAM instead of NVDIMM-NVDIMM.

Figure 5: NVM-DRAM 1-100 iteration, 3rd version

Figure 6: NVM-DRAM 1-100 iteration, 3rd version

Figure 7: NVM-DRAM graph 1-20, 3rd version

2.4.4 DRAM-NVM

This benchmark is also similar to the other two benchmarks. This time some of the threads will transfer from DRAM-NVDIMM.

Figure 8: DRAM-NVM 1-100 iteration, 3rd version

Figure 9: DRAM-NVM 1-100 iteration, 3rd version

Figure 10: DRAM-NVM graph 1-20, 3rd version

3 Simulation

3.1 DRAM only

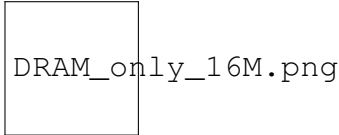


Table 1: Simulation with only DRAM, n=16M

The generation of data happens inside a while-loop that will repeat the generation and analyzing of data 5000 times. The total time is measured by taking the time before and after the while-loop. The analyzing time is measured the same way by taking the time before and after the analyzing part of the program. Data generation time is measured by subtracting the analyze time from the total time. All the time measurements are made inside a pragma omp single so there are no reason to be worried if the mysecond-method is thread-safe. The mysecond-method have been copied from the original stream benchmark. All the arrays used in the code have been subjected to first touch before the time is measured.

Listing 4: while-loop

```
1 #pragma omp single
2 {
3     data_generation_time, = mysecond();
4 }
5 while( n<5000 ){
6     #pragma omp barrier
7     #pragma omp single
8     {
9         n++;
10        diff=0.0;
11        average = 0;
12        //completes the first part of the formula.
13        Wk_1_product = (omd + (d*Wk_1))*iN;
14    }
15
16    Data generation, see chapter 5.1.1
17
18    #pragma omp barrier
```

```

19  #pragma omp single
20  {
21      temp_x = xk_1;
22      xk_1 = x;
23      x = temp_x;
24      //starting time measurement of calculation.
25      temp_calc=mysecond();
26  }
27
28  Analyzing the data, see chapter 5.1.2
29
30  #pragma omp barrier
31  #pragma omp single
32  {
33      average *= iN;
34      analyze_time+=mysecond()-temp_calc;
35  }
36  }
37  #pragma omp single
38  {
39      data_generation_time, = mysecond() -
        data_generation_time;
40  }

```

Listing 5: while-loop

```

1  double mysecond() {
2      struct timeval tp;
3      struct timezone tzp;
4      int i;
5      i = gettimeofday(&tp,&tzp);
6      return ( (double) tp.tv_sec + (double) tp.tv_usec *
        1.e-6 );
7  }

```

3.1.1 Data generation

The arrays `x` and `xk_1` are double arrays that have the length of 4'000'000 elements. `CRS_row_ptr` and `CRS_col_idx` are int array that have the length of 31'976'004 elements, `CRS_values` have the same length and a double array. The code run through the `x`-array one time and `xk_1` twice. It also runs through `CRS_row_ptr`, `CRS_col_idx` and `CRS_values`

once. The formula for calculating memory traffic is $3 \times 4'000'000 + 2 \times 31'976'004$ this would amount to 607 MB per iteration of the while-loop.

Listing 6: Generation of data.

```

1 #pragma omp for reduction(max:diff)
2 for( i=0; i<nodes; i++){
3     x[i] = 0;
4     for( j=CRS_row_ptr[i]; j<CRS_row_ptr[i+1]; j++)
5         x[i] += CRS_values[j] * xk_1[CRS_col_idx[j]];
6     x[i] *= d;
7     x[i] += Wk_1_product;
8     //Comuting the difference between x^k and x^k-1
9     //and adds the biggest diff to diffX[thread_id]
10    if( x[i]-xk_1[i] > diff ){
11        diff = x[i] - xk_1[i];
12    }
13 }
```

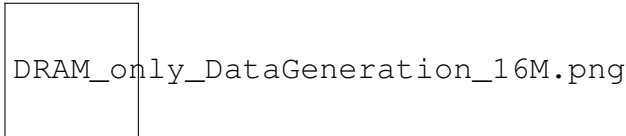



Table 2: Prediction of time taken for data generation, n=16M

3.1.2 Analyze

The analyze part run through the nodes array five times and add all the elements to the average variable. The average variable is divided by the number of elements in array, this is shown in the code in chapter 5.1. The memory traffic will amount of 160MB per while-loop iteration.

Listing 7: Analyzing the data.

```
1 //Analyze part
2 #pragma omp for reduction(+ : average)
3   for(i=0;i<nodes;i++){
4     average += xk_1[i];
5   }
6 #pragma omp for reduction(+ : average)
7   for(i=0;i<nodes;i++){
8     average += xk_1[i];
9   }
10 #pragma omp for reduction(+ : average)
11   for(i=0;i<nodes;i++){
12     average += xk_1[i];
13   }
14 #pragma omp for reduction(+ : average)
15   for(i=0;i<nodes;i++){
16     average += xk_1[i];
17   }
18 #pragma omp for reduction(+ : average)
19   for(i=0;i<nodes;i++){
20     average += xk_1[i];
21   }
```



DRAM_only_Analyze_16M.png

Table 3: Prediction of time taken for analyzing the data, n=16M

3.1.3 Stream benchmark, sum

This benchmark is the STREAM benchmark with an added benchmark. The sum is found by adding all the elements into a single variable. The STREAM benchmark was changed by increasing several array from four to five and added the sum benchmark after the four other benchmark. The STREAM benchmark for NVDIMM is the same as the one described above, but the code has been changed so the benchmark will read and write to the NVDIMM.

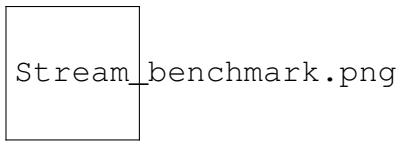


Table 4: New Stream benchmark, DRAM

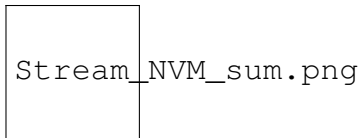


Table 5: New Stream benchmark, NVM

3.1.4 Calculation only

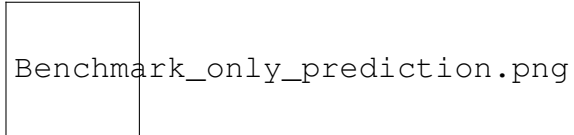


Table 6: Prediction of time taken for calculation of the data, with a code that only do calculation

3.2 NVDIMM Analyze only

Table 7: Measurement of analyzations only on NVDIMM.

3.3 NVM simulation

3.3.1 Locks

The program are divided into two parts, the calculation of data and the analyzing of the data that have been generated. The two parts synchronize by using two locks called lock_a and lock_b, lock_a will start in unlocked state and lock_b in locked state. When the two parts starts the analyzing part is put on hold by lock_b until the calculation part has generated the first set of data. Then the calculation part will lock lock_a, swap the pointers x and xk_1 and then unlock lock_b. The calculation part will then start the calculation of the next set of data, but wont swap pointers until analyzing part has transferred the content in xk_1 to NVDIMM and unlocked lock_a.

When the calculation part unlocks lock_b the analyzing will start transferring data from xk_1 to NVDIMM and unlock lock_a when it's done with the transfer. The analyzing part will then start analyzing the data on NVDIMM. When its done it will encounter lock_b and will wait there until calculation has a new set of data ready and has swapped the pointers.

Before and after the set lock in calculation and analyze part there is a time measurement that measure how long the threads have waited for the lock to be unlocked by the other part. All the individual times the threads have waited in calculation or analyze gets added to a variable called iteration_idle_time or transfer_idle_time that will be the total time the threads have waited.

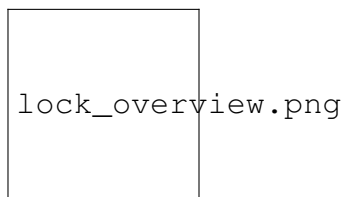


Figure 11: A simplified version of how lock works.

Listing 8: Calculation

```
1 while( n<5000 ){  
2     #pragma omp barrier  
3     /*  
4         Calculation of Data  
5     */
```



```

6  #pragma omp single
7  {
8      temp_time = mysecond();
9      omp_set_lock(&lock_a);
10     iteration_idle_time += mysecond() - temp_time;
11
12     temp_x = xk_1;
13     xk_1 = x;
14     x = temp_x;
15
16     omp_unset_lock(&lock_b);
17 }
18 } //end of while-loop

```

Listing 9: Analyze

```

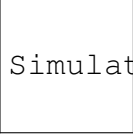
1  while(1==1){
2      #pragma omp single
3      {
4          temp_time = mysecond();
5          omp_set_lock(&lock_b);
6          transfer_idle_time += mysecond() - temp_time;
7          temp_time = mysecond();
8          average=0.0;
9      }
10     /*
11     Transfer of array from DRAM to NVDIMM
12     */
13     #pragma omp single
14     {
15         DRAM_to_NVM_time += mysecond() - temp_time;
16         omp_unset_lock(&lock_a);
17         temp_time = mysecond();
18     }
19     /*
20     Analyzations of data
21     */
22     }
23     #pragma omp barrier
24     #pragma omp single
25     {
26         Analyse_time += mysecond() - temp_time;
27     }

```

```

28 //if sentence for exiting while-loop.
29 if(iteration_ongoing==0){
30     break;
31 }
32 }


```



Simulation_NVM_2.png


Table 8: Simulation with both NVDIMM and DRAM

getconf LEVEL1_DCACHE_LINESIZE have been used to find the cacheline size in bytes.



Predicted_calculation.png

Table 9: Predicted calculationbased on papi.



Predicted_calculation_3.png

Table 10: Tried to split load and store misses. Server do not support store misses.



Array-counting.png

Table 11: Manually counted the number of int and double tranfers.

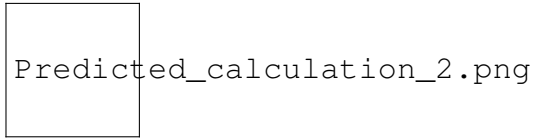


Table 12: Predicted calculationbased on papi with L3.

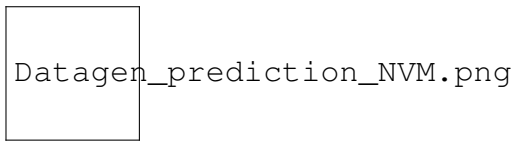


Table 13: Time prediction, data generation

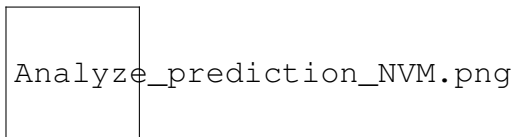


Table 14: Time prediction, transfer and analyze

3.4 2D-array test

Listing 10: Kildekode

```
1 https://github.com/SveinGunnar/Master\_Thesis\_2020/tree/master/ArrayCopyTest
```

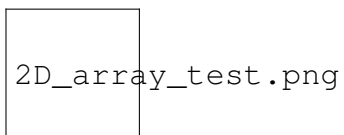


Table 15: 2D-Array test

4 Large array benchmark

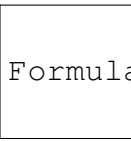
The size of data that must be analyzed keeps increasing year after year and the prize for DRAM are not getting cheaper. NVDIMM offer a lot of storage at a cheaper prize. This opens the opportunity to save money by offloading some of the data to the NVDIMM where the data will be analyzed the same way 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.

Calculation

This program have an two dimensional array filled with data. The program 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 program does this for every element between (1,1) and (m-2,n-2).

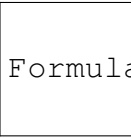
Formula

$$\frac{dram_data - nvdim_data}{dram_speed} = \frac{nvdim_data}{nvdim_speed}$$
$$nvdim_data = \frac{nvdim_speed * dram_data}{nvdim_speed + dram_speed}$$



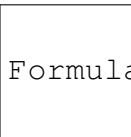
Formula_calculation.png

Table 16: Formula calculation



Formula_First_version.png

Table 17: First version



Formula_Second_version.png

Table 18: Second version

4.1 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 will be the last thread and in the NVDIMM group that would be the first group.

Listing 11: First version

```
1 while(k<K_length){
2     #pragma omp barrier
3     #pragma omp single
4     {
5         total_time[k] = mysecond();
6     }
7     #pragma omp barrier
8     if( thread_id < dram_threads ){
9         //for the thread bordering on nvdimmm thread.
10        if( thread_id==(dram_threads-1) ){
11            individual_time[thread_id] = mysecond();
12            for( i=slice_start; i<slice_end-1; i++){
13                for( j=1; j<nMinusOne; j++){
14                    temp = A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+
15                        A[i][j-1] + A[i][j+1]+
16                        A[i+1][j-1] + A[i+1][j] + A[i+1][j+1];
17                    B[i][j] = temp*inverseEigth;
18                }
19            }
20
21            i = slice_end-1;
22            for( j=1; j<nMinusOne; j++){
23                temp = A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+
24                    A[i][j-1] + A[i][j+1]+
25                    D_RO(C)[i*n+j] + D_RO(C)[i*n+j] +
26                    D_RO(C)[i*n+j];
27                B[i][j] = temp*inverseEigth;
28            }
29            individual_time[thread_id] = mysecond() -
30                individual_time[thread_id];
31        }else{
32            //For all the threads not bordering with nvdimmm.
33            individual_time[thread_id] = mysecond();
34            for( i=slice_start; i<slice_end; i++){
35                for( j=1; j<nMinusOne; j++){
36                    temp = A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+
37                        A[i][j-1] + A[i][j+1]+
38                        A[i+1][j-1] + A[i+1][j] + A[i+1][j+1];
39                    B[i][j] = temp*inverseEigth;
40                }
41            }
42            individual_time[thread_id] = mysecond() -
```

```

        individual_time[thread_id];
41     }
42 }else{
43     if( thread_id==dram_threads ){
44         individual_time[thread_id] = mysecond();
45         i=0;
46         for( j=1; j<nMinusOne; j++){
47             temp =
48                 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)] +
50                 D_RO(C)[i*n+(j+1)]+
51                 D_RO(C)[(i+1)*n+(j-1)] + D_RO(C)[(i+1)*n+j] +
52                 D_RO(C)[(i+1)*n+(j+1)];
53             D_RW(D)[i*n+j] = temp*inverseEigth;
54         }
55         for( i=slice_start+1; i<slice_end-1; i++){
56             for( j=1; j<nMinusOne; j++){
57                 temp = D_RO(C)[(i-1)*n+(j-1)] +
58                     D_RO(C)[(i-1)*n+j] + D_RO(C)[(i-1)*n+(j+1)]+
59                     D_RO(C)[i*n+(j-1)] +
60                     D_RO(C)[i*n+(j+1)]+
61                     D_RO(C)[(i+1)*n+(j-1)] +
62                     D_RO(C)[(i+1)*n+j] +
63                     D_RO(C)[(i+1)*n+(j+1)];
64                 D_RW(D)[i*n+j] = temp*inverseEigth;
65             }
66         }
67         individual_time[thread_id] = mysecond() -
68             individual_time[thread_id];
69     }else{
70         individual_time[thread_id] = mysecond();
71         for( i=slice_start; i<slice_end; i++){
72             for( j=1; j<nMinusOne; j++){
73                 temp = D_RO(C)[(i-1)*n+(j-1)] +
74                     D_RO(C)[(i-1)*n+j] + D_RO(C)[(i-1)*n+(j+1)]+
75                     D_RO(C)[i*n+(j-1)] +
76                     D_RO(C)[i*n+(j+1)]+
77                     D_RO(C)[(i+1)*n+(j-1)] +
78                     D_RO(C)[(i+1)*n+j] +
79                     D_RO(C)[(i+1)*n+(j+1)];
80                 D_RW(D)[i*n+j] = temp*inverseEigth;
81             }
82         }
83     }
84 }

```



```

71         individual_time[thread_id] = mysecond() -
            individual_time[thread_id];
72     }
73 }
74 #pragma omp barrier
75 #pragma omp single
76 {
77     total_time[k] = mysecond() - total_time[k];
78     dram_time[k]=individual_time[0];
79     for(i=1;i<dram_threads;i++){
80         if(dram_time[k]<individual_time[i])
81             dram_time[k]=individual_time[i];
82     }
83     nvdimmm_time[k]=individual_time[dram_threads];
84     for(i=dram_threads+1;i<dram_threads+nvdimmm_threads;i++){
85         if(nvdimmm_time[k]<individual_time[i])
86             nvdimmm_time[k]=individual_time[i];
87     }
88     k++;
89 }
90 #pragma omp barrier
91 }//End of while

```

4.2 Second version

Listing 12: Second version

```

1  while(k<K_length){
2      #pragma omp barrier
3      #pragma omp single
4      {
5          total_time[k] = mysecond();
6      }
7      #pragma omp barrier
8      if( thread_id < dram_threads ){
9          individual_time[thread_id] = mysecond();
10         for( i=slice_start; i<slice_end; i++){
11             for( j=1; j<nMinusOne; j++){
12                 temp = A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]+
13                     A[i][j-1] + A[i][j+1]+
14                     A[i+1][j-1] + A[i+1][j] + A[i+1][j+1];
15                 B[i][j] = temp*inverseEigth;

```

```

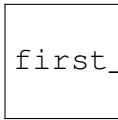
16         }
17     }
18     individual_time[thread_id] = mysecond() -
        individual_time[thread_id];
19 }else{
20     individual_time[thread_id] = mysecond();
21     for( i=slice_start; i<slice_end; i++){
22         for( j=1; j<nMinusOne; j++){
23             temp = D_RO(C)[(i-1)*n+(j-1)] +
                D_RO(C)[(i-1)*n+j] + D_RO(C)[(i-1)*n+(j+1)] +
24             D_RO(C)[i*n+(j-1)] +
                D_RO(C)[i*n+(j+1)] +
25             D_RO(C)[(i+1)*n+(j-1)] + D_RO(C)[(i+1)*n+j]
                + D_RO(C)[(i+1)*n+(j+1)];
26             D_RW(D)[i*n+j] = temp*inverseEigth;
27         }
28     }
29     individual_time[thread_id] = mysecond() -
        individual_time[thread_id];
30 }
31 #pragma omp barrier
32 #pragma omp single
33 {
34     total_time[k] = mysecond() - total_time[k];
35     dram_time[k]=individual_time[0];
36     for(i=1;i<dram_threads;i++){
37         if(dram_time[k]<individual_time[i])
38             dram_time[k]=individual_time[i];
39     }
40     nvdimmm_time[k]=individual_time[dram_threads];
41     for(i=dram_threads+1;i<dram_threads+nvdimmm_threads;i++){
42         if(nvdimmm_time[k]<individual_time[i])
43             nvdimmm_time[k]=individual_time[i];
44     }
45     k++;
46 }
47 #pragma omp barrier
48 }//End of while

```



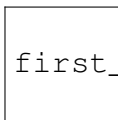
First_version_DRAM_only_17_9_21.png

Table 19: First version, dram only



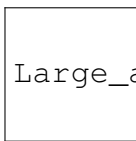
first_version_more_detailed_1.png

Table 20: First version, more detailed 1



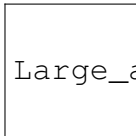
first_version_more_detailed_2.png

Table 21: First version, more detailed 2.



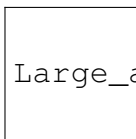
Large_array_test_first_version_v5.png

Table 22: First version.



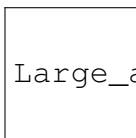
Large_array_test_second_version_v3.png

Table 23: Second version.



Large_array_test_first_version_v2.png

Table 24: First version. OLD



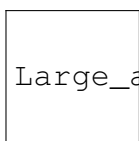
Large_array_test_first_version_v3.png

Table 25: First version. OLD



Large_array_test_second_version_v2.png

Table 26: Second version. OLD



Large_array_test_second_version_NVDIMM_only_v2.png

Table 27: NVDIMM only of second version. OLD



DRAM_only_on_n50.png

Table 28: DRAM only on n50. OLD