Assignment # 3 : Memory Hierarchy Optimizations Report

- ## Deliverables :

<table>
<tr><td>matrix_block.cpp</td><td colspan="2">Matrix Multiplication (maximum 1100 X 1100) implementing blocking with different Block sizes and different matrix sizes.</td></tr>
<tr><td rowspan="4">Datalayout.cpp</td><td colspan="2">Sample program implementing two different data layouts.</td></tr>
<tr><td>Layout 1</td><td>Layout 2</td></tr>
<tr><td>struct Sample1{
int x;
double y[100];
};
Sample1 s[SIZE];
for(i=0;i<SIZE;i++)
s[i].x = 25;</td><td>struct Sample2{
int x[SIZE];
double y[100];
};
Sample2 s2;
for(j=0;j<SIZE;j++)
s2.x[i] = 25;</td></tr>
<tr><td></td><td></td></tr>
<tr><td>Makefile</td><td colspan="2">Creates the executables matrix_block.out and datalayout.out</td></tr>
</table>

- ## NOTE:

The programs were run on PUMBAA with below configuration:

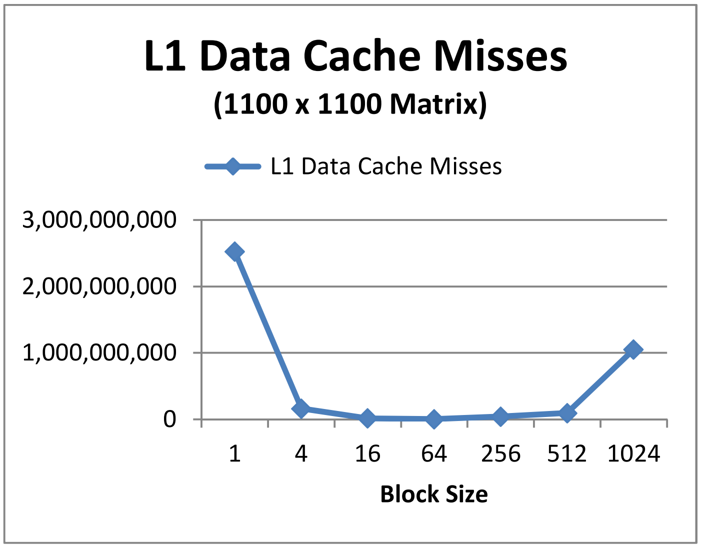
8 Core Machine:

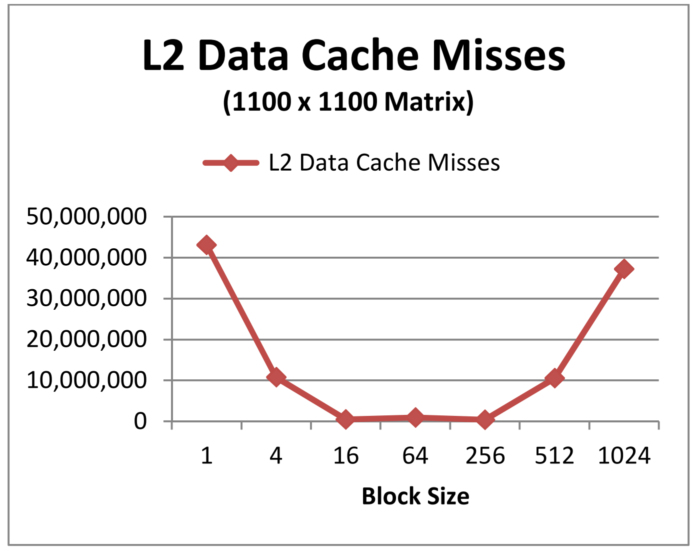
Cache line size = 64 bytes on each index (Total 4 index on each core)

- ## Analysis for Part 1:

Matrix size = 1100 x 1100

Block Size	L1 Data Cache Misses	L2 Data Cache Misses
1	2524952177	43097142
4	160412344	10747937
16	14998730	454398
64	1727098	923593
256	41885974	373443
512	94004584	10526208
1024	1050323666	37207257





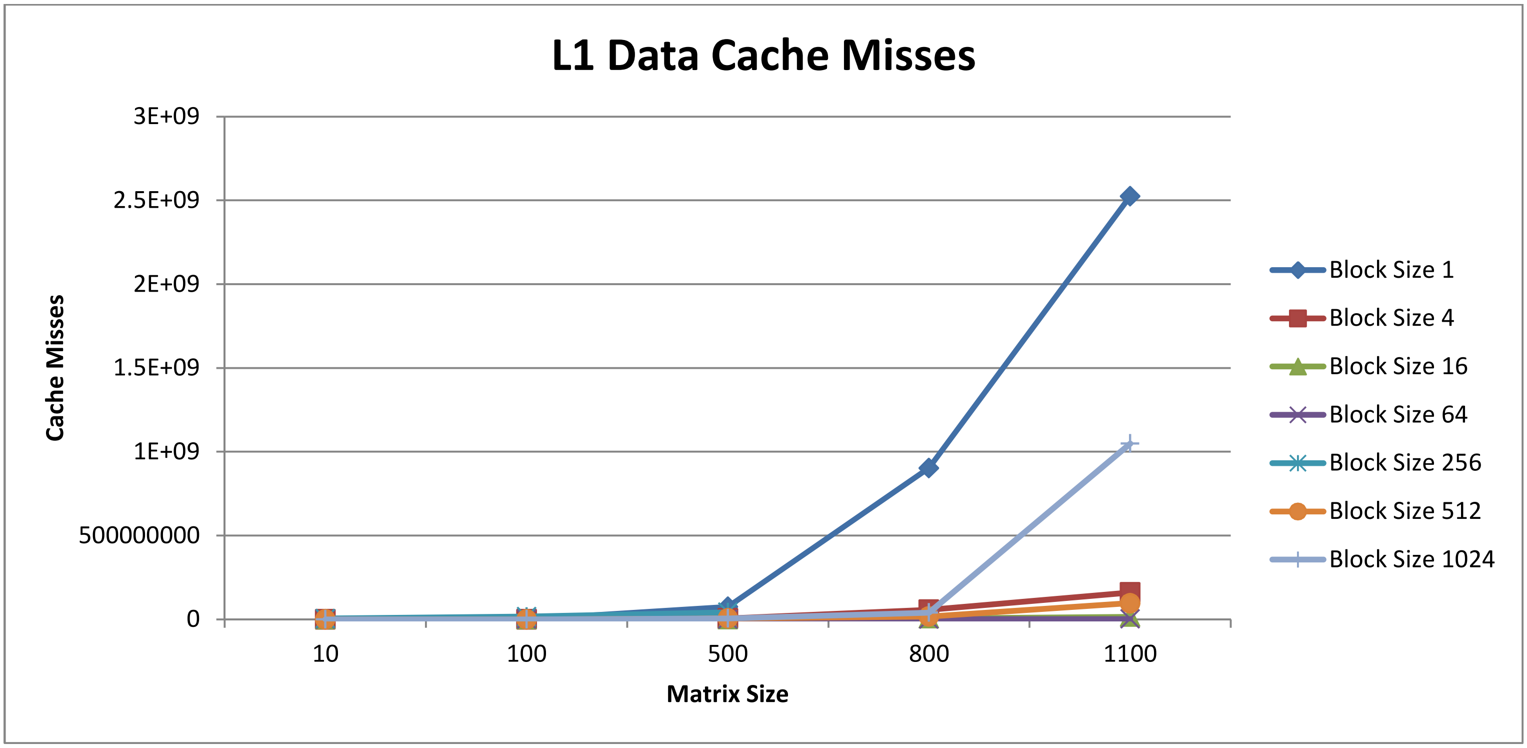
Explanation :

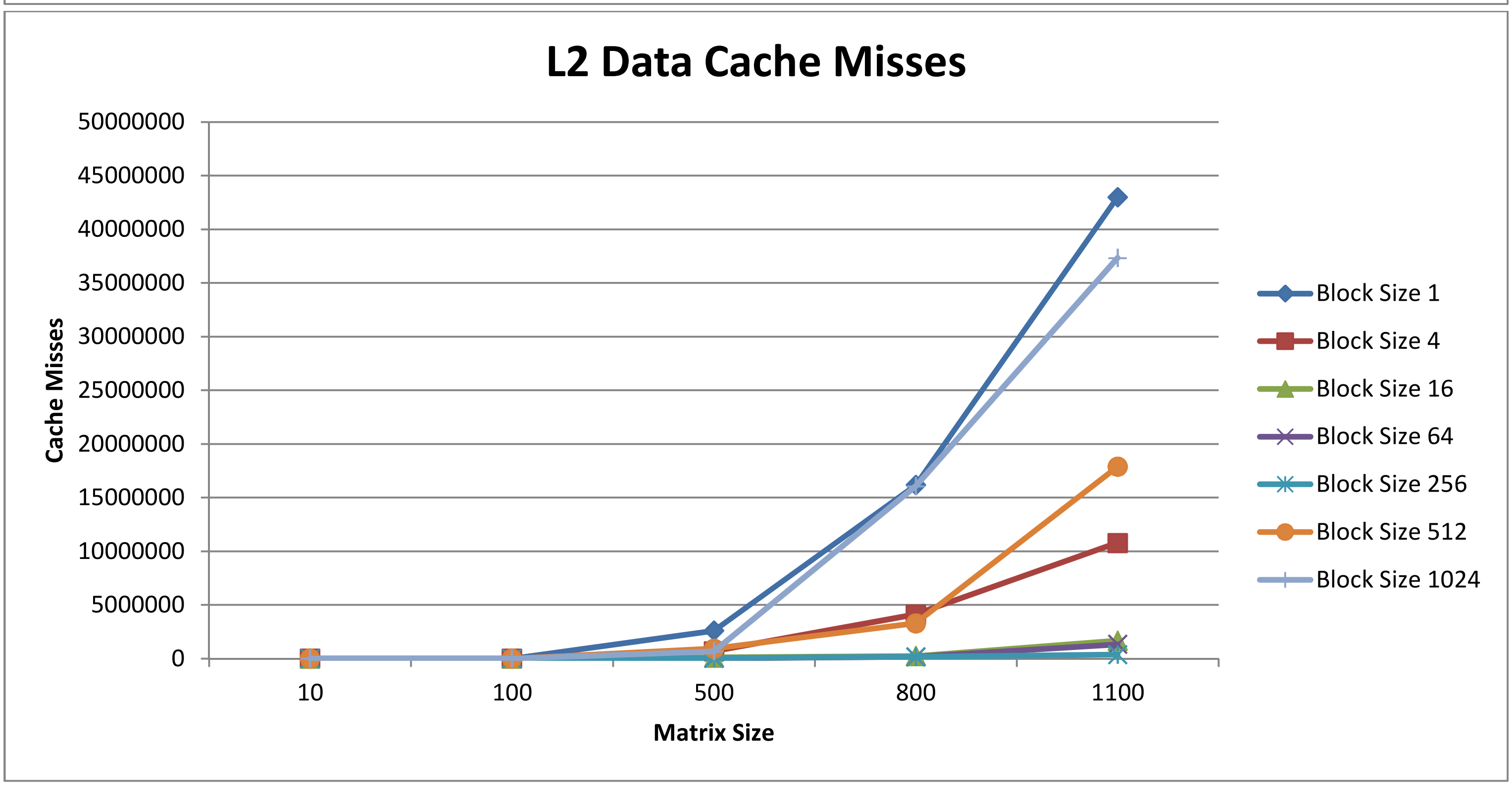
- As the block size increases from 1, subsequent data is available in cache due to temporal locality. Hence reduction in the number of cache misses.
- But as the block size increases beyond a certain level (in the above case 256) the amount of data that can be accomodated in a cache line is limited. Hence, rise in cache misses.

Analysis with Different Matrix Sizes:

L1 Data Cache Misses							
Matrix Size	**Block Size 1**	**Block Size 4**	**Block Size 16**	**Block Size 64**	**Block Size 256**	**Block Size 512**	**Block Size 1024**
10	85	91	96	84	134	73	98
100	1433	1344	1428	917	588	527	531
500	75039562	5630696	1308354	158001	4407889	5467973	5500649
800	902736743	57114256	6044457	933019	16616791	17637483	40194796
1100	2524934907	160360162	15202815	2190729	42550320	96241422	1049286914

L2 Data Cache Misses							
Matrix Size	**Block Size 1**	**Block Size 4**	**Block Size 16**	**Block Size 64**	**Block Size 256**	**Block Size 512**	**Block Size 1024**
10	58	70	57	59	63	60	54
100	992	994	911	479	415	409	413
500	2593040	692807	103808	34903	40027	927808	670625
800	16168224	4109630	234950	168403	170307	3285606	16055812
1100	42986243	10784785	1671032	1328504	375480	17880677	37324339


L1 Data Cache Misses
3E+09
2.5E+09
2E+09
1.5E+09
1E+09
500000000
0
Cache Misses
10
100
500
800
1100
Matrix Size
Block Size 1
Block Size 4
Block Size 16
Block Size 64
Block Size 256
Block Size 512
Block Size 1024

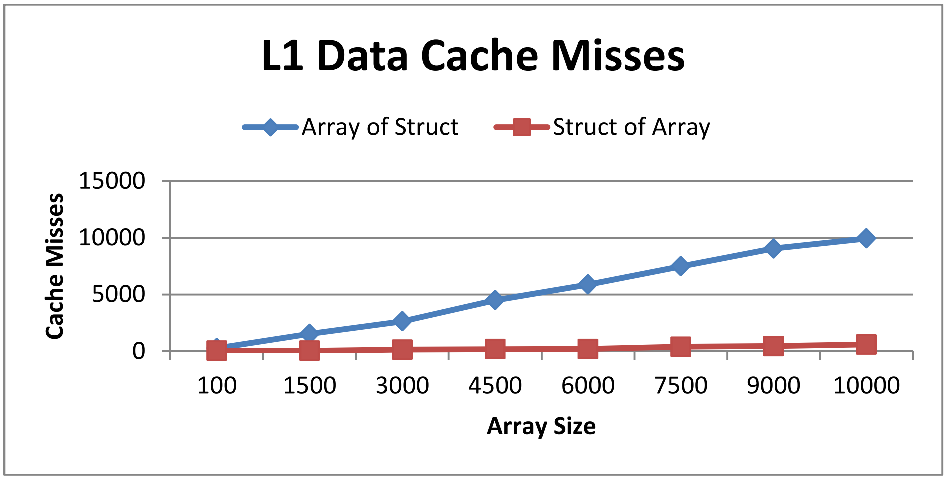

L2 Data Cache Misses
50000000
45000000
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Cache Misses
10
100
500
800
1100
Matrix Size
Block Size 1
Block Size 4
Block Size 16
Block Size 64
Block Size 256
Block Size 512
Block Size 1024

- **Analysis for Part 2:**

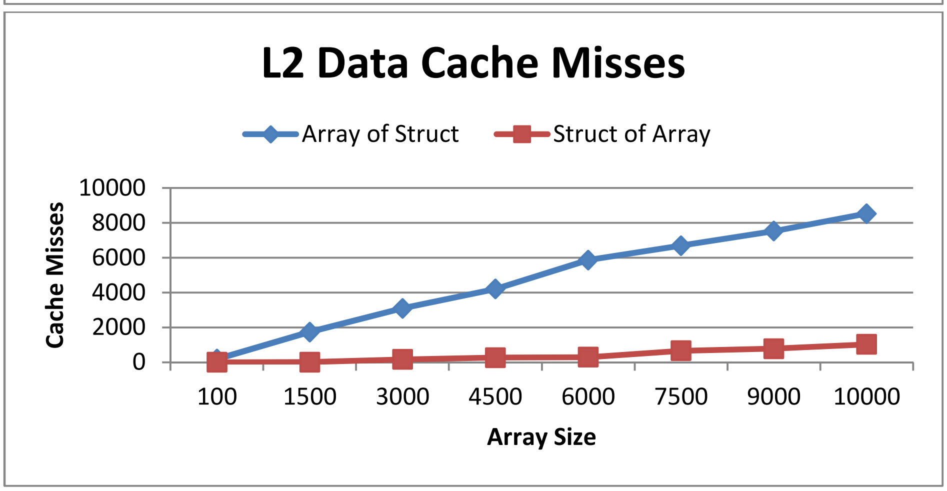
Sample program implementing two different data layouts.	
Layout 1	**Layout 2**

struct Sample1{
int x;
double y[100];
};
 	
Sample1 s[SIZE];
 	
for(i=0;i<SIZE;i++)
s[i].x = 25;
	struct Sample2{
	int x[SIZE];
	double y[100];
	};
	
	Sample2 s2;
	
	for(j=0;j<SIZE;j++)
	s2.x[i] = 25;

L1 Data Cache Misses		
Array Size	Array of Struct	Struct of Array
100	263	48
1500	1520	40
3000	2634	141
4500	4494	183
6000	5880	192
7500	7481	393
9000	9056	448
10000	9939	593



L2 Data Cache Misses		
Array Size	Array of Struct	Struct of Array
100	186	4
1500	1735	8
3000	3091	166
4500	4210	266
6000	5868	297
7500	6696	665
9000	7531	781
10000	8529	1026



Explanation:

- In case of Array of struct, there are 100 double variables (800 bytes) beside the variable x. Hence, when there is a compulsory miss for the first time, double variables are also fetched into the cache but never used. The subsequent value of x is available far apart and hence to access the value of x in the subsequent array index, we need to fetch the data from the memory into the cache. **There is no spatial locality**. Hence the number of misses keeps on increasing with the increase in the array size. Or in other words, the number of cache misses is more than the case of struct of array.
- In case of **Struct of Array**, the data which needs to be accessed are available next to each other (**Spatial locality**). Hence when the compulsory miss is encountered for the first time, the subsequent data is also fetched into cache, thereby avoiding future cache misses. Hence, the struct of array has less cache misses as compared to array of struct.
- Diagrammatically, both the data layout can be represented in the following form:

Data Layout 1 (Array of struct) –The data to be accessed are far apart. No spatial locality.

X	y[0]	y[1]	……..	y[99]	X	y[0]	y[1]	……..	y[99]	……..	x	y[0]	y[1]	……..	y[99]

Data Layout 2 (Struct of array) – The data to be accessed are next to each other. Less misses due to spatial locality.

X[0]	X[1]	…….	X[SIZE-2]	X[SIZE-1]	y[0]	y[1]	……..	y[99]