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 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.

3 Coding with NVDIMM

3.1 Creating pmempool

Before NVDIMM can be used, the user must create whats called a memory pool on the NVDIMM. The NVDIMM has several modes, in order to be able to create a memory pool the mode must be set to fsdax. On a server this must be done by the system administrator. To see what mode the NVDIMM is in can be done by the command ndctl-list. A program called pmempool must also be installed on the server, it is this program that will create the memory pool. The command used for creating the memory pool for this thesis is

```
1  pmempool create --layout my_layout --size=50G obj pool.obj
```

The layout is a string stored in the memory pool. When a program access a memory pool it need to send a string that match the string in the memory pool in order to use it. The user can specify the size of the memory pool, if size is not specified the pmempool will create a pool

with the lowest size allowed. There are three different types of memory pool to choose from, they are obj, log and blk. Which type of memory pool to use depends on which type of library is used in the program. In this thesis the libpmemobj library was used and that is why obj was used in the creating of memory pool. For log and blk are for the libraries libpmemlog and libpmemblk. The last part of the command line is the name and file address of the memory pool.

3.2 Different types of libraries

The libpmemobj library
libpmemlog
libpmemblk

3.3 Opening the memory pool

4 Benchmarks

There are four different benchmarks.

4.1 STREAM DRAM

The stream DRAM benchmark measure the memory speed of the DRAM. The benchmark uses the STREAM[1] benchmark without any changes in order to measure how fast the memory speed is. The benchmark run the test 32 times and only on one socket, every times it restart one extra thread is added. 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.

4.2 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

Figure 1: DRAM Stream

```
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,
           0666);
```

3

```
13    if (pop == NULL)
14        pop = pmemobj_open(path, LAYOUT_NAME);
15
16        if (pop == NULL) {
17            perror(path);
18            exit(1);
19        }
20  }
```
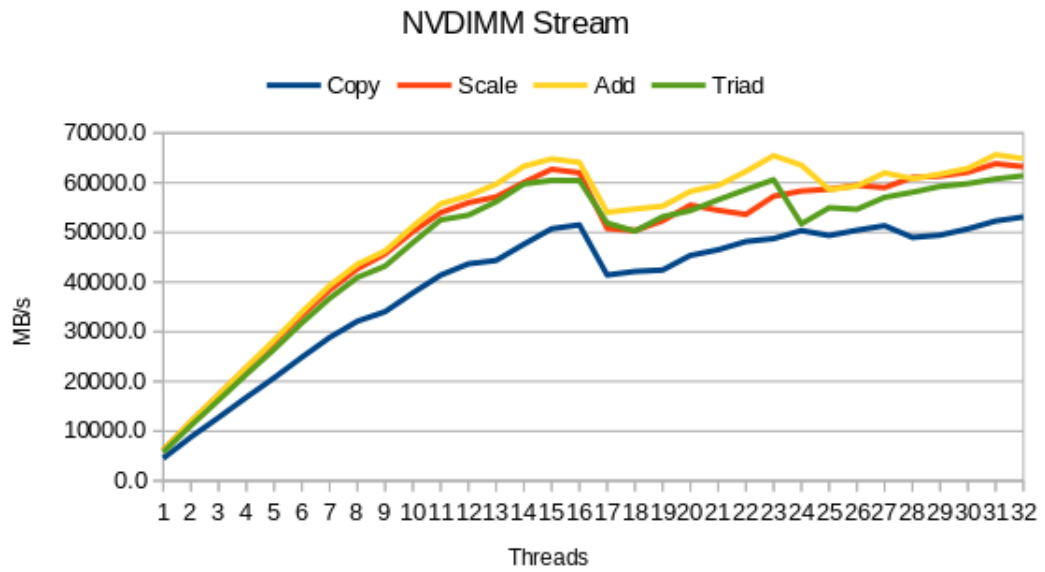

4.3 benchmark 3

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.

4.3.1 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

```
if(thread_id < totalThreads-nvmThreads){
    //From DRAM to DRAM
    drm_read_array =
        (double*)malloc(ARRAY_LENGTH*sizeof(double));
    drm_write_array =
        (double*)malloc(ARRAY_LENGTH*sizeof(double));
    #pragma omp critical
    {
        for(i=0;i<ARRAY_LENGTH;i++){
            drm_read_array[i] =
                ((double)rand()/(double)(RAND_MAX));
            drm_write_array[i] =
                ((double)rand()/(double)(RAND_MAX));
        }
    }
}
else if(thread_id >= totalThreads-nvmThreads){
    //From NVDIMM to NVDIMM
    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 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    //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.

4.3.2 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 2: NVM-NVM 1-100 iteration, 16 threads total, 3rd version

Nvm-threads	SUM					
	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

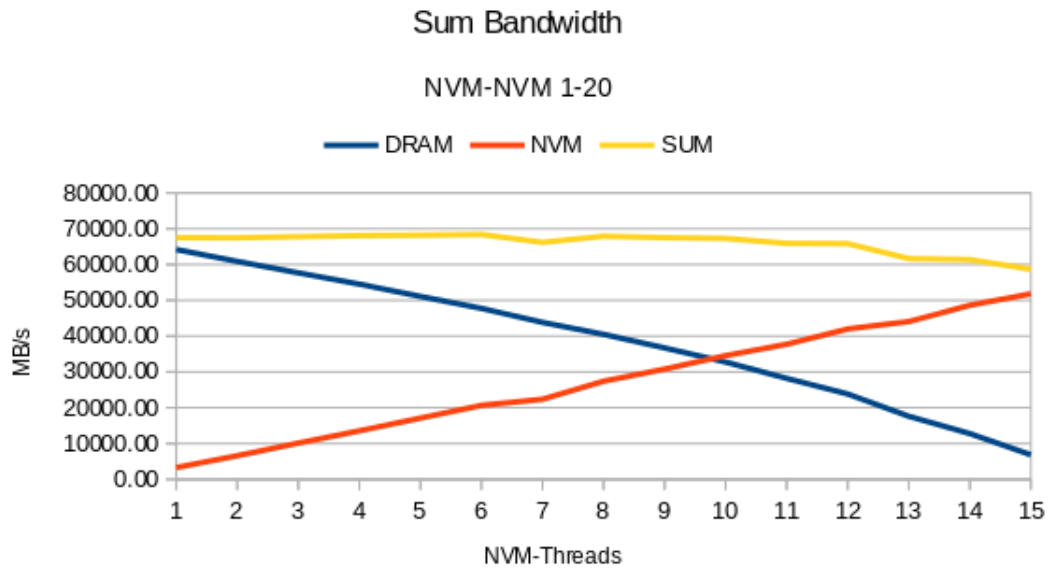Figure 3: NVM-NVM 1-100 iteration, 16 threads total, 3rd version

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

4.3.3 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 5: 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 6: NVM-DRAM 1-100 iteration, 3rd version
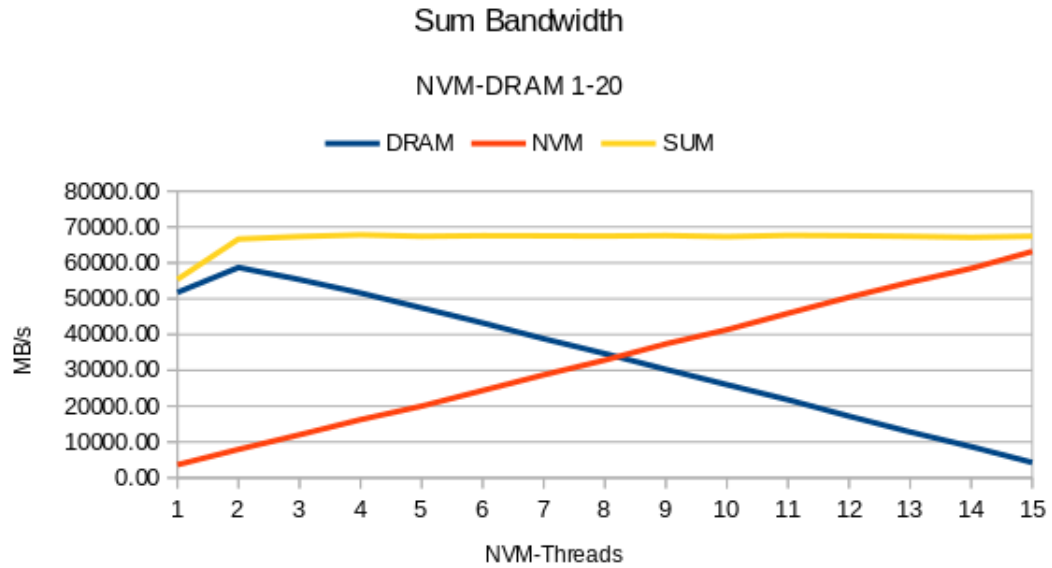
Figure 7: NVM-DRAM graph 1-20, 3rd version

4.3.4 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 8: 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 9: DRAM-NVM 1-100 iteration, 3rd version
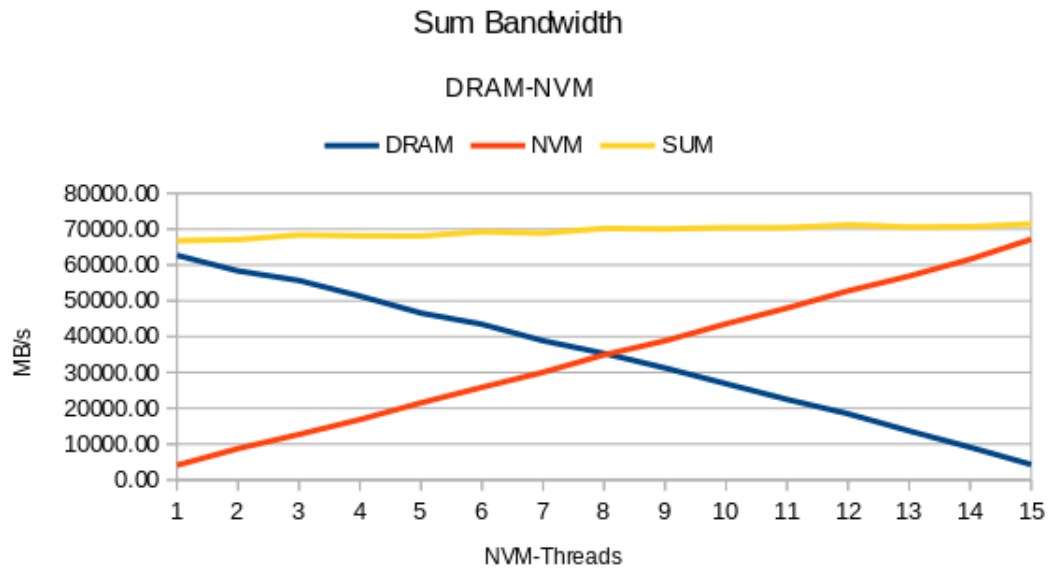
Figure 10: DRAM-NVM graph 1-20, 3rd version

5 Simulation

5.1 DRAM only

Threads	Total time	Iteration time	Analyze time
1	331.08	203.54	127.54
2	172.55	105.99	66.56
3	117.41	72.53	44.88
4	89.79	55.41	34.38
5	71.93	44.57	27.36
6	60.55	37.63	22.93
7	52.62	32.80	19.82
8	48.40	30.04	18.36
9	44.11	27.64	16.47
10	41.41	26.38	15.04
11	39.42	25.63	13.79
12	38.04	25.19	12.85
13	36.66	24.71	11.95
14	35.86	24.74	11.12
15	35.23	24.78	10.45
16	34.76	24.89	9.87

Table 1: Simulation with only DRAM

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 singel so there are no reason to be worried if the mysecond-method is thead-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
```

14

```
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    }
```

5.1.1   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 arrray 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*4'000'000+2*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  }
```

Calculation of	Data generation				
		Factor	MB	repeated	total
n	4,000,000.00	3	96	5000	480,000.00
edges	31,976,004.00	2	511.6	5000	2,558,080.32
		sum	607.6		3,038,080.32
	Benchmark	Predicted	Data generated		
DRAM-thread	DRAM speed	time	time measurement		
1	12255.3	247.90	203.538163		
2	23792.2	127.69	105.991421		
3	34637.4	87.71	72.526187		
4	45202.7	67.21	55.408966		
5	52089.8	58.32	44.569655		
6	55372.6	54.87	37.625648		
7	56527	53.75	32.802805		
8	59893.5	50.72	30.038089		
9	62237.5	48.81	27.636179		
10	63772.9	47.64	26.379294		
11	64766.9	46.91	25.62663		
12	65396.8	46.46	25.194585		
13	66093.7	45.97	24.70847		
14	66283	45.83	24.740348		
15	66351.8	45.79	24.775877		
16	66332.1	45.80	24.885608		

Table 2: Prediction of time taken for data generation

5.1.2 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  //Analyse 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 }
```

	copy	scale	add	triad	Average
1	13072.2	14053.7	13810.8	13821.3	6652.3
2	24586.2	25448.2	26622	25639.9	12920.8
3	35144.7	36424.1	38554.9	38323.4	18936.7
4	45261.6	46928	49559	49473.3	25020.5
5	52061.5	52960.9	58313.3	58783.2	31049.9
6	55459.1	55960	62524.6	62896.5	37123.1
7	57928.9	58281.5	65642	66144.9	42689.6
8	59714.4	60302.5	68166.4	68608.6	47295.1
9	61802.5	62368.3	70859.2	71188.9	50991.5
10	63482.1	63966.8	72520.9	72851	56010.9
11	64636.5	65069.6	73669.4	73741.7	61519.2
12	65450.4	65659.8	74202.1	74163.3	64427.4
13	66145.8	66209.1	74432.5	74503.6	66549.8
14	66337.4	66343.3	74314.9	74386.3	71372.7
15	66439.8	66274.5	74195	74232.2	76140.7
16	66417.5	66115.8	74067.2	74154.5	80702.4

Table 3: New Stream benchmark

	copy	scale	add	triad	Average
16	66666.2	66353.8	74338.0	74414.4	80628.7
15	66660.9	66491.8	74481.0	74467.2	75901.3
14	66694.0	66559.1	74629.0	74584.7	70941.1
13	66340.6	66368.2	74727.0	74650.0	66280.4
12	65713.8	65751.7	74096.7	74255.2	64289.1
11	65022.3	65194.7	73898.1	74032.9	61050.2
10	63757.8	63951.0	72837.8	72917.5	55831.0
9	62273.9	62480.3	71330.7	71492.3	50349.5
8	60062.3	60373.0	68207.0	68511.1	46814.0
7	58049.1	58420.0	66026.9	66531.4	42330.4
6	55561.5	55895.2	62483.8	62977.5	36584.7
5	52385.0	52877.0	58354.5	58669.5	30581.0
4	45432.5	46682.5	49615.4	49386.9	24665.5
3	35123.2	36222.2	38081.9	37902.2	18702.4
2	24126.9	24996.5	26259.4	25715.4	12920.5
1	12447.7	12887.6	13596.3	13613.0	6611.8

Table 4: Stream benchmark after modification, 16-1 threads

Calculation For analyze		Factor	MB	repeated	total MB
n	4,000,000	5	160	5,000	800,000.00
	Benchmark		Analyze		Analyze time
DRAM-thread	DRAM speed		Prediction		Measurement
1	6895.4		116.02		127.54
2	13231.7		60.46		66.56
3	19502.7		41.02		44.88
4	25413.1		31.48		34.38
5	31550.7		25.36		27.36
6	37725.3		21.21		22.93
7	43686.7		18.31		19.82
8	48440.8		16.52		18.36
9	51649.2		15.49		16.47
10	57225		13.98		15.04
11	62227.7		12.86		13.79
12	66044.2		12.11		12.85
13	67842.9		11.79		11.95
14	72694.7		11.00		11.12
15	77677.7		10.30		10.45
16	82705.5		9.67		9.87

Table 5: DRAM analyze with stream average benchmark

Calculation For analyze		Factor	MB	repeated	total MB
n	4,000,000	5	160	5,000	800,000.00
	Benchmark		Analyze		Analyze time
DRAM-thread	DRAM speed		Prediction		Measurement
1	12255.3		65.28		127.54
2	23792.2		33.62		66.56
3	34637.4		23.10		44.88
4	45202.7		17.70		34.38
5	52089.8		15.36		27.36
6	55372.6		14.45		22.93
7	56527		14.15		19.82
8	59893.5		13.36		18.36
9	62237.5		12.85		16.47
10	63772.9		12.54		15.04
11	64766.9		12.35		13.79
12	65396.8		12.23		12.85
13	66093.7		12.10		11.95
14	66283		12.07		11.12
15	66351.8		12.06		10.45
16	66332.1		12.06		9.87

Table 6: Prediction of time taken for analyzing the data

5.1.3 Analyze only

Here I have separated the code to see if the code will run faster.

Calculation For analyze					
		Factor	MB	repeated	total MB
n	4,000,000.00	5	160	5000	800,000
	Benchmark		Analyze		Analyze time
DRAM-thread	DRAM speed		Prediction		Measurement
1	12255.3		65.28		121.664498
2	23792.2		33.62		62.471152
3	34637.4		23.10		42.580493
4	45202.7		17.70		32.5764
5	52089.8		15.36		26.249728
6	55372.6		14.45		22.270504
7	56527		14.15		19.418105
8	59893.5		13.36		17.636347
9	62237.5		12.85		15.957231
10	63772.9		12.54		14.607838
11	64766.9		12.35		13.442421
12	65396.8		12.23		12.530874
13	66093.7		12.10		11.989188
14	66283		12.07		11.171952
15	66351.8		12.06		10.272031
16	66332.1		12.06		9.678643

Table 7: Prediction of time taken for analyzing the data, with a code that only analyze data

5.2   With NVM

Simulation									
1	Total threads								
2	Data Generation Threads								
3	Analyze threads								
4	Data Generation time								
5	Data Generation idle time								
6	Transfer time								
7	Transfer idle time								
8	DRAM-NVM transfer time								
9	Analyze time								
10	Total time								
1	2	3	4	5	6	7	8	9	10
16	15	1	247.67	222.32	247.71	0.01	40.18	207.50	247.71
16	14	2	138.73	112.50	138.75	0.02	20.66	118.05	138.75
16	13	3	98.85	72.16	98.87	0.01	17.31	81.54	98.87
16	12	4	75.21	47.62	75.23	0.01	13.09	62.12	75.23
16	11	5	60.83	32.40	60.84	0.01	10.63	50.20	60.84
16	10	6	51.26	20.98	51.27	0.01	9.21	42.05	51.27
16	9	7	45.28	13.86	45.29	0.04	8.21	37.04	45.29
16	8	8	40.79	6.61	40.80	0.08	7.55	33.15	40.80
16	7	9	40.79	0.69	40.80	4.37	7.03	29.39	40.80
16	6	10	43.71	0.02	43.72	12.04	6.09	25.58	43.72
16	5	11	50.03	0.02	50.04	20.81	5.65	23.56	50.04
16	4	12	61.02	0.02	61.03	35.00	4.99	21.03	61.03
16	3	13	78.70	0.02	78.71	55.42	4.47	18.80	78.71
16	2	14	113.37	0.01	113.38	90.80	4.76	17.80	113.38
16	1	15	207.54	0.00	207.55	186.93	4.38	16.23	207.55

Table 8: Simulation with both NVDIMM and DRAM

Calculation					
		Factor	repeated	total	total MB
n	4,000,000.00	4	5000	80,000,000,000.00	640,000.00
edges	31,976,004.00	1.5	5000	239,820,030,000.00	1,918,560.24
	Benchmark	Predicted	Data generated		
Nvm-threads	DRAM speed	time	time measurement		
1	63599.61	40.23	25.35		
2	59717.87	42.84	26.22		
3	55935.54	45.74	26.70		
4	52177.37	49.04	27.59		
5	47839.01	53.48	28.42		
6	44094.70	58.02	30.28		
7	39483.39	64.80	31.43		
8	35796.48	71.48	34.17		
9	31454.26	81.34	40.10		
10	27021.04	94.69	43.69		
11	22910.98	111.67	50.01		
12	18354.52	139.40	61.00		
13	13744.87	186.15	78.69		
14	9276.00	275.83	113.36		
15	4610.87	554.90	207.54		

Table 9: Time prediction, data generation

Calculation For analyze					
		Factor	repeated	total	total MB
n	4,000,000.00	5	5000	100,000,000,000.00	800,000.00
DRAM-NVM	4,000,000.00	1	5000	20,000,000,000.00	160,000.00
	Benchmark	DRAM-NVM	Analyze	DRAM-NVM	Analyze time
Nvm-threads	NVM speed	Predicted	Prediction	transfer time	Measurement
1	3937.43	40.64	203.18	40.18	207.50
2	7879.02	20.31	101.54	20.66	118.05
3	12124.44	13.20	65.98	17.31	81.54
4	16083.88	9.95	49.74	13.09	62.12
5	20526.30	7.79	38.97	10.63	50.20
6	24538.40	6.52	32.60	9.21	42.05
7	29382.89	5.45	27.23	8.21	37.04
8	33247.98	4.81	24.06	7.55	33.15
9	37219.56	4.30	21.49	7.03	29.39
10	42070.58	3.80	19.02	6.09	25.58
11	46424.18	3.45	17.23	5.65	23.56
12	50338.82	3.18	15.89	4.99	21.03
13	56058.91	2.85	14.27	4.47	18.80
14	60171.88	2.66	13.30	4.76	17.80
15	63521.86	2.52	12.59	4.38	16.23

Table 10: Time prediction, transfer and analyze

References

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