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 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.

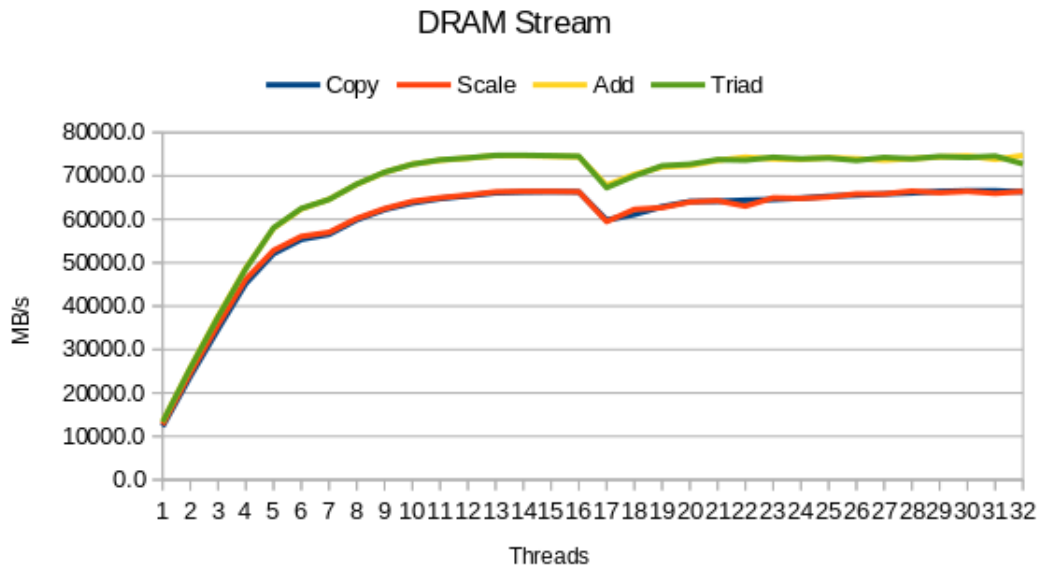

Figure 1: DRAM Stream

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  }
```

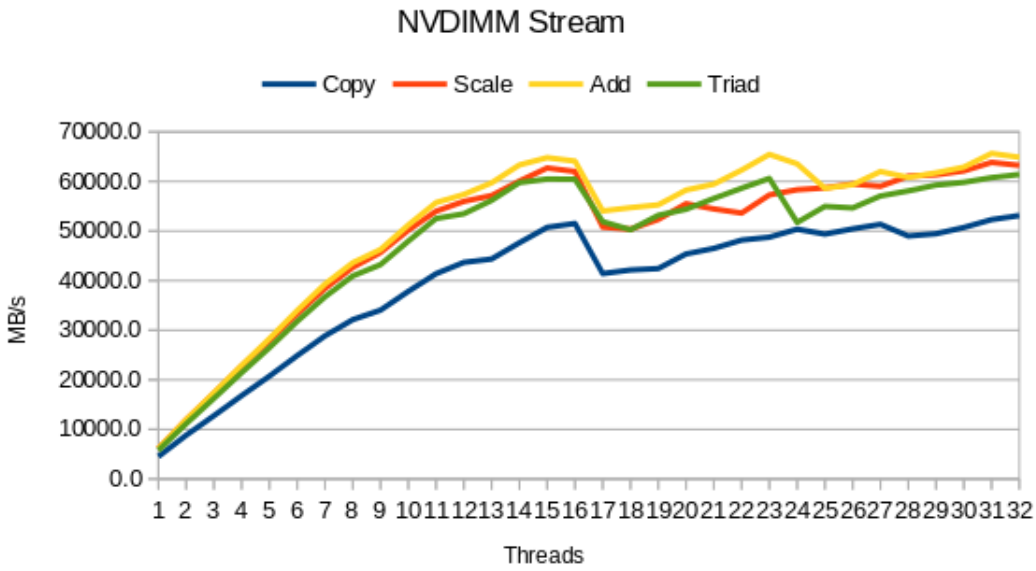

Figure 2: NVDIMM Stream

1.4 benchmark 3

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. There might

1.4.1 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 3: Creation of DRAM and NVDIMM arrays

```
1  if(thread_id < totalThreads-nvmThreads){
2    //From DRAM to DRAM
3    drm_read_array =
         (double*)malloc(ARRAY_LENGTH*sizeof(double));
4    drm_write_array =
         (double*)malloc(ARRAY_LENGTH*sizeof(double));
5    #pragma omp critical
6    {
7      for(i=0;i<ARRAY_LENGTH;i++){
8        drm_read_array[i] =
             ((double)rand()/(double)(RAND_MAX));
9        drm_write_array[i] =
             ((double)rand()/(double)(RAND_MAX));
10     }
11   }
12 }
```

```
13 else if(thread_id >= totalThreads-nvmThreads){
14   //From NVDIMM to NVDIMM
15   POBJ_ALLOC(pop, &nvm_read_array, double, sizeof(double)
        * ARRAY_LENGTH, NULL, NULL);
16   POBJ_ALLOC(pop, &nvm_write_array, double, sizeof(double)
        * ARRAY_LENGTH, NULL, NULL);
17   #pragma omp critical
18   {
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 4: 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        //Time stop.
21        test_time[thread_id][i] = mysecond2() -
              test_time[thread_id][i];
22      }
23  }
```

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.

1.4.2 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.

	NVM-NVM					
	DRAM					
Nvm-threads	1	1-20	21-40	41-60	61-80	81-100
1	33979.84	64179.89	64788.61	64951.02	64650.49	64447.29
2	33153.33	60854.29	62114.14	61912.29	61761.77	61872.50
3	32879.35	57643.15	58614.63	58759.60	58625.89	58423.27
4	26839.12	54508.48	55674.22	55682.26	55415.60	55367.41
5	25687.57	51076.28	52241.04	52083.85	51964.28	51955.32
6	24209.41	47721.43	48861.96	48786.82	48552.39	48656.74
7	22425.14	43819.63	45050.64	44975.15	44882.40	44672.94
8	20117.60	40489.02	41752.05	41501.44	41524.94	41286.00
9	17713.80	36687.45	37360.21	37385.79	37156.23	37232.82
10	15212.97	32731.79	33432.15	33379.32	33220.57	33296.06
11	12941.82	28177.13	28767.32	28814.29	28732.10	28833.90
12	9894.49	23814.89	24401.20	24408.00	24416.73	24525.18
13	7341.90	17595.41	18030.92	18359.65	17857.03	18532.65
14	4758.41	12750.63	13284.93	13351.80	13790.00	13569.13
15	2394.31	6834.38	6992.16	7515.33	7445.93	7353.06
	NVM					
Nvm-threads	1	1-20	21-40	41-60	61-80	81-100
1	5114.00	3289.73	3253.33	3250.62	3250.58	3248.86
2	10235.69	6599.16	6520.83	6507.40	6520.58	6500.57
3	15165.52	10119.48	9983.90	9984.04	9966.19	9979.90
4	20114.02	13554.15	13438.86	13435.23	13423.07	13416.59
5	24936.64	17103.23	16991.10	16958.34	16961.03	16933.81
6	29623.07	20662.40	20464.90	20437.07	20468.27	20438.51
7	33972.44	22367.64	22014.43	21953.45	21980.77	22105.95
8	37897.59	27397.05	27264.10	27141.45	27219.06	27083.62
9	41598.76	30795.78	30818.58	30922.06	30851.27	30427.67
10	44808.35	34539.87	34494.76	34500.89	34350.06	34203.29
11	48545.49	37723.50	37394.55	37303.10	37247.85	36857.64
12	51215.90	41988.10	41845.44	41750.50	41731.61	41533.89
13	54109.52	44058.49	43833.39	43794.52	43761.88	42853.98
14	57066.38	48613.84	48218.46	48161.35	48318.21	47953.55
15	58853.11	51816.83	51824.03	51488.10	51789.92	51100.06

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

	SUM					
Nvm-threads	1	1-20	21-40	41-60	61-80	81-100
1	39093.84	67469.62	68041.93	68201.64	67901.07	67696.15
2	43389.02	67453.45	68634.96	68419.69	68282.35	68373.06
3	48044.87	67762.63	68598.53	68743.64	68592.08	68403.17
4	46953.14	68062.63	69113.08	69117.49	68838.67	68784.00
5	50624.21	68179.51	69232.14	69042.18	68925.31	68889.13
6	53832.48	68383.83	69326.86	69223.88	69020.66	69095.26
7	56397.58	66187.27	67065.07	66928.60	66863.17	66778.89
8	58015.19	67886.07	69016.15	68642.89	68744.00	68369.63
9	59312.56	67483.22	68178.79	68307.84	68007.50	67660.49
10	60021.32	67271.67	67926.91	67880.21	67570.63	67499.35
11	61487.31	65900.63	66161.87	66117.39	65979.96	65691.54
12	61110.39	65802.98	66246.64	66158.50	66148.34	66059.07
13	61451.42	61653.90	61864.31	62154.17	61618.91	61386.63
14	61824.79	61364.48	61503.39	61513.15	62108.21	61522.68
15	61247.42	58651.21	58816.19	59003.42	59235.85	58453.12

Figure 4: NVM-NVM 1-100 iteration, 16 threads total, 3rd version

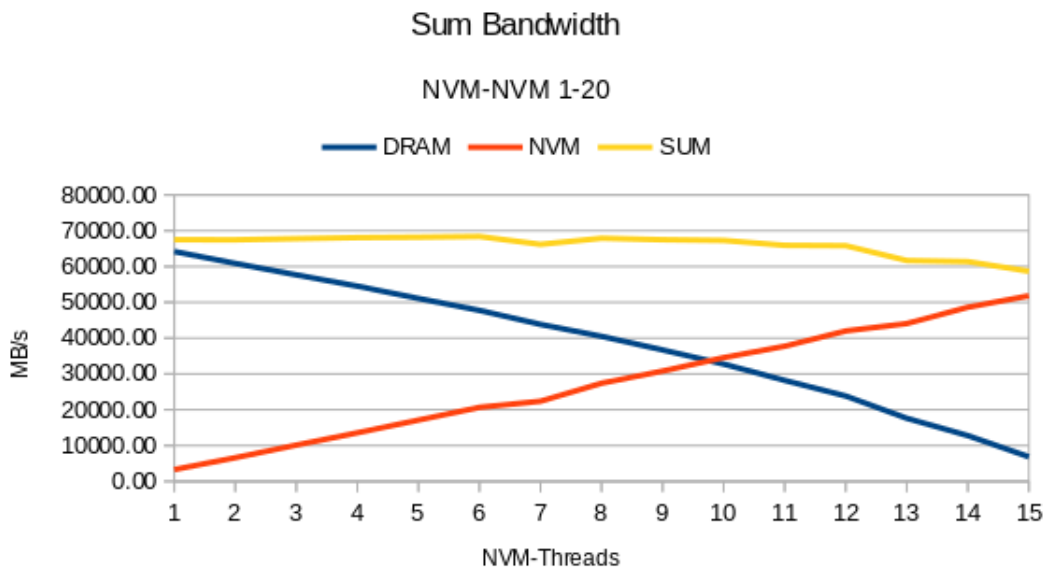

Figure 5: NVM-NVM graph 1-20, 3rd version

1.4.3 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.

	NVM-DRAM					
	DRAM					
Nvm-threads	1	1-20	21-40	41-60	61-80	81-100
1	32437.33	51729.43	53490.40	54419.38	56626.69	57295.70
2	31785.84	58694.61	59123.37	59135.90	58803.95	59160.35
3	27089.46	55323.72	55359.79	55449.94	55124.84	55079.11
4	26523.71	51565.09	51498.67	51032.22	50944.45	51155.26
5	23943.39	47390.53	47067.10	47291.86	46960.13	47119.51
6	21332.19	43205.17	43003.44	43143.48	42653.83	42962.85
7	18079.52	38801.31	38958.62	38995.27	38681.43	39000.71
8	18219.63	34652.20	34829.29	34582.21	34809.29	34763.03
9	15716.34	30244.85	30604.89	30504.82	30347.98	30531.90
10	12959.49	25984.54	26132.58	26116.43	26282.85	25981.46
11	11550.12	21756.54	21772.80	21935.54	21740.99	21776.18
12	8936.14	17175.26	17608.93	17216.47	17843.56	17341.46
13	6236.36	12779.91	13194.20	13003.21	13181.25	13119.76
14	3603.24	8669.61	8644.75	8731.08	8916.27	8621.84
15	1793.63	4250.55	4490.40	4267.18	4502.44	4260.46
	NVM					
Nvm-threads	1	1-20	21-40	41-60	61-80	81-100
1	2170.83	3635.35	3940.80	4127.15	4313.01	4234.35
2	4446.39	7927.66	7856.40	7977.52	7974.72	7800.26
3	6524.15	12018.48	11973.65	11978.96	11956.69	11868.26
4	8968.17	16245.16	16145.53	16224.22	16169.83	15979.78
5	11227.93	20004.18	20569.08	20226.56	20499.64	20114.15
6	13267.48	24362.23	24984.69	24557.47	24613.37	24477.62
7	15471.30	28739.42	29026.57	28754.67	28980.22	28589.24
8	18094.36	32833.01	33428.60	33380.75	33069.79	33033.76
9	20805.37	37368.77	37715.52	37628.86	37511.17	37290.54
10	22740.89	41319.93	41966.16	42139.10	42023.48	41966.67
11	25533.28	45907.61	46625.38	46449.12	46306.21	46181.20
12	28124.41	50396.18	50722.28	51065.63	50556.47	50735.26
13	30225.94	54592.24	55465.44	55422.83	55294.43	55153.29
14	32659.06	58440.82	59868.63	59725.88	59542.50	59570.12
15	35484.88	63168.26	64013.14	64235.39	63940.74	64110.34

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

	SUM					
Nvm-threads	1	1-20	21-40	41-60	61-80	81-100
1	34608.16	55364.78	57431.19	58546.53	60939.70	61530.05
2	36232.23	66622.28	66979.77	67113.41	66778.68	66960.61
3	33613.61	67342.20	67333.44	67428.90	67081.53	66947.36
4	35491.88	67810.25	67644.20	67256.43	67114.28	67135.04
5	35171.32	67394.71	67636.19	67518.42	67459.77	67233.66
6	34599.67	67567.40	67988.12	67700.95	67267.20	67440.47
7	33550.82	67540.73	67985.19	67749.94	67661.65	67589.95
8	36313.99	67485.21	68257.89	67962.96	67879.08	67796.79
9	36521.71	67613.62	68320.41	68133.69	67859.15	67822.43
10	35700.38	67304.47	68098.74	68255.52	68306.33	67948.13
11	37083.40	67664.15	68398.18	68384.66	68047.19	67957.39
12	37060.55	67571.44	68331.21	68282.10	68400.02	68076.72
13	36462.30	67372.15	68659.64	68426.04	68475.68	68273.05
14	36262.30	67110.42	68513.38	68456.96	68458.77	68191.96
15	37278.51	67418.81	68503.54	68502.57	68443.18	68370.81

Figure 7: NVM-DRAM 1-100 iteration, 3rd version

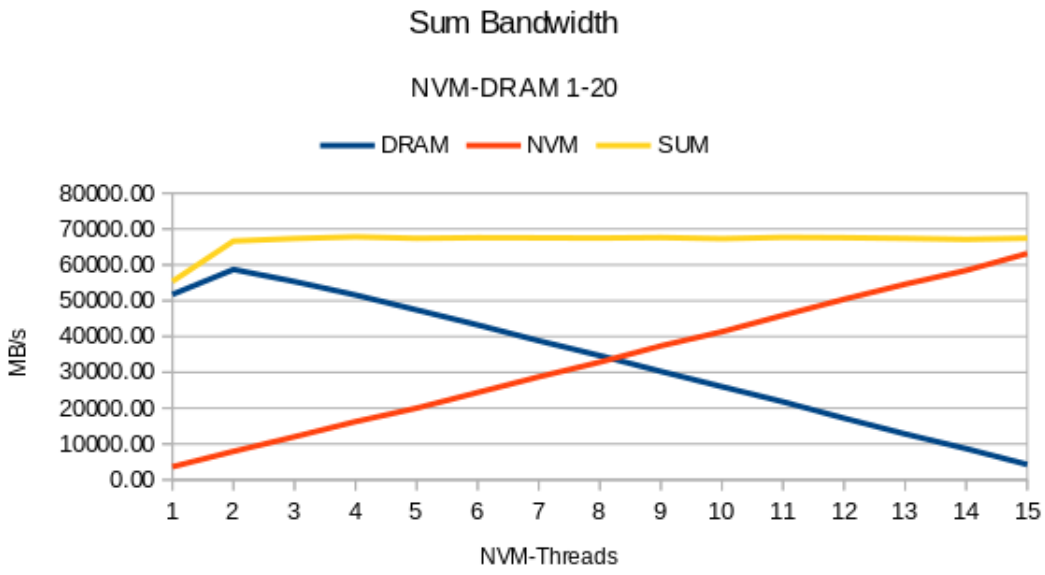

Figure 8: NVM-DRAM graph 1-20, 3rd version

1.4.4 DRAM-NVM

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

	DRAM-NVM					
	DRAM					
Nvm-threads	1	1-20	21-40	41-60	61-80	81-100
1	18762.63	62678.30	63604.73	63196.58	63136.35	63599.61
2	27382.09	58321.86	59823.75	59850.20	59792.36	59717.87
3	25055.31	55650.92	55734.25	56127.18	55630.90	55935.54
4	24531.82	51279.78	52005.76	52282.87	51782.02	52177.37
5	23933.44	46555.25	48391.69	47941.57	48093.38	47839.01
6	18038.64	43432.14	44116.64	44225.90	43685.08	44094.70
7	16733.09	38827.55	39930.92	39448.07	39906.00	39483.39
8	14948.52	35285.18	35922.06	35737.11	35492.87	35796.48
9	12648.41	31228.15	31784.53	31492.60	31380.92	31454.26
10	11685.27	26830.40	27658.00	27610.40	27050.44	27021.04
11	9125.40	22445.38	23500.71	22790.09	22663.17	22910.98
12	6988.07	18442.56	18641.75	18849.18	18429.07	18354.52
13	5241.80	13668.18	14290.47	14034.79	14232.71	13744.87
14	3489.73	9087.65	9672.94	9461.29	9476.61	9276.00
15	1706.71	4273.06	4665.14	4342.46	4592.38	4610.87
	NVM					
Nvm-threads	1	1-20	21-40	41-60	61-80	81-100
1	3375.57	4094.13	3968.66	4013.27	4024.96	3937.43
2	5969.30	8738.95	8141.12	7939.10	8120.81	7879.02
3	14655.50	12710.47	12200.25	12147.26	12139.38	12124.44
4	12183.05	16846.06	16447.48	16153.87	16528.34	16083.88
5	15414.28	21516.95	20524.82	20689.27	20457.96	20526.30
6	23350.53	25835.47	24888.70	24740.28	25046.00	24538.40
7	24810.32	30042.63	29430.07	29410.65	29124.31	29382.89
8	25246.16	34903.09	33344.27	33607.80	33385.67	33247.98
9	29498.49	38826.97	37916.57	38188.18	37813.69	37219.56
10	36784.96	43563.46	42352.29	42525.81	42036.90	42070.58
11	39799.65	47967.83	46875.74	47224.03	46257.64	46424.18
12	38783.93	52752.69	51908.85	51408.20	51339.13	50338.82
13	43481.73	56883.56	56624.74	55455.89	55879.32	56058.91
14	49369.10	61602.79	60907.89	60678.79	60065.98	60171.88
15	49133.33	67153.54	66972.79	63557.38	63874.03	63521.86

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

	SUM					
Nvm-threads	1	1-20	21-40	41-60	61-80	81-100
1	22138.20	66772.43	67573.40	67209.85	67161.31	67537.04
2	33351.39	67060.81	67964.86	67789.29	67913.17	67596.89
3	39710.81	68361.39	67934.50	68274.44	67770.28	68059.98
4	36714.87	68125.85	68453.24	68436.74	68310.35	68261.24
5	39347.72	68072.20	68916.50	68630.84	68551.33	68365.32
6	41389.17	69267.61	69005.34	68966.18	68731.08	68633.10
7	41543.41	68870.19	69360.99	68858.72	69030.31	68866.28
8	40194.68	70188.28	69266.33	69344.90	68878.54	69044.46
9	42146.90	70055.12	69701.09	69680.78	69194.61	68673.83
10	48470.23	70393.86	70010.29	70136.21	69087.34	69091.62
11	48925.05	70413.21	70376.45	70014.13	68920.81	69335.17
12	45772.00	71195.25	70550.60	70257.38	69768.20	68693.34
13	48723.53	70551.74	70915.22	69490.69	70112.04	69803.78
14	52858.83	70690.44	70580.83	70140.07	69542.59	69447.88
15	50840.04	71426.60	71637.93	67899.84	68466.41	68132.72

Figure 10: DRAM-NVM 1-100 iteration, 3rd version

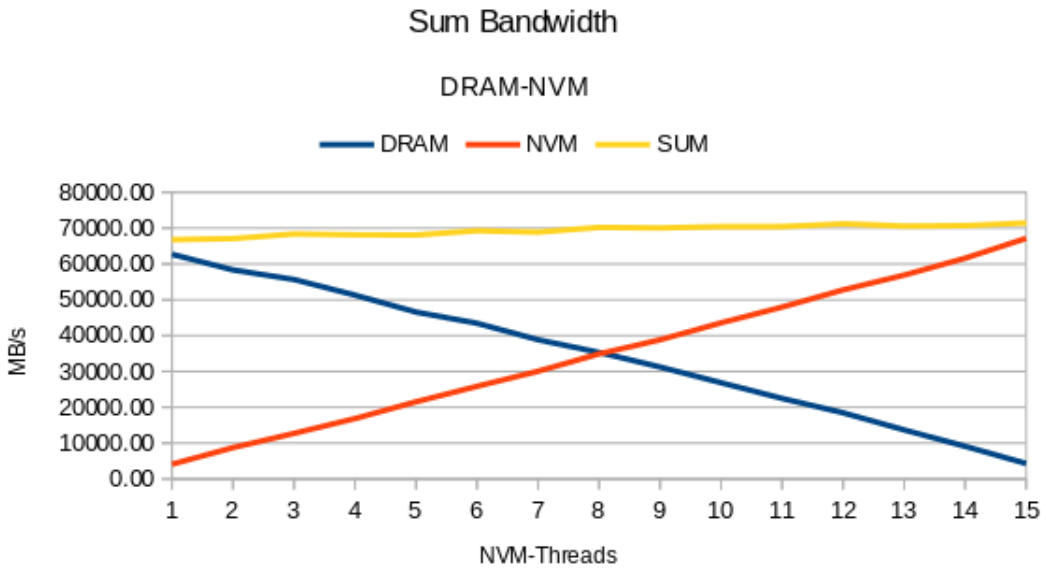

Figure 11: DRAM-NVM graph 1-20, 3rd version

References

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