1 Simulation

1.1 DRAM only

Threads	Total Time	Iteration Time	Calculation Time
1	1261.69	779.77	481.92
2	658.48	411.28	247.20
3	447.77	278.46	169.31
4	348.40	219.89	128.52
5	282.04	178.16	103.88
6	237.25	148.97	88.29
7	212.72	135.14	77.58
8	189.07	120.76	68.31
9	176.45	112.84	63.62
10	165.79	108.61	57.17
11	158.76	105.66	53.10
12	153.55	103.31	50.24
13	148.73	100.92	47.81
14	144.26	99.55	44.71
15	141.09	99.16	41.93
16	139.71	99.81	39.90

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 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 1: 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 1.1.1
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 1.1.2
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 2: 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   }
```

1.1.1 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 3: 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					
		Factor	repeated	total	total MB
n	16,000,000	2.5	5000	200,000,000,000	1,600,000.00
edges	127,952,004	1.5	5000	959,640,030,000	7,677,120.24
	Benchmark	Predicted	Data generated		
Threads	DRAM speed	time	time measurement		
1	13265.2	699.36	747.35		
2	26431.7	350.98	399.27		
3	38557.2	240.61	275.08		
4	49447.8	187.61	211.63		
5	58565.4	158.41	171.95		
6	62627.3	148.13	146.63		
7	66290.8	139.95	129.06		
8	69130.7	134.20	118.63		
9	71409.6	129.91	111.71		
10	73271.5	126.61	107.15		
11	74191.2	125.04	103.95		
12	74211.4	125.01	102.13		
13	74659.4	124.26	100.01		
14	74715.4	124.17	98.89		
15	74486	124.55	98.31		
16	74328.7	124.81	98.45		

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

1.1.2 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 4: 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 }
```

Calculation For analyze					
		Factor	MB	repeated	total MB
n	16,000,000.00	5	640	5000	3,200,000
	Benchmark		Analyze		Analyze time
Threads	DRAM speed		Prediction		Measurement
1	6665.4		480.09		470.279564
2	13183.1		242.74		242.670301
3	18704.7		171.08		167.853153
4	24971.9		128.14		128.155062
5	31169.6		102.66		104.356358
6	37259.5		85.88		87.260722
7	42771.7		74.82		75.473611
8	47211.9		67.78		68.165916
9	50706.4		63.11		63.171366
10	56235.2		56.90		56.994655
11	61401		52.12		52.156971
12	65062		49.18		49.364902
13	66645		48.02		47.953591
14	71518.7		44.74		44.625985
15	76423.3		41.87		41.718259
16	81259.4		39.38		39.382296

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

1.1.3 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.

	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 4: New Stream benchmark, DRAM

NVM sum test cpu 0-15					
threads	copy	scale	add	triad	Sum
1	6455.2	4492.9	6134.2	5451.6	5422.2
2	12606.9	8783.7	11978.3	10631.0	10423.9
3	18284.2	12732.8	17392.6	15414.4	15141.5
4	24033.7	16760.6	22869.3	20280.1	19841.8
5	29584.8	20683.6	28242.3	25052.1	24489.6
6	35355.3	24818.6	33865.8	30064.3	29347.0
7	37564.9	24631.4	29185.7	26727.6	34074.4
8	45400.6	31945.8	43520.9	38752.7	37750.0
9	48372.0	33983.1	46176.0	41174.1	40607.1
10	52953.8	37741.9	51064.9	45665.5	45129.2
11	56689.4	41441.1	55640.6	50013.6	49416.7
12	58537.3	43532.5	56954.3	51616.4	52280.9
13	59904.7	44258.9	59200.8	53467.5	53727.5
14	62678.7	47558.2	63158.1	57325.7	57790.7
15	50577.2	36717.5	49718.4	44182.5	45334.0
16	66047.5	52866.6	66134.0	60444.3	65552.6

Table 5: New Stream benchmark, NVM

1.1.4 Calculation only

Calculation					
		Factor	repeated	total	total MB
n	16,000,000	2.5	5000	200,000,000,000	1,600,000.00
edges	127,952,004	1.5	5000	959,640,030,000	7,677,120.24
	Benchmark	Predicted	Data generated		
Threads	DRAM speed	time	time measurement		
1	13265.2	699.36	727.54		
2	26431.7	350.98	383.48		
3	38557.2	240.61	268.12		
4	49447.8	187.61	208.36		
5	58565.4	158.41	168.77		
6	62627.3	148.13	146.18		
7	66290.8	139.95	126.40		
8	69130.7	134.20	116.22		
9	71409.6	129.91	110.54		
10	73271.5	126.61	105.72		
11	74191.2	125.04	102.53		
12	74211.4	125.01	101.56		
13	74659.4	124.26	99.68		
14	74715.4	124.17	98.92		
15	74486	124.55	99.10		
16	74328.7	124.81	98.76		

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

1.2 NVDIMM Analyze only

1	912.95
2	417.38
3	277.17
4	217.33
5	174.67
6	188.40
7	233.93
8	234.64
9	220.51
10	207.10
11	188.31
12	188.65
13	174.08
14	168.63
15	161.06
16	162.10

Table 7: Mearsurement of analyzations only on NVDIMM.

1.3 NVM simulation

1.3.1 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.

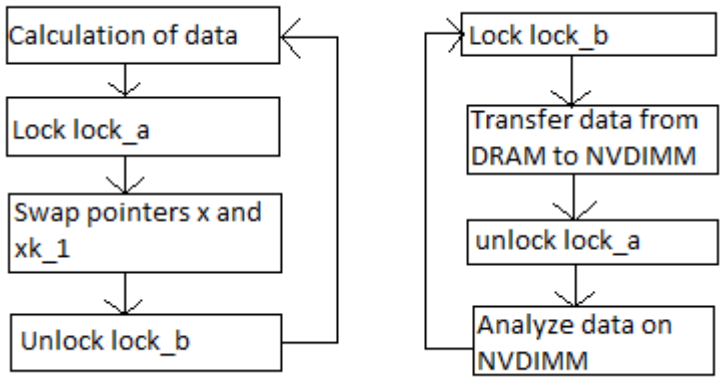

Figure 1: A simplified version of how lock works.

1.3.2 Calculation

The total time for calculation are measured before and after the while-loop. One thread will start the time measurement at line 3 and the measurement will be ended by one thread at line 33. The calculation starts with a while-loop at line 5. It will be repeated 5000 times. There is a barrier at line 6 that will synchronize all the threads. One thread will enter a single bracket a line 7 and increase the variable n by one, zero out the variables diff and Wk_1 and calculate one part of the calculation that only need to be done by one thread at line 11.

The calculation of data will happens at line 17. All threads will enter the for-loop at line 18, this for-loop will pass through all the nodes in the array. A temporary variable is created at line 20, this is to ensure that data is only stored at the end of the for-loop. Each thread will then enter a second for-loop at line 21. The arrays that begins with CRS is part of a compressed sparse matrix, that means that a 2D-array has been turned into a 1D-array in order to perform faster. The for-loop will add together edges that is connected to node i, they will be stored in the double_temp variable. Once out of the array the double_temp will be multiplied with a constant d at line 25 and added together with the Wk_product at line 27, this is a variable that was calculated by one thread in line 13. The double_temp value will be written to array at line 28.

One thead will enter the single bracket at line 36 and will start a new time measurement at line 38. This time measurement will measure how much time the calculation threads must wait for the analyze threads to finish. Once the analyze threads have unlocked lock_a the thread will pass through the lock and lock it again at line 39. It will also end the time measurement at line 40 and add the time waited to the iteration_idle_time variable. This variable represent the total amount of time the calculation threads have been idle. At line 42-44 the arrays x and xk_1 will be swapped. At line 46 the thread will unlock the lock_b, this will allow the analyze threads to do their jobs. The threads will then return to the beginning of the while-loop at line 5 and repeat the process if n is lower than 5000.

Listing 5: Calculation

```
1 #pragma omp single
2 {
3   iteration_time = mysecond();
4 }
```

```
while( n<5000 ){
  #pragma omp barrier
  #pragma omp single
  {
    n++;
    diff=0.0;
    Wk_1=0;
    //completes the first part of the formula.
    Wk_1_product = (omd + (d*Wk_1))*iN;
  }

  /* Calculation of Data */
  #pragma omp for reduction(max:diff)
  for( i=0; i<nodes; i++){
    //This is A*x^k-1
    double_temp = 0;
    for( j=CRS_row_ptr[i]; j<CRS_row_ptr[i+1]; j++){
      double_temp += CRS_values[j] * xk_1[CRS_col_idx[j]];
    }
    //d*Ax^k-1
    double_temp *= d;
    //Adding the first part and second part together.
    double_temp += Wk_1_product;
    x[i] = double_temp;

    //Comuting the difference between x^k and x^k-1
    //and adds the biggest diff to diffX[thread_id]
    if( x[i]-xk_1[i] > diff )
      diff = x[i] - xk_1[i];
  }

  #pragma omp single
  {
    temp_time = mysecond();
    omp_set_lock(&lock_a);
    iteration_idle_time += mysecond() - temp_time;

    temp_x = xk_1;
    xk_1 = x;
    x = temp_x;

    omp_unset_lock(&lock_b);
  }
```

```
48 }//end of while-loop
49 #pragma omp single
50 {
51   iteration_time = mysecond() - iteration_time;
52 }
```

1.3.3 Analyze

The time will be measured before and after the while-loop that start at line 1. The if test for this while-loop has been moved to the end of the while-loop, that is why the it test at line 1 is 1==1 and will allways be true. One thread will enter the single bracket at line 2 and start a time measurement that will measure the idle time of the analyze threads. The threads will wait until the calculation threads unlocks the lock_b. When its unlocked the thread will pass through line 5 and lock lock_b again. It will also add the time waited to the variable transfer_idle_time. The thread will then start a new time measurement at line 7, this measurement will measure the time it takes to transfer data from DRAM to NVDIMM. the variable average will be turned to zero at line 8. At line 11-14 all the threads will work together to transfer the xk_1 array from DRAM to NVDIMM. Once done one thread will enter a new single at line 15. The thread will then end the time measurement at line 17 and add the time taken to transfer the data to the variable DRAM_to_NVM_time. It will then unlock lock_a at line 18, this will allow the calculation threads to swap x and xk_1. The thread will also start a new time measurement at line 19, this measurement will measure how long it takes for the threads to analyze the data. From line 22-25 is where the data gets analyzed, in this case its just an average of all the data. The threads will synchronize at a barrier at line 26. One thread will enter the single thread at line 27 and will divide the sum of all the nodes with the number of nodes. It will then end the time measurement at line 30 and add it to the total time it takes to analyze the data. All the threads will then go through an if test at line 33. This test will always be true until the calculation threads changes the iteration_ongoing variable from 1 to 0.This will only be done when calculation threads are done with calculation and left their while-loop that was explained above.

Listing 6: Analyze

```
1 while(1==1){
```

```
    #pragma omp single
    {
      temp_time = mysecond();
      omp_set_lock(&lock_b);
      transfer_idle_time += mysecond() - temp_time;
      temp_time = mysecond();
      average=0.0;
    }
    /* Transfer of array from DRAM to NVDIMM */
    #pragma omp for
    for(i=0; i<nodes; i++){
      D_RW(nvm_values)[i]=xk_1[i];
    }
    #pragma omp single
    {
      DRAM_to_NVM_time += mysecond() - temp_time;
      omp_unset_lock(&lock_a);
      temp_time = mysecond();
    }
    /* Analyzations of data */
    #pragma omp for reduction(+ : average)
    for(i=0;i<nodes;i++){
      average += D_RO(nvm_values)[i];
    }
    #pragma omp barrier
    #pragma omp single
    {
      average /= nodes;
      Analyse_time += mysecond() - temp_time;
    }
    //if sentence for exiting while-loop.
    if(iteration_ongoing==0){
      break;
    }
}
```

Total threads	DataGen Threads	Analyze threads	DataGen Total time	DataGen time	DataGen idle time	Total Analyze time	Transfer idle time	DRAM-NVM time	Analyze time	Total Time
16	15	1	1062.43	111.72	950.71	1062.59	0.03	187.98	874.55	1062.59
16	14	2	552.65	110.52	442.12	552.77	0.04	94.15	458.55	552.77
16	13	3	402.48	122.27	280.21	402.56	0.04	69.63	332.85	402.56
16	12	4	318.89	122.86	196.03	318.94	0.03	53.51	265.37	318.94
16	11	5	249.39	134.60	114.80	249.44	0.03	43.49	205.88	249.44
16	10	6	209.85	139.75	70.09	209.90	0.03	36.53	173.31	209.90
16	9	7	197.75	137.66	60.09	197.80	0.04	34.26	163.48	197.80
16	8	8	172.14	166.51	5.62	172.17	2.87	34.66	134.63	172.17
16	7	9	168.92	168.12	0.80	168.95	9.15	34.06	125.72	168.95
16	6	10	183.17	183.06	0.12	183.21	42.11	30.59	110.49	183.21
16	5	11	213.80	213.79	0.00	213.83	95.21	26.73	91.87	213.83
16	4	12	245.78	245.78	0.00	245.81	130.60	26.05	89.14	245.81
16	3	13	327.35	327.35	0.00	327.40	223.58	23.98	79.81	327.40
16	2	14	493.42	493.42	0.00	493.46	394.71	23.41	75.31	493.46
16	1	15	890.83	890.83	0.00	890.88	784.89	25.76	80.03	890.88

Table 8: Simulation with both NVDIMM and DRAM

1.3.4 Old

Total Threads	DataGen Threads	Analyze Threads	DataGen Time	DataGen Idle Time	Transfer Idle Time	DRAM-NVM Time	Analyze Time	Total Time
16	15	1	106.48	851.99	0.04	170.84	787.73	958.63
16	14	2	107.57	396.76	0.03	86.31	418.06	504.42
16	13	3	111.85	249.67	0.03	61.27	300.26	361.58
16	12	4	137.93	300.53	0.03	167.58	270.87	438.50
16	11	5	147.57	218.52	0.03	142.66	223.43	366.14
16	10	6	162.87	158.12	0.03	130.22	190.75	321.03
16	9	7	171.85	90.88	0.03	102.66	160.03	262.76
16	8	8	199.35	51.40	0.66	95.37	154.62	250.78
16	7	9	235.15	42.02	9.91	105.50	161.31	277.18
16	6	10	235.51	31.83	35.55	93.38	138.33	267.39
16	5	11	240.33	15.70	19.51	89.37	147.13	256.06
16	4	12	288.71	0.73	73.63	83.42	132.35	289.49
16	3	13	351.75	0.00	177.44	70.45	103.91	351.81
16	2	14	464.62	0.00	338.75	48.58	77.36	464.72
16	1	15	823.71	0.00	738.35	21.70	63.65	823.76

Table 9: Simulation with both NVDIMM and DRAM

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

Predicted calculation	Load/store instructions completed	In MB	speed	seconds
15	161,347,231,158	1,290,777.85	74486.0	17.33
14	172,773,952,639	1,382,191.62	74715.4	18.50
13	185,957,951,084	1,487,663.61	74659.4	19.93
12	201,671,615,621	1,613,372.92	74211.4	21.74
11	219,943,929,111	1,759,551.43	74191.2	23.72
10	241,801,250,630	1,934,410.01	73271.5	26.40
9	268,439,799,619	2,147,518.40	71409.6	30.07
8	301,317,613,347	2,410,540.91	69130.7	34.87
7	343,673,207,234	2,749,385.66	66290.8	41.47
6	400,700,828,066	3,205,606.62	62627.3	51.19
5	480,131,664,993	3,841,053.32	58565.4	65.59
4	599,842,341,666	4,798,738.73	49447.8	97.05
3	799,776,584,260	6,398,212.67	38557.2	165.94
2	1,199,717,220,704	9,597,737.77	26431.7	363.11
1	2,399,341,092,964	19,194,728.74	13265.2	1447.00

Table 10: Predicted calculationbased on papi.

	L3 cache misses	L3 cache miss	L3 load misses	L3 load misses
threads	PAPI_L3_TCM	in MB	PAPI_L3_LDM	in MB
15	9,702,008,094	620,928.52	25,760,001	1,648.64
14	10,414,505,930	666,528.38	24,990,133	1,599.37
13	11,267,940,351	721,148.18	18,020,986	1,153.34
12	12,187,448,519	779,996.71	18,274,879	1,169.59
11	13,274,582,641	849,573.29	18,637,339	1,192.79
10	14,593,925,871	934,011.26	20,253,335	1,296.21
9	16,197,929,846	1,036,667.51	22,522,674	1,441.45
8	18,226,402,006	1,166,489.73	24,733,748	1,582.96
7	20,895,036,560	1,337,282.34	26,738,318	1,711.25
6	24,379,213,623	1,560,269.67	30,061,653	1,923.95
5	29,243,830,364	1,871,605.14	36,954,105	2,365.06
4	36,569,741,426	2,340,463.45	44,680,152	2,859.53
3	48,621,709,482	3,111,789.41	56,447,194	3,612.62
2	72,963,408,589	4,669,658.15	80,746,396	5,167.77
1	146,650,902,016	9,385,657.73	149,083,280	9,541.33

Table 11: Tried to split load and store misses. Server do not support store misses.

	arrays	size	While-loop	sum
int	6	16,000,000	5000	480,000,000,000
int	2	127,952,004	5000	1,279,520,040,000
double	1	127,952,004	5000	639,760,020,000
			sum	2,399,280,060,000

Table 12: Manually counted the number of int and double tranfers.

	PAPI_L3_TCM: 145437044007 cache misses.			
	PAPI_L3_TCW: 239627393 total cache writes.			
	Load/store	In MB	speed	seconds
	instructions completed			
1	145,676,671,400	9,323,306.97	13265.2	702.84

Table 13: Predicted calculationbased on papi with L3.

Calculation					
		Factor	repeated	total	total MB
n	16,000,000	2.5	5000	200,000,000,000.00	1,600,000.00
edges	127,952,004	1.5	5000	959,640,030,000.00	7,677,120.24
	Benchmark	Predicted	Data generated		
Nvm-threads	DRAM speed	time	time measurement		
1	74486.0	124.55	106.48		
2	74715.4	124.17	107.57		
3	74659.4	124.26	111.85		
4	74211.4	125.01	137.93		
5	74191.2	125.04	147.57		
6	73271.5	126.61	162.87		
7	71409.6	129.91	171.85		
8	69130.7	134.20	199.35		
9	66290.8	139.95	235.15		
10	62627.3	148.13	235.51		
11	58565.4	158.41	240.33		
12	49447.8	187.61	288.71		
13	38557.2	240.61	351.75		
14	26431.7	350.98	464.62		
15	13265.2	699.36	823.71		

Table 14: Time prediction, data generation

Analyze						
		Factor	repeated	total	total MB	
n	16,000,000	5	5000	400,000,000,000.00	3,200,000.00	
DRAM-NVM	16,000,000	1	5000	80,000,000,000.00	640,000.00	
	Benchmark	DRAM-NVM	DRAM-NVM	Benchmark	Analyze	Analyze time
Nvm-threads	NVM speed	Predicted	transfer time	NVM sum	Prediction	Measurement
1	3937.43	162.54	170.84	5449	587.26	787.73
2	7879.02	81.23	86.31	10216.5	313.22	418.06
3	12124.44	52.79	61.27	14921.8	214.45	300.26
4	16083.88	39.79	167.58	19562.3	163.58	270.87
5	20526.30	31.18	142.66	24142.1	132.55	223.43
6	24538.40	26.08	130.22	28837.2	110.97	190.75
7	29382.89	21.78	102.66	33421.1	95.75	160.03
8	33247.98	19.25	95.37	37228.5	85.96	154.62
9	37219.56	17.20	105.50	40080.3	79.84	161.31
10	42070.58	15.21	93.38	44540.9	71.84	138.33
11	46424.18	13.79	89.37	48703.0	65.70	147.13
12	50338.82	12.71	83.42	51520.0	62.11	132.35
13	56058.91	11.42	70.45	53418.7	59.90	103.91
14	60171.88	10.64	48.58	57326.6	55.82	77.36
15	63521.86	10.08	21.70	61542.9	52.00	63.65

Table 15: Time prediction, transfer and analyze

1.4 2D-array test

Listing 7: Kildekode

```
1 https://github.com/SveinGunnar/Master_Thesis_2020/tree/master/ArrayCopyTest
```

4000*4000		
Threads	Time	Speed
1	138.47	4621.91
2	71.90	8901.87
3	49.98	12805.97
4	37.97	16854.35
5	30.83	20759.23
6	26.20	24426.96
7	23.37	27388.78
8	22.03	29050.67
9	21.13	30290.55
10	20.28	31558.77
11	19.77	32372.93
12	19.53	32769.09
13	19.95	32072.81
14	18.87	33919.44
15	18.83	33997.04
16	18.90	33864.18
10000*10000		
Threads	Time	Speed
1	856.44	4670.51
2	445.47	8979.21
3	306.34	13057.28
4	237.02	16876.45
5	195.01	20511.92
6	165.48	24172.77
7	153.00	26143.97
8	147.82	27059.68
9	146.34	27334.50
10	137.83	29021.36
11	134.70	29695.01
12	133.70	29917.48
13	131.01	30533.18
14	127.94	31263.72
15	128.40	31153.04
16	128.13	31217.93

Table 16: 2D-Array test