1 Benchmarks

1.1 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.

1.1.1 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.

1.2 STREAM DRAM

The STREAM[1] 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 what was expected, 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.

This means that it might be possible to allocate five of the sixteen threads to work on NVDIMM and not loose a significant amount of performance for the eleven remaining threads that are still working on DRAM.

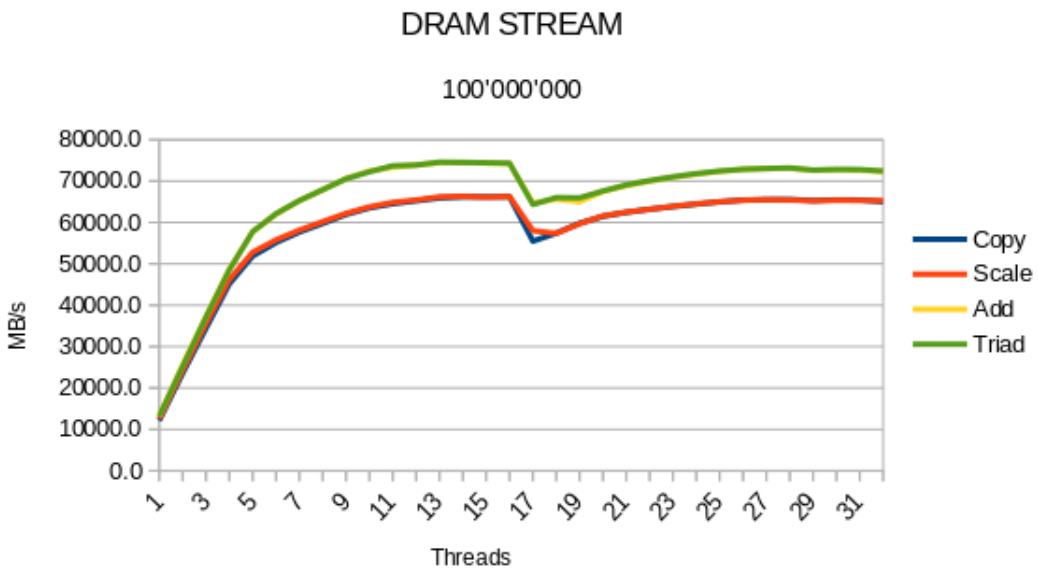

Figure 1: DRAM Stream, 100’000’000 elements

	DRAM STREAM 100'000'000			
	Copy	Scale	Add	Triad
1	12143.9	12585.3	13217.8	13189.5
2	23511.1	24199.9	25549.6	25207.2
3	34425.6	35527.3	37202.9	37129.4
4	45148.0	46198.5	48695.5	48527.2
5	51879.2	52805.5	57734.0	57779.7
6	55134.3	55781.8	62010.8	62113.3
7	57672.5	58137.6	65254.3	65275.9
8	59757.5	60179.8	67911.6	67902.0
9	61919.4	62198.9	70544.4	70542.4
10	63491.7	63750.5	72417.6	72237.2
11	64511.0	64845.7	73333.9	73682.8
12	65220.7	65415.3	73821.2	73861.8
13	65911.3	66175.8	74416.6	74555.5
14	66145.8	66315.7	74504.1	74488.2
15	66170.6	66121.0	74340.2	74423.2
16	66181.7	66283.0	74229.4	74325.9
17	55471.0	57998.5	64697.0	64334.7
18	57395.3	57347.7	65624.0	66015.6
19	59797.4	59696.9	64989.3	65950.3
20	61460.1	61578.5	67489.5	67586.5
21	62423.9	62492.5	68884.2	69070.9
22	63176.1	63188.6	69985.3	70101.7
23	63834.2	63894.9	70899.1	71022.7
24	64420.0	64489.9	71663.3	71792.1
25	64977.6	64995.8	72343.7	72419.6
26	65385.9	65359.2	72817.8	72844.7
27	65533.4	65506.6	73028.0	73025.9
28	65579.5	65436.9	73110.2	73161.8
29	65268.3	65191.6	72590.9	72650.1
30	65452.9	65370.0	72696.3	72800.4
31	65415.3	65322.3	72617.1	72784.6
32	65006.6	65265.8	72343.7	72398.3

Table 1: DRAM Stream, 100’000’000 elements

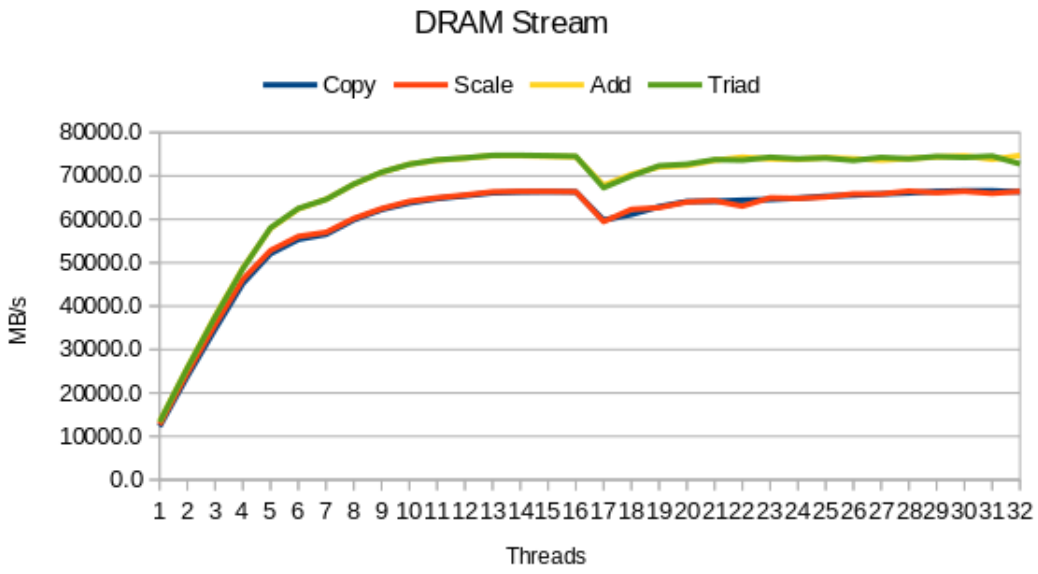

Figure 2: DRAM Stream, Gammel

1.3 STREAM NVDIMM

The stream NVDIMM benchmark measure the memory speed of the NVDIMM. This benchmark is the same as the STREAM benchmark has been descibed in chapter 1.2. The different is that the memory type have been changed from DRAM to NVDIMM. The code shown in listing 1 is part of the original code that have been removed from the code.

Listing 1: Original STREAM benchmark code at line 175-181.

```
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 has been replaced with the code shown in listing 2. The code starts by opening the memory pool at line 21-27. The code will use a method called initiate at line 28 that will initiate the three arrays. Once this is done the code will continue executing the rest of the STREAM benchmark code which is identical to the original STREAM benchmark code.

Listing 2: Code that has replaced original code.

```
1  PMEMobjpool *pop;
2  POBJ_LAYOUT_BEGIN(array);
3  POBJ_LAYOUT_TOID(array, double);
4  POBJ_LAYOUT_END(array);
5  //Declearing the arrays
6  TOID(double) a;
7  TOID(double) b;
8  TOID(double) c;
9
10 void initiate()
11 {
12   //Initiating the arrays.
13   POBJ_ALLOC(pop, &a, double,
        (STREAM_ARRAY_SIZE+OFFSET)*sizeof(STREAM_TYPE), NULL,
        NULL);
14   POBJ_ALLOC(pop, &b, double,
        (STREAM_ARRAY_SIZE+OFFSET)*sizeof(STREAM_TYPE), NULL,
        NULL);
```

```
15    POBJ_ALLOC(pop, &c, double,
          (STREAM_ARRAY_SIZE+OFFSET)*sizeof(STREAM_TYPE), NULL,
          NULL);
16  }
17
18  int main()
19  {
20    const char path[] = "/mnt/pmem0-xfs/pool.obj";
21    pop = pmemobj_create(path, LAYOUT_NAME, 10737418240,
          0666);
22    if (pop == NULL)
23      pop = pmemobj_open(path, LAYOUT_NAME);
24    if (pop == NULL) {
25      perror(path);
26      exit(1);
27    }
28    initiate();
29    //The rest of the STREAM benchmark after this.
30  }
```

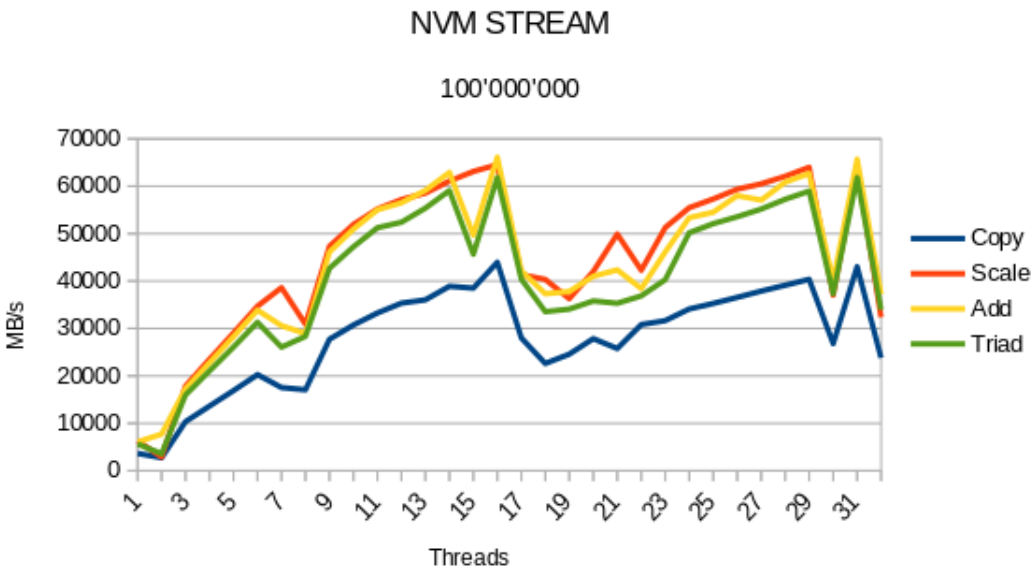

Figure 3: NVDIMM Stream, 100’000’000 elements

	Copy	Scale	Add	Triad
1	3647.2	6268.5	6072.5	5629
2	2719.8	2889.4	7708.7	3579.6
3	10350.8	17840.8	17298.7	15964.5
4	13641.5	23513.1	22793.1	21049.5
5	16894.4	29050.3	28273.8	26068.8
6	20265.5	34692.9	33867.3	31296.9
7	17533.2	38607.4	30576.8	26039.4
8	17069.6	30887.5	28925.5	28337
9	27728.6	47323.1	46128.2	42681.1
10	30723.7	51934.6	50947.9	47230.1
11	33257.1	55176.4	55028	51238.3
12	35331.8	57182.1	56505.1	52396.1
13	36040.1	58539.8	59105.9	55363
14	38893.5	61082.5	62896.1	59064
15	38500.6	63116.7	49695.5	45669.1
16	43959.7	64521.5	66122.8	61809.7
17	27986.6	41352.2	42130.5	40353.8
18	22615.1	40333.7	37307.2	33533.5
19	24570.8	36265.7	37788.8	34077.4
20	27851.3	42018.9	41013.1	35795.7
21	25762.4	49905.1	42352.8	35300.9
22	30820.2	42276.5	38398.7	36864.1
23	31653.6	51273.9	46133.3	40276.6
24	34094.8	55453.2	53348.8	50169.4
25	35277.9	57302.7	54510.7	52099.4
26	36558	59360.5	57992.1	53558.3
27	37881.4	60498.8	57062.8	55178.8
28	39113.2	62085.5	60856.5	57245.1
29	40363.3	63969.3	62679.5	58945
30	26802.1	37024.2	39942.1	37342.5
31	43011.9	64133.1	65694.2	61791.9
32	23808.5	32356.6	37162.5	33923.6

Table 2: NVDIMM Stream, 100’000’000 elements

	NVM STREAM 20GB			
	Copy	Scale	Add	Triad
1	1534.5	2269.9	2175.4	2325.7
2	2196.1	2560.9	6801.6	3216.8
3	6432.2	10650.0	10198.8	9404.2
4	8307.8	12921.3	13213.3	11562.8
5	9942.1	18827.1	17788.5	16620.8
6	11548.5	19840.6	19084.3	18638.0
7	14049.5	23887.1	22385.9	21206.7
8	15926.7	26563.6	25914.3	23795.2
9	18073.7	30424.5	29111.7	27281.1
10	20054.2	34149.0	32817.0	30627.3
11	22908.7	39059.6	37225.2	33025.5
12	23306.5	40364.1	39846.6	36301.7
13	26541.5	43092.4	43114.6	40241.1
14	28994.7	41673.4	46557.7	42332.3
15	28128.7	45687.3	43216.7	38941.2
16	29851.8	47268.3	43722.5	36094.5
17	23868.8	39847.9	36990.4	34928.9
18	25482.8	40696.9	39725.1	35272.5
19	25951.3	42525.3	41027.2	37471.4
20	28299.6	45874.7	43539.6	41503.7
21	28401.9	46766.1	44525.5	41259.7
22	28668.6	48732.7	44690.3	42046.9
23	29773.1	45643.1	42763.8	39977.5
24	29302.6	42847.3	44418.7	41323.7
25	29173.6	45911.8	44880.7	42645.2
26	28936.8	47565.8	45983.8	42203.4
27	29714.1	50534.2	46335.5	43239.9
28	31421.9	53229.2	48906.1	45456.2
29	32937.0	54783.3	53012.2	46619.2
30	34182.5	56887.2	54071.7	34449.6
31	35035.9	54007.4	50137.1	46001.1
32	29957.0	50851.2	43928.3	43147.4

Table 3: NVDIMM Stream, 2’000’000’000 elements

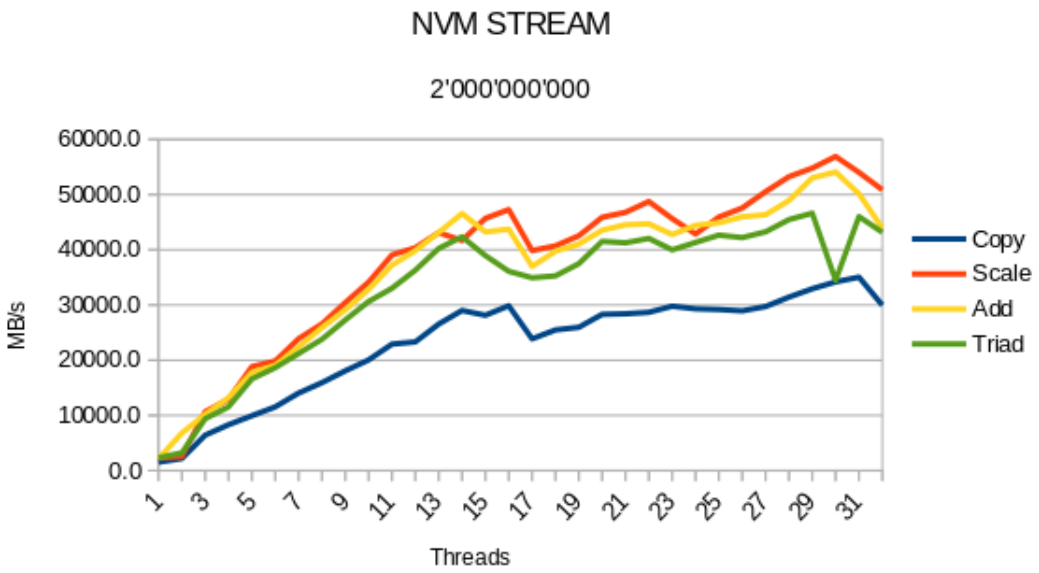

Figure 4: NVDIMM Stream, 2'000'000'000 elements

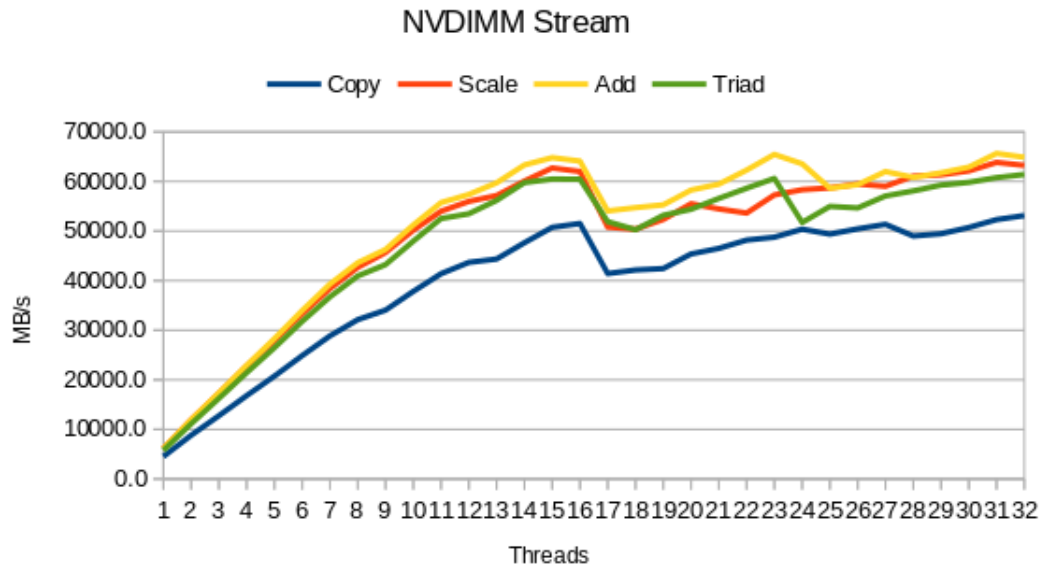

Figure 5: NVDIMM Stream, Gammel måling

1.4 3 benchmarks

This chapter are about three different benchmarks. In the first benchmark data will be copied for a DRAM array to another DRAM array and from a NVDIMM array to another NVDIMM array simultaneously. In the second benchmark data will be copied from DRAM-DRAM arrays and from DRAM-NVDIMM arrays simultaneously. In the last benchmark data will be copied from DRAM-DRAM and NVDIMM-DRAM simultaneously.

The purpose of these benchmarks is to get an understanding of how performance will be affected when different threads generates traffic from DRAM and NVDIMM simultaneously. That is why there is three types of benchmarks in order to test all possible combination of traffic. Does a combination of DRAM and NVDIMM threads exist where the total bandwidth of DRAM and NVDIMM threads exceeds what a DRAM alone can archive.

1.4.1 NVM-NVM

The code for this benchmark are described in listing 3. From line 2-14 the code is declaring variables and creating the arrays where the result from the benchmark will be stored. There will be declared two DRAM array and two NVDIMM arrays that will be used in the benchmark. When the threads arrive at line 16 they will synchronize before they are split into two group. Threads with a thread id lower than the number of DRAM threads will pass the if-sentence at line 17, where they will initiate two DRAM array and place values into each element. The rest of the threads will move on to line 27 and enter this bracket. These threads will initiate two NVDIMM arrays and place values into each element. All the threads will then synchronize at line 40 before they will divide into two group at line 41 in the same fashion they did in line 17. All the threads will then start to copy data from one array to the other array. The DRAM threads will copy from DRAM-DRAM array and the NVDIMM threads will copy from NVDIMM-NVDIMM array. They will repeat this for as many times as the user of the benchmark has decided by defining the total_test variable as an argument in the command line when the program was started. The time measurement will be started at the beginning of the for-loop at line 45 or 54 and end at the end of the for-loop at line 50 or 60. When the benchmark testing is over the threads will free up their arrays and the benchmark will print out the result.

When the threads pass the barrier at line 40 and begin the benchmark test they will never synchronize another time. Because of this the DRAM threads will complete their tasks a lot earlier then NVDIMM threads because DRAM speed is faster then NVIDMM speed. This also means that once the fastest thread is done the rest of the threads will share more bandwidth among themself and become faster. When more and more threads complete their tasks the faster the remaining threads will become. In order to get a correct benchmark where all threads have been working the user need to throw out the data where some threads are working when other threads have completed their tasks. Throwing out the last one third is usually enough. There is also a need to throw out atleast the first 25 iterations. This is because the NVDIMM is a lot slower to get started then DRAM.

Listing 3: Benchmark code.

```
1  #pragma omp parallel
2  {
3    //Declearing variables.
4    int thread_id = omp_get_thread_num();
5    int i,j;
6    double *drm_read_array;
7    double *drm_write_array;
8    TOID(double) nvm_read_array;
9    TOID(double) nvm_write_array;
10   srand((unsigned int)time(NULL));
11   #pragma omp master
12   {
13     /* Creates array where the test result will be added.
         */
14   }
15   //Creates all the arrays needed for the test.
16   #pragma omp barrier
17   if(thread_id < totalThreads-nvmThreads){
18     drm_read_array =
          (double*)malloc(ARRAY_LENGTH*sizeof(double));
19       drm_write_array =
            (double*)malloc(ARRAY_LENGTH*sizeof(double));
20     #pragma omp critical
21     {
22       for(i=0;i<ARRAY_LENGTH;i++){
23         drm_read_array[i] =
              ((double)rand()/(double)(RAND_MAX));
```

```
24            drm_write_array[i] =
                  ((double)rand()/(double)(RAND_MAX));
25          }
26        }
27      }else if(thread_id >= totalThreads-nvmThreads){
28        POBJ_ALLOC(pop, &nvm_read_array, double,
              sizeof(double) * ARRAY_LENGTH, NULL, NULL);
29        POBJ_ALLOC(pop, &nvm_write_array, double,
              sizeof(double) * ARRAY_LENGTH, NULL, NULL);
30        #pragma omp critical
31        {
32          for(i=0;i<ARRAY_LENGTH;i++){
33            D_RW(nvm_read_array)[i] =
                  ((double)rand()/(double)(RAND_MAX));
34            D_RW(nvm_write_array)[i] =
                  ((double)rand()/(double)(RAND_MAX));
35          }
36          //printf("NVM thread_id: %d, %f\n", thread_id,
                D_RO(nvm_read_array)[11235]);
37        }
38      }
39      //Doing the test.
40      #pragma omp barrier
41      if(thread_id < totalThreads-nvmThreads){
42        //From DRAM to DRAM:
43        for(i=0;i<total_tests;i++){
44          //Time start
45          test_time[thread_id][i] = mysecond();
46          for(j=0;j<ARRAY_LENGTH;j++){
47            drm_write_array[j] = drm_read_array[j];
48          }
49          //Time stop.
50          test_time[thread_id][i] = mysecond() -
                test_time[thread_id][i];
51        }
52      }else if(thread_id >= totalThreads-nvmThreads){
53        //From NVM to NVM:
54        for(i=0;i<total_tests;i++){
55          //Time start
56          test_time[thread_id][i] = mysecond2();
57          for(j=0;j<ARRAY_LENGTH;j++)
58            D_RW(nvm_write_array)[j] =
                  D_RO(nvm_read_array)[j];
```

```
59          //Time stop.
60          test_time[thread_id][i] = mysecond2() -
                test_time[thread_id][i];
61        }
62      }else
63        printf("ERROR\n");
64      /* Freeing up DRAM and NVDIMM arrays */
65  }
```

	NVM-NVM DRAM	16GB NVDIMM	SUM
1	62713.21	2519.19	65232.40
2	58458.38	5919.25	64377.62
3	55062.77	8123.82	63186.59
4	52461.90	11463.70	63925.59
5	48391.34	13890.82	62282.16
6	44665.65	17745.60	62411.25
7	42584.70	18978.09	61562.79
8	39013.32	20679.79	59693.11
9	34744.85	24145.36	58890.21
10	31167.82	27854.89	59022.71
11	27352.41	30522.81	57875.22
12	23329.42	34424.44	57753.86
13	18577.03	35012.49	53589.52
14	14095.33	38166.42	52261.75
15	6904.22	38987.55	45891.77

Table 4: NVM-NVM, 16 GB

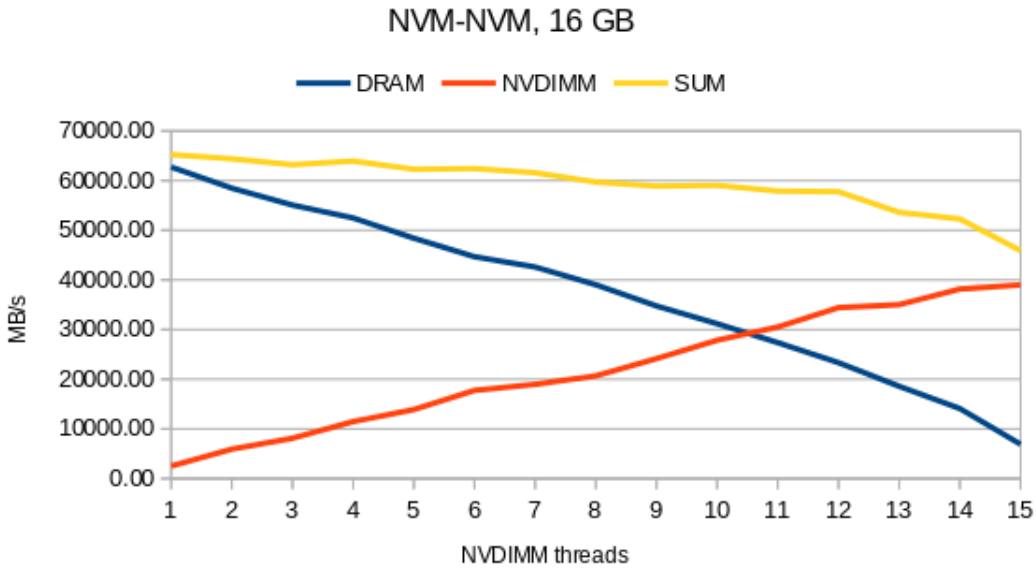

Figure 6: NVM-NVM, 16 GB

	NVM-NVM	80GB	
	DRAM	NVDIMM	SUM
1	60406.49	2295.23	62701.71
2	57505.90	5876.92	63382.82
3	54141.64	8618.66	62760.30
4	50581.48	12416.96	62998.44
5	46607.93	16007.46	62615.38
6	42703.94	18831.07	61535.01
7	39398.09	22254.55	61652.64
8	35483.75	25020.22	60503.97
9	32182.08	29616.33	61798.41
10	28875.53	32740.51	61616.04
11	26478.87	37293.93	63772.80
12	22967.14	42703.07	65670.20
13	23554.87	45971.20	69526.07
14	16848.35	48878.36	65726.71
15	8475.39	52861.01	61336.40

Table 5: NVM-NVM, 80 GB

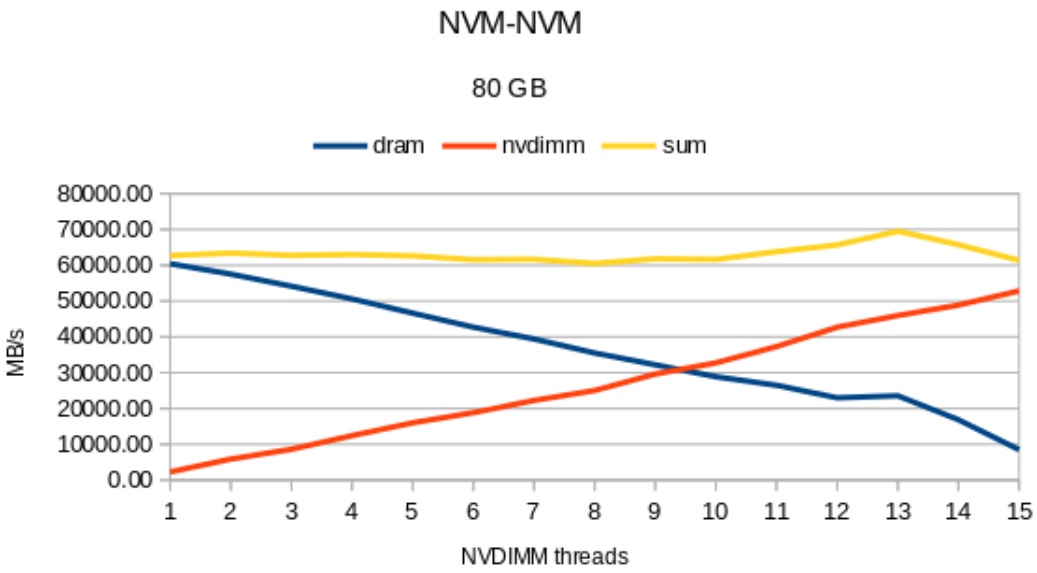

Figure 7: NVM-NVM, 80 GB

New graphs with second on x-axis and bandwidth on Y-axis. 300 iterations.

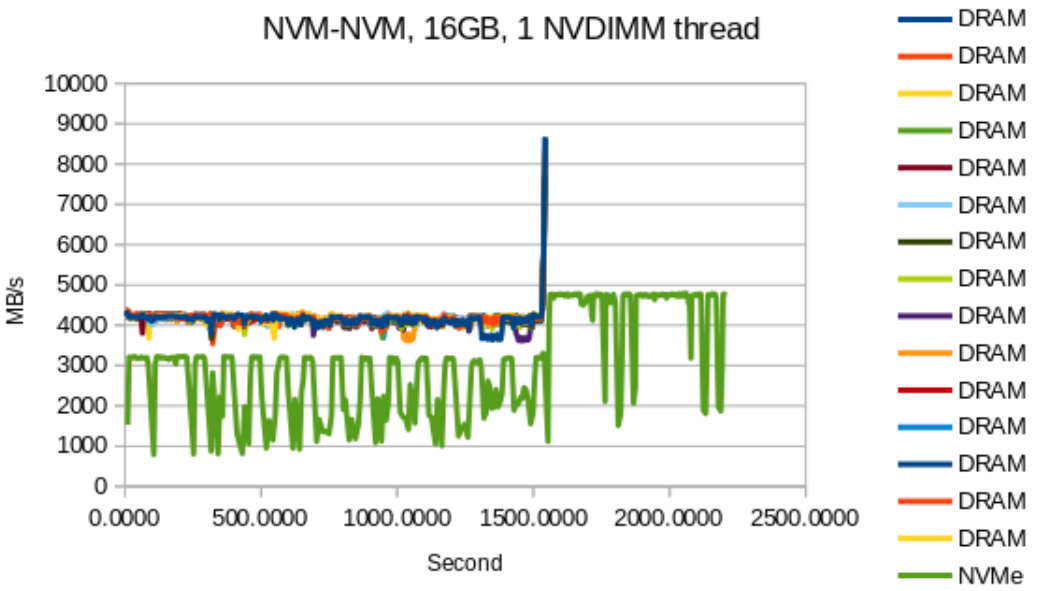

Figure 8: NVM-NVM, 16 GB, 1 NVDIMM Thread

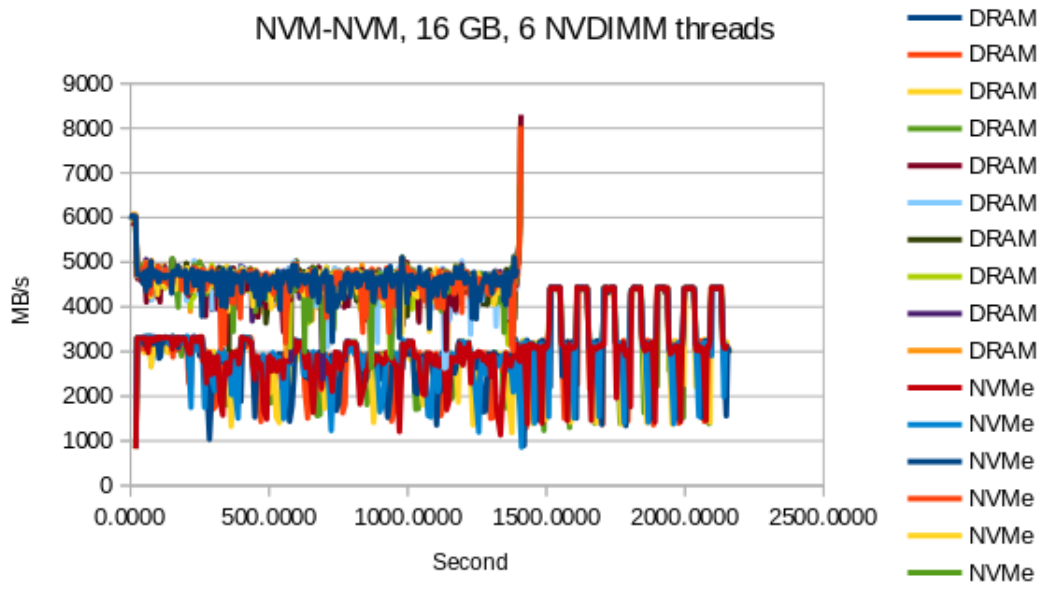

Figure 9: NVM-NVM, 16 GB, 6 NVDIMM Threads

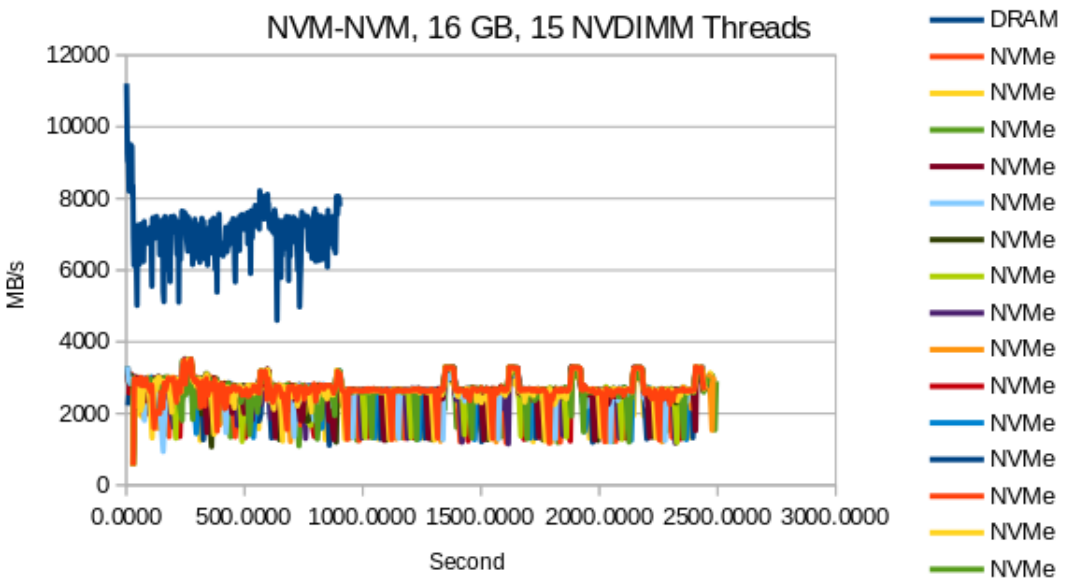

Figure 10: NVM-NVM, 16 GB, 15 NVDIMM Threads

1.4.2 NVM-DRAM

In this version of the benchmark there will be one group of threads that transfer data from DRAM-DRAM and another group of threads that will transfer data from NVDIMM-DRAM. This code is very similar to the previous code. The differences are at line 26-36 where the code will initiate and add values to one DRAM array and one NVDIMM array instead of two NVDIMM arrays. The other difference is at line 51-61 where the code will copy data from NVDIMM-DRAM instead of NVDIMM-NVDIMM.

Listing 4: Benchmark code.

```
1  #pragma omp parallel
2  {
3    int thread_id = omp_get_thread_num();
4    int i,j;
5    double *drm_read_array;
6    double *drm_write_array;
7    TOID(double) nvm_read_array;
8    srand((unsigned int)time(NULL));
9    #pragma omp master
10   {
11     /* Creates array where the test result will be added.
         */
12   }
13   //Creates all the arrays needed for the test.
14   #pragma omp barrier
15   if(thread_id < totalThreads-nvmThreads){
16     drm_read_array =
           (double*)malloc(ARRAY_LENGTH*sizeof(double));
17     drm_write_array =
           (double*)malloc(ARRAY_LENGTH*sizeof(double));
18     #pragma omp critical
19     {
20       for(i=0;i<ARRAY_LENGTH;i++){
21         drm_read_array[i] =
               ((double)rand()/(double)(RAND_MAX));
22         drm_write_array[i] =
               ((double)rand()/(double)(RAND_MAX));
23       }
24     }
25   }
26   else if(thread_id >= totalThreads-nvmThreads){
```

```
27      drm_write_array =
            (double*)malloc(ARRAY_LENGTH*sizeof(double));
28      POBJ_ALLOC(pop, &nvm_read_array, double,
            sizeof(double) * ARRAY_LENGTH, NULL, NULL);
29      #pragma omp critical
30      {
31        for(i=0;i<ARRAY_LENGTH;i++){
32          D_RW(nvm_read_array)[i] =
                ((double)rand()/(double)(RAND_MAX));
33          drm_write_array[i] =
                ((double)rand()/(double)(RAND_MAX));
34        }
35      }
36    }
37    //Doing the test.
38    #pragma omp barrier
39    if(thread_id < totalThreads-nvmThreads){
40      //From DRAM to DRAM:
41      for(i=0;i<total_tests;i++){
42        //Time start
43        test_time[thread_id][i] = mysecond();
44        for(j=0;j<ARRAY_LENGTH;j++){
45          drm_write_array[j] = drm_read_array[j];
46        }
47        //Time stop.
48        test_time[thread_id][i] = mysecond() -
              test_time[thread_id][i];
49      }
50    }
51    else if(thread_id >= totalThreads-nvmThreads){
52      //From NVM to DRAM:
53      for(i=0;i<total_tests;i++){
54        //Time start
55        test_time[thread_id][i] = mysecond();
56        for(j=0;j<ARRAY_LENGTH;j++)
57          drm_write_array[j] = D_RO(nvm_read_array)[j];
58        //Time stop.
59        test_time[thread_id][i] = mysecond() -
              test_time[thread_id][i];
60      }
61    }
62    else
63      printf("ERROR\n");
```

```
64    /* Freeing up DRAM and NVDIMM arrays */
65 }
```

	NVM-DRAM	16GB	
	DRAM	NVDIMM	SUM
1	62453.64	3783.70	66237.34
2	58494.93	7498.73	65993.66
3	54875.77	11389.55	66265.32
4	50722.40	15186.09	65908.48
5	46805.50	19236.51	66042.01
6	42939.75	23048.58	65988.33
7	38685.69	26942.67	65628.36
8	34762.19	31106.72	65868.91
9	30744.19	35116.96	65861.14
10	26630.12	39415.57	66045.69
11	22192.83	43084.15	65276.98
12	17966.14	47951.27	65917.41
13	13703.78	51471.26	65175.04
14	9188.70	55298.75	64487.45
15	4402.69	60441.55	64844.24

Table 6: NVM-DRAM, 16 GB

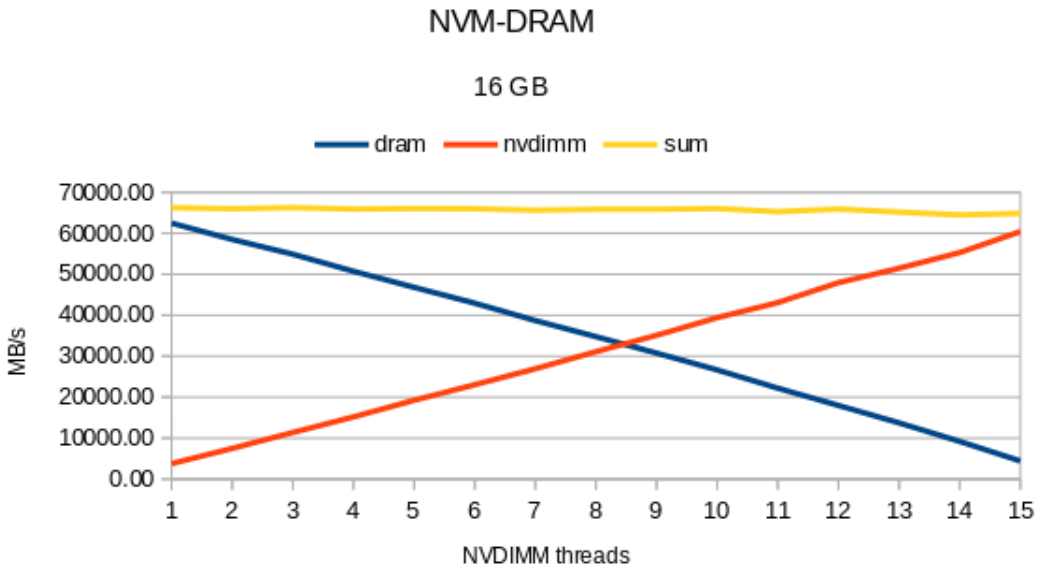

Figure 11: NVM-DRAM, 16 GB

1.4.3 DRAM-NVM

In this version of the benchmark there will be one group of threads that transfer data from DRAM-DRAM and another group of threads that will transfer data from DRAM-NVDIMM. This code only have one difference when compared to the code in 1.4.2. The difference is found in line 51-61 where data is transferred from DRAM-NVDIMM insted of from NVDIMM-DRAM.

Listing 5: Benchmark code.

```
1  #pragma omp parallel
2  {
3    int thread_id = omp_get_thread_num();
4    int i,j;
5    double *drm_read_array;
6    double *drm_write_array;
7    TOID(double) nvm_write_array;
8    srand((unsigned int)time(NULL));
9    #pragma omp master
10   {
11     /* Creates array where the test result will be added.
         */
12   }
13   //Creates all the arrays needed for the test.
14   #pragma omp barrier
15   if(thread_id < totalThreads-nvmThreads){
16     drm_read_array =
           (double*)malloc(ARRAY_LENGTH*sizeof(double));
17     drm_write_array =
           (double*)malloc(ARRAY_LENGTH*sizeof(double));
18     #pragma omp critical
19     {
20       for(i=0;i<ARRAY_LENGTH;i++){
21         drm_read_array[i] =
               ((double)rand()/(double)(RAND_MAX));
22         drm_write_array[i] =
               ((double)rand()/(double)(RAND_MAX));
23       }
24     }
25   }
26   else if(thread_id >= totalThreads-nvmThreads){
27     drm_read_array =
           (double*)malloc(ARRAY_LENGTH*sizeof(double));
```

```
28      POBJ_ALLOC(pop, &nvm_write_array, double,
           sizeof(double) * ARRAY_LENGTH, NULL, NULL);
29      #pragma omp critical
30      {
31        for(i=0;i<ARRAY_LENGTH;i++){
32          drm_read_array[i] =
               ((double)rand()/(double)(RAND_MAX));
33          D_RW(nvm_write_array)[i] =
               ((double)rand()/(double)(RAND_MAX));
34        }
35      }
36    }
37    //Doing the test.
38    #pragma omp barrier
39    if(thread_id < totalThreads-nvmThreads){
40      //From DRAM to DRAM:
41      for(i=0;i<total_tests;i++){
42        //Time start
43        test_time[thread_id][i] = mysecond();
44        for(j=0;j<ARRAY_LENGTH;j++){
45          drm_write_array[j] = drm_read_array[j];
46        }
47        //Time stop.
48        test_time[thread_id][i] = mysecond() -
             test_time[thread_id][i];
49      }
50    }
51    else if(thread_id >= totalThreads-nvmThreads){
52      //From DRAM to NVM:
53      for(i=0;i<total_tests;i++){
54        //Time start
55        test_time[thread_id][i] = mysecond();
56        for(j=0;j<ARRAY_LENGTH;j++)
57          D_RW(nvm_write_array)[j] = drm_read_array[j];
58        //Time stop.
59        test_time[thread_id][i] = mysecond() -
             test_time[thread_id][i];
60      }
61    }
62    else
63      printf("ERROR\n");
64    /* Freeing up DRAM and NVDIMM arrays */
65  }
```

	DRAM-NVM	16GB	
	DRAM	NVDIMM	SUM
1	61509.33	1970.46	63479.80
2	56723.35	6201.00	62924.35
3	53781.05	10121.76	63902.80
4	50036.13	13694.33	63730.46
5	45775.77	17819.93	63595.69
6	41973.60	21447.00	63420.60
7	37594.78	24884.46	62479.25
8	33978.96	29499.69	63478.65
9	29813.82	32596.12	62409.95
10	26008.07	38262.03	64270.10
11	21701.21	40146.44	61847.65
12	17201.25	43910.70	61111.95
13	12893.72	47396.46	60290.18
14	8804.73	51979.44	60784.17
15	4351.54	55995.01	60346.55

Table 7: DRAM-NVM, 16 GB

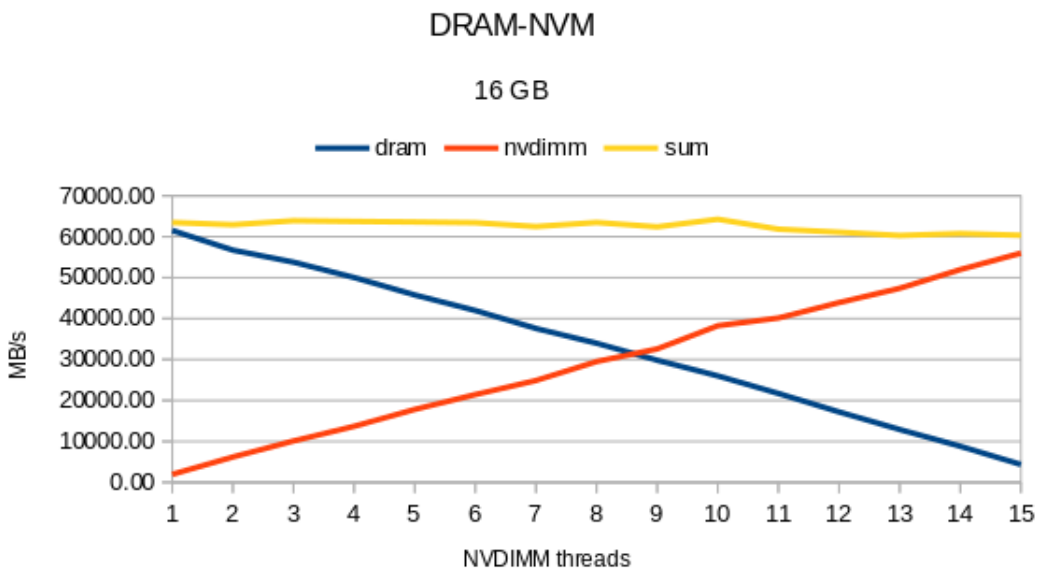

Figure 12: DRAM-NVM, 16 GB

References

[1] John D. McCalpin. *STREAM source code*. URL: `https://www.cs.virginia.edu/stream/FTP/Code/stream.c` (visited on 12/20/2020).