CS-525 Homework 1

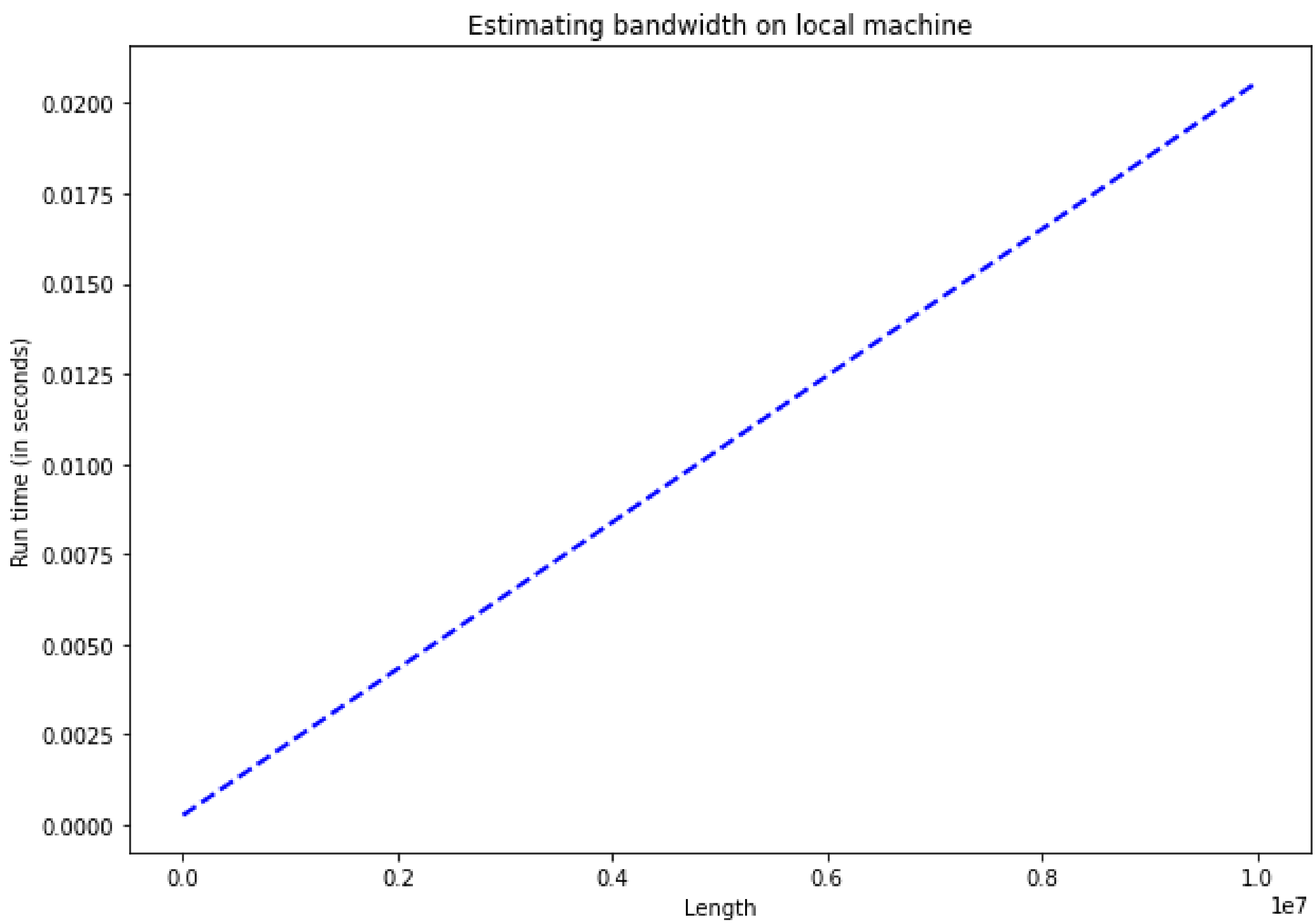
Q1. Sol - Following is the line chart of length vs runtime **recorded on local machine** for running the array on length varying from 1000 to 10000000. Here, the array is an integer array.

I calculated the slope of the line using python's NumPy function, polyfit, which gave the value as 2.0313350303314353e-09.

As Bandwidth is calculated as (number of iterations * sizeof(int) / runtime),

The bandwidth of the local machine will be (1/2.0313350303314353e-09) * 4 Bytes, which is approximately 1.96 GB/sec.

Therefore, the **bandwidth of the local machine is 1.96 GB/sec.**



Output of the code:

Length: 1000 Time Measured: 5e-06 seconds

Length: 10000 Time Measured: 4.7e-05 seconds

Length: 100000 Time Measured: 0.000446 seconds

Length: 1000000 Time Measured: 0.002871 seconds

Length: 10000000 Time Measured: 0.02052 seconds

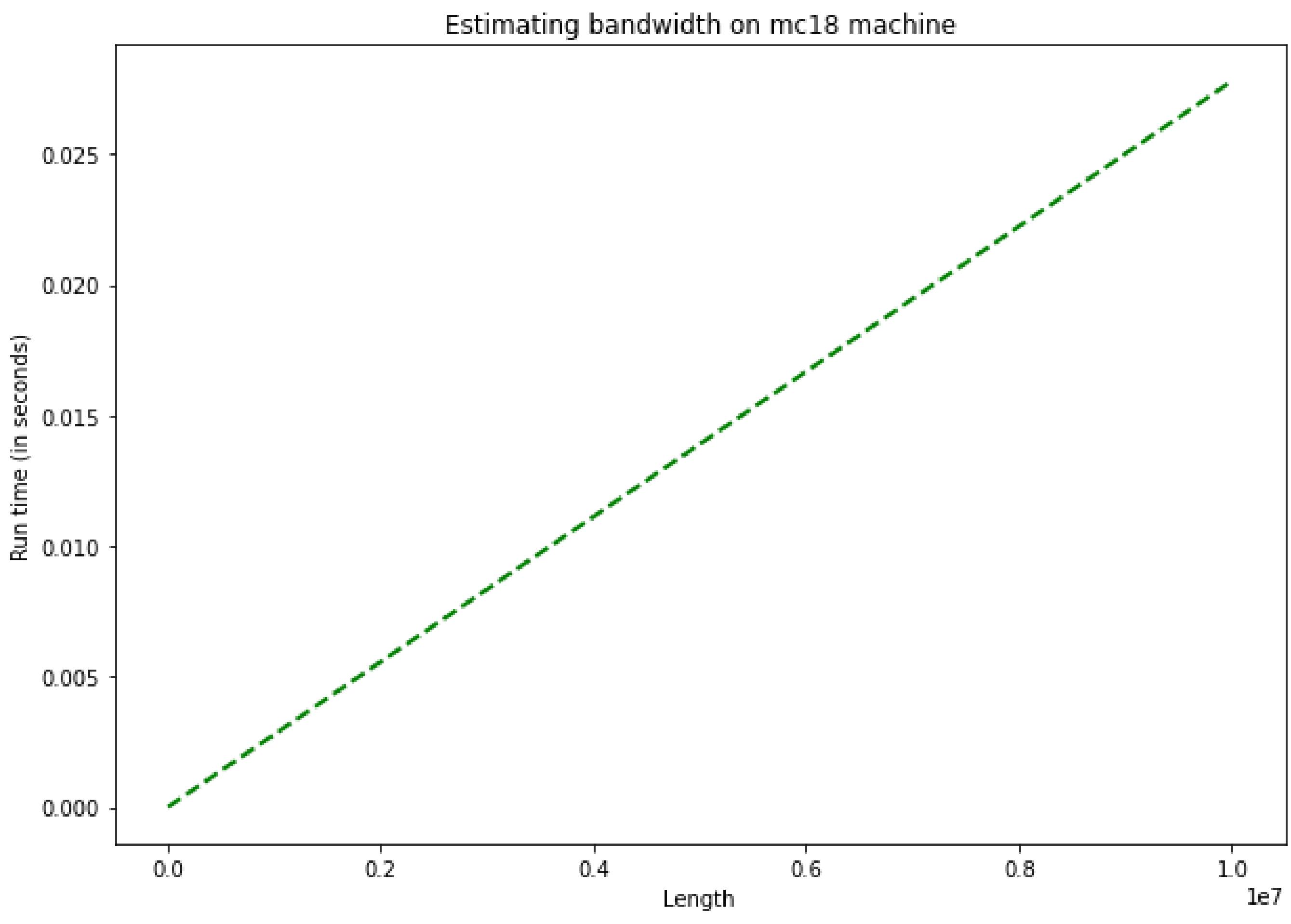
Following is the line chart of length vs runtime **recorded on mc18 machine** for running the array on length varying from 1000 to 10000000. Here, the array is an integer array.

I calculated the slope of the line using python's NumPy function, polyfit, which gave the value as 2.779163334086171e-09.

As Bandwidth is calculated as (number of iterations * sizeof(int) / runtime),

The bandwidth of the local machine will be (1/2.779163334086171e-09) * 4 Bytes, which is approximately 1.4 GB/sec.

Therefore, the **bandwidth of the mc18 machine is 1.4 GB/sec.**



Output of the code:

Length: 1000 Time Measured: 4e-06 seconds

Length: 10000 Time Measured: 3.3e-05 seconds

Length: 100000 Time Measured: 0.000319 seconds

Length: 1000000 Time Measured: 0.003292 seconds

Length: 10000000 Time Measured: 0.027753 seconds

Execute the code:

Code will be found inside hw1/question1/   folder

Compile: g++ main.cpp -o ./run

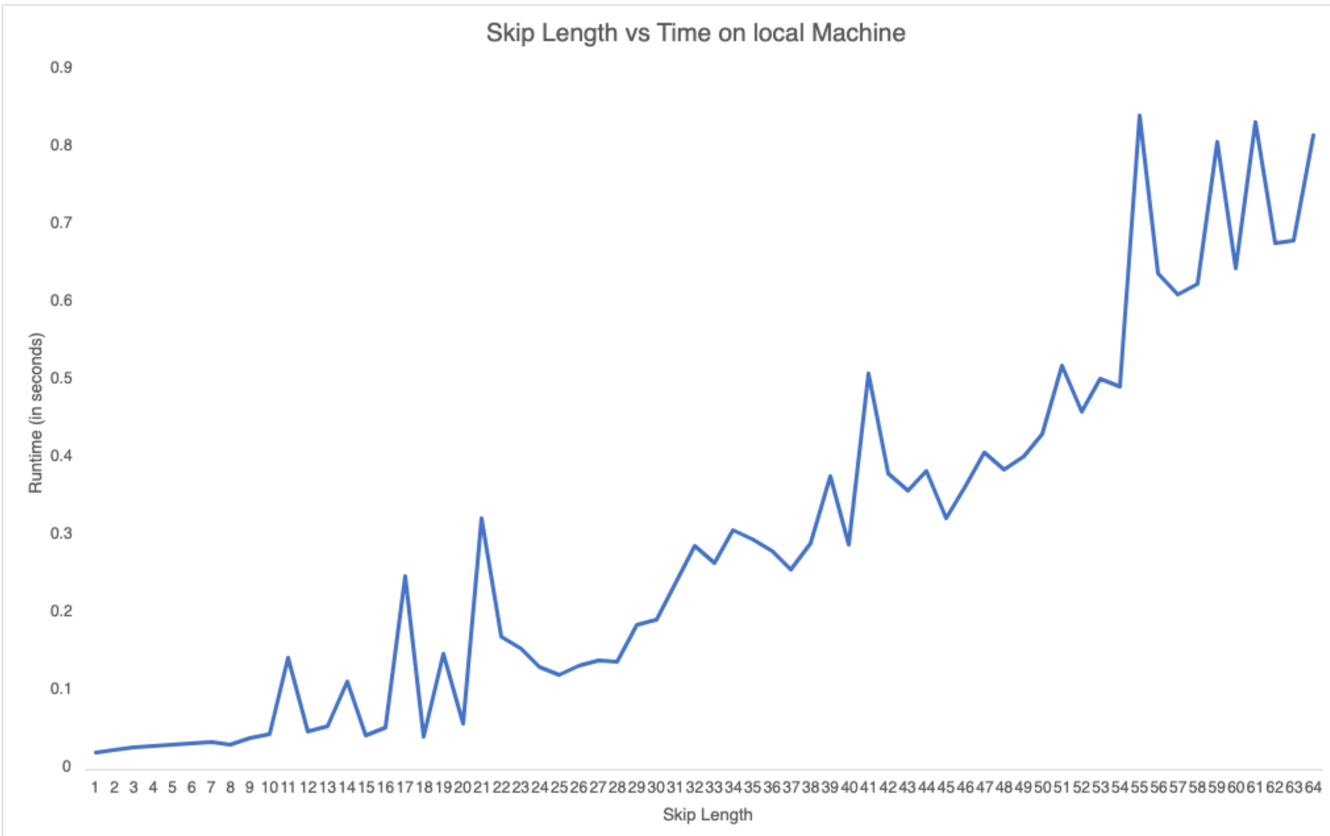
Execute: ./run

Output: output will be saved in output.txt file

Q2. Sol – Following is the line chart of skip length vs the runtime **recorded on local machine** for strided access on an array of length 10000000.

Here, we are trying to change the skip value from 1 to 64, and accordingly determining the time it takes for each execution.

We see multiple spikes/jumps in the graph. The First Spike occurs when the L1 cache size = skip size * 4 Bytes (As integer array is used), hence leading to cache miss and increasing the runtime. Therefore, the cache line size is 11*4 = 44 Bytes as the first spike occurs on skip = 11.



Output of the code:

Skip	:	1	Time	Measured:	0.021581	seconds
Skip	:	2	Time	Measured:	0.025748	seconds
Skip	:	3	Time	Measured:	0.028772	seconds
Skip	:	4	Time	Measured:	0.03065	seconds
Skip	:	5	Time	Measured:	0.031108	seconds

Skip	:	6	Time	Measured:	0.034324	seconds
Skip	:	7	Time	Measured:	0.035256	seconds
Skip	:	8	Time	Measured:	0.031555	seconds
Skip	:	9	Time	Measured:	0.039959	seconds
Skip	:	10	Time	Measured:	0.044796	seconds
Skip	:	11	Time	Measured:	0.143305	seconds
Skip	:	12	Time	Measured:	0.048847	seconds
Skip	:	13	Time	Measured:	0.056347	seconds
Skip	:	14	Time	Measured:	0.112299	seconds
Skip	:	15	Time	Measured:	0.043228	seconds
Skip	:	16	Time	Measured:	0.054685	seconds
Skip	:	17	Time	Measured:	0.248118	seconds
Skip	:	18	Time	Measured:	0.041528	seconds
Skip	:	19	Time	Measured:	0.147947	seconds
Skip	:	20	Time	Measured:	0.059452	seconds
Skip	:	21	Time	Measured:	0.322596	seconds
Skip	:	22	Time	Measured:	0.170589	seconds
Skip	:	23	Time	Measured:	0.155386	seconds
Skip	:	24	Time	Measured:	0.132185	seconds
Skip	:	25	Time	Measured:	0.121409	seconds
Skip	:	26	Time	Measured:	0.13389	seconds
Skip	:	27	Time	Measured:	0.140865	seconds

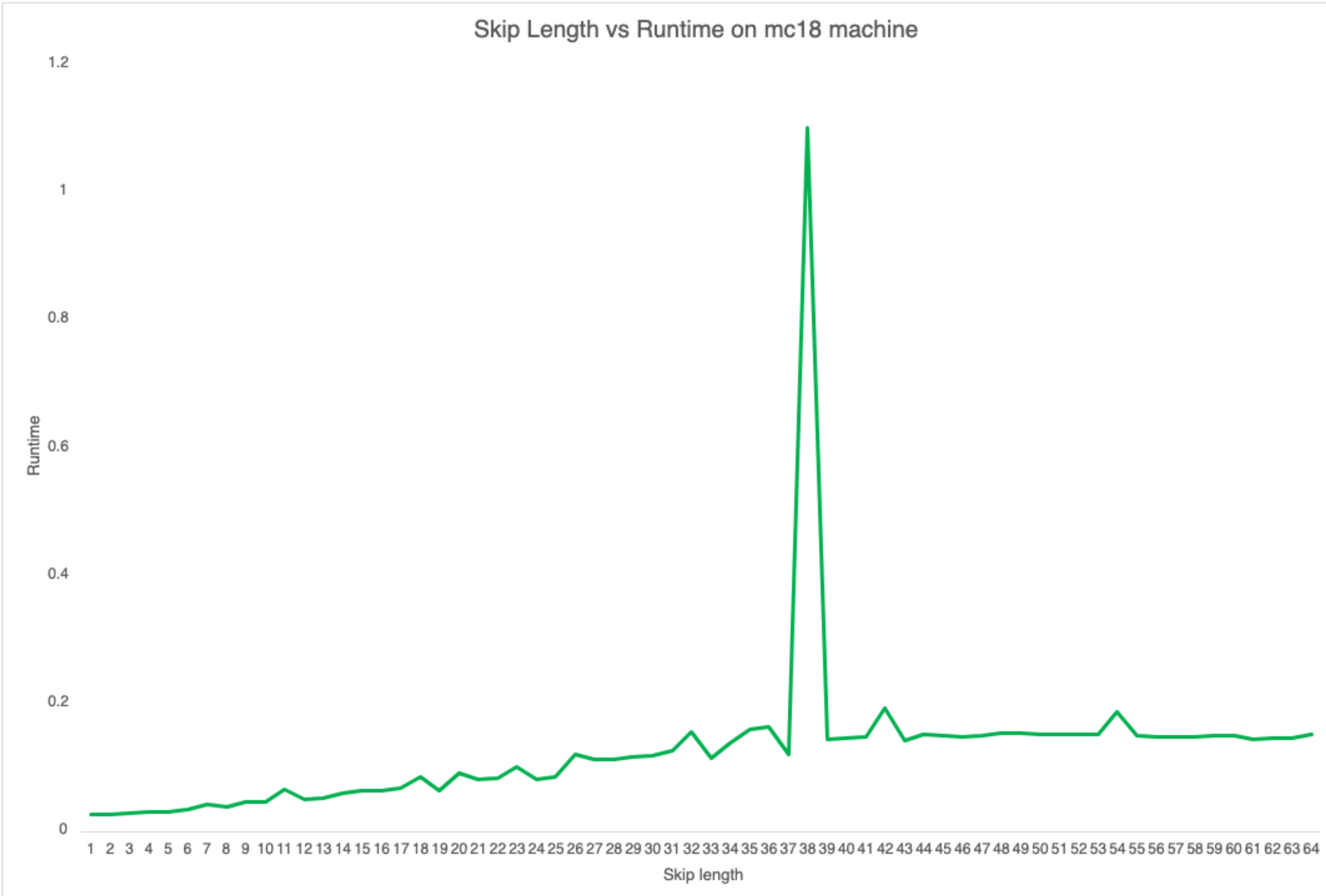
Skip	:	28	Time	Measured:	0.13886	seconds
Skip	:	29	Time	Measured:	0.185856	seconds
Skip	:	30	Time	Measured:	0.192943	seconds
Skip	:	31	Time	Measured:	0.238499	seconds
Skip	:	32	Time	Measured:	0.28716	seconds
Skip	:	33	Time	Measured:	0.265853	seconds
Skip	:	34	Time	Measured:	0.30833	seconds
Skip	:	35	Time	Measured:	0.296642	seconds
Skip	:	36	Time	Measured:	0.280089	seconds
Skip	:	37	Time	Measured:	0.257914	seconds
Skip	:	38	Time	Measured:	0.291773	seconds
Skip	:	39	Time	Measured:	0.376962	seconds
Skip	:	40	Time	Measured:	0.289595	seconds
Skip	:	41	Time	Measured:	0.508869	seconds
Skip	:	42	Time	Measured:	0.380746	seconds
Skip	:	43	Time	Measured:	0.359069	seconds
Skip	:	44	Time	Measured:	0.384285	seconds
Skip	:	45	Time	Measured:	0.322423	seconds
Skip	:	46	Time	Measured:	0.364728	seconds
Skip	:	47	Time	Measured:	0.407175	seconds
Skip	:	48	Time	Measured:	0.386364	seconds
Skip	:	49	Time	Measured:	0.403257	seconds

Skip	:	50	Time	Measured:	0.431628	seconds
Skip	:	51	Time	Measured:	0.519867	seconds
Skip	:	52	Time	Measured:	0.460685	seconds
Skip	:	53	Time	Measured:	0.50291	seconds
Skip	:	54	Time	Measured:	0.492047	seconds
Skip	:	55	Time	Measured:	0.840808	seconds
Skip	:	56	Time	Measured:	0.63862	seconds
Skip	:	57	Time	Measured:	0.610418	seconds
Skip	:	58	Time	Measured:	0.625422	seconds
Skip	:	59	Time	Measured:	0.807112	seconds
Skip	:	60	Time	Measured:	0.64452	seconds
Skip	:	61	Time	Measured:	0.832844	seconds
Skip	:	62	Time	Measured:	0.677818	seconds
Skip	:	63	Time	Measured:	0.680385	seconds
Skip	:	64	Time	Measured:	0.816769	seconds

Following is the line chart of skip length vs the runtime **recorded on mc18 machine** for strided access on an array of length 10000000.

Here, we are trying to change the skip value from 1 to 64, and accordingly determining the time it takes for each execution.

We see multiple spikes/jumps in the graph. The First Spike occurs when the L1 cache size = skip size * 4 Bytes (As integer array is used), hence leading to cache miss and increasing the runtime. Therefore, the cache line size is 11*4 = 44 Bytes as the first spike occurs on skip = 11.



Output of the code:

Skip	:	1	Time	Measured:	0.026748	seconds
Skip	:	2	Time	Measured:	0.02771	seconds
Skip	:	3	Time	Measured:	0.028636	seconds
Skip	:	4	Time	Measured:	0.030494	seconds
Skip	:	5	Time	Measured:	0.031865	seconds
Skip	:	6	Time	Measured:	0.034405	seconds
Skip	:	7	Time	Measured:	0.041722	seconds
Skip	:	8	Time	Measured:	0.038625	seconds
Skip	:	9	Time	Measured:	0.046553	seconds

Skip	:	10	Time	Measured:	0.046963	seconds
Skip	:	11	Time	Measured:	0.06698	seconds
Skip	:	12	Time	Measured:	0.049992	seconds
Skip	:	13	Time	Measured:	0.052923	seconds
Skip	:	14	Time	Measured:	0.06104	seconds
Skip	:	15	Time	Measured:	0.063565	seconds
Skip	:	16	Time	Measured:	0.06396	seconds
Skip	:	17	Time	Measured:	0.068081	seconds
Skip	:	18	Time	Measured:	0.085509	seconds
Skip	:	19	Time	Measured:	0.065079	seconds
Skip	:	20	Time	Measured:	0.091044	seconds
Skip	:	21	Time	Measured:	0.082685	seconds
Skip	:	22	Time	Measured:	0.083122	seconds
Skip	:	23	Time	Measured:	0.101676	seconds
Skip	:	24	Time	Measured:	0.082315	seconds
Skip	:	25	Time	Measured:	0.086149	seconds
Skip	:	26	Time	Measured:	0.121566	seconds
Skip	:	27	Time	Measured:	0.112489	seconds
Skip	:	28	Time	Measured:	0.112785	seconds
Skip	:	29	Time	Measured:	0.116453	seconds
Skip	:	30	Time	Measured:	0.118352	seconds
Skip	:	31	Time	Measured:	0.127022	seconds

				Measur	0.1569	second
Skip	:	32	Time	ed:	96	s
				Measur	0.1152	second
Skip	:	33	Time	ed:	73	s
				Measur	0.1379	second
Skip	:	34	Time	ed:	86	s
				Measur	0.1597	second
Skip	:	35	Time	ed:	17	s
				Measur	0.1640	second
Skip	:	36	Time	ed:	03	s
				Measur	0.1208	second
Skip	:	37	Time	ed:	83	s
				Measur	1.1012	second
Skip	:	38	Time	ed:	9	s
				Measur	0.1435	second
Skip	:	39	Time	ed:	22	s
				Measur	0.1457	second
Skip	:	40	Time	ed:	62	s
				Measur	0.1491	second
Skip	:	41	Time	ed:	05	s
				Measur	0.1942	second
Skip	:	42	Time	ed:	29	s
				Measur	0.1429	second
Skip	:	43	Time	ed:	55	s
				Measur		second
Skip	:	44	Time	ed:	0.1519	s
				Measur	0.1496	second
Skip	:	45	Time	ed:	32	s
				Measur	0.1483	second
Skip	:	46	Time	ed:	15	s
				Measur	0.1510	second
Skip	:	47	Time	ed:	74	s
				Measur	0.1532	second
Skip	:	48	Time	ed:	48	s
				Measur	0.1532	second
Skip	:	49	Time	ed:	97	s
				Measur	0.1526	second
Skip	:	50	Time	ed:	47	s
				Measur	0.1523	second
Skip	:	51	Time	ed:	86	s
				Measur	0.1523	second
Skip	:	52	Time	ed:	08	s
				Measur	0.1531	second
Skip	:	53	Time	ed:	21	s

Skip	:	54	Time	Measured:	0.187995	seconds
Skip	:	55	Time	Measured:	0.150578	seconds
Skip	:	56	Time	Measured:	0.148834	seconds
Skip	:	57	Time	Measured:	0.149065	seconds
Skip	:	58	Time	Measured:	0.148769	seconds
Skip	:	59	Time	Measured:	0.149927	seconds
Skip	:	60	Time	Measured:	0.150008	seconds
Skip	:	61	Time	Measured:	0.145307	seconds
Skip	:	62	Time	Measured:	0.146463	seconds
Skip	:	63	Time	Measured:	0.146184	seconds
Skip	:	64	Time	Measured:	0.151852	seconds

Execute the code:

Code will be found inside hw1/question2/   folder

Compile: g++ main.cpp -o ./run

Execute: ./run

Output: output will be saved in output.txt file

Q3. Sol -

Following is the output of loop unrolling from 1, 2, 3, 4, 5, 6 on **local machine**

Unroll count: 1 Time Measured: 0.015959 seconds

Unroll count: 2 Time Measured: 0.008089 seconds

Unroll count: 3 Time Measured: 0.007784 seconds

Unroll count: 4 Time Measured: 0.006991 seconds

Unroll count: 5 Time Measured: 0.006633 seconds

Unroll count: 6 Time Measured: 0.006363 seconds

Following is the output of unrolling from 1, 2, 3, 4, 5, 6 on **mc18 machine**

Unroll count: 1 Time Measured: 0.015094 seconds

Unroll count: 2 Time Measured: 0.013467 seconds

Unroll count: 3 Time Measured: 0.012783 seconds

Unroll count: 4 Time Measured: 0.012211 seconds

Unroll count: 5 Time Measured: 0.012026 seconds

Unroll count: 6 Time Measured: 0.011794 seconds

From the above results on both the platforms, we can comment that as the unrolling factor increases from 1 to 6, the time to execute to loop decreases. This is because:

- It reduces the loop overhead, i.e., incrementing and testing of the loop counter is reduced ,
- Improves the pipeline behavior by reducing jumps back to a loop's entry.
- It increases instruction-level parallelism, i.e., based on the unrolling factor, more instructions can be executed in parallel.

But, at the same time it increases the code size.

For example, consider the loop:

```
for (int i=0; i<length; i++) {

sum += arr[i];

}
```

Here, let us start with sum += arr[0]. There will be a cache miss for arr[0] leading to accessing 4 words, .e., arr[0], arr[1], arr[2], arr[3] from the memory. Now, all these 4 been in cache memory cannot be used in parallel, as the operation sum += arr[i] needs to be executed sequentially due to dependency on sum value.

Now, consider the loop unrolled twice:

```
for (int i=0; i<length; i=i+3) {

temp1 += arr[i];

temp2 += arr[i+1];

temp3 += arr[i+2];

}

int temp = temp1 + temp2 + temp3 ;
```

Here, let us start with temp1 += arr[0]. There will be a cache miss for arr[0] leading to accessing 4 words, I.e. arr[0], arr[1], arr[2], arr[3] from the memory. Now, all these 4 been in cache memory can be used in parallel increasing instruction-level parallelism, as all the 4 instructions in the loop can be executed in parallel as there is no dependency between the variables. Also at each step, the counter is incremented by 3, which reduces the loop overhead, therefore, improving the performance.

Execute the code:

Code will be found inside hw1/question3/   folder

Compile: g++ main.cpp -o ./run

Execute: ./run

Q4. Sol -

Following is the output of impact of loop interchanging on **local machine**

Matrix Vector Multiply, **length :1000** Time Measured: 0.006192 seconds

Matrix Vector Multiply InterChange, length :1000 Time Measured: 0.0053 seconds

Matrix Vector Multiply, **length :10000** Time Measured: 0.356515 seconds

Matrix Vector Multiply InterChange, length :10000 Time Measured: 0.83891 seconds

For local machine, I was unable to perform Matrix Vector Multiplication for length of 100000 due to out of memory issue. Hence, I have not implemented it on local machine, but it works on mc18 machine.

Following is the output of impact of loop interchanging on **mc18 machine**

Matrix Vector Multiply, **length :1000** Time Measured: 0.006871 seconds

Matrix Vector Multiply InterChange, length :1000 Time Measured: 0.015166 seconds

Matrix Vector Multiply, **length :10000** Time Measured: 0.38019 seconds

Matrix Vector Multiply InterChange, length :10000 Time Measured: 1.65345 seconds

Matrix Vector Multiply, **length :100000** Time Measured: 41.8175 seconds

Matrix Vector Multiply InterChange, length :100000 Time Measured: 669.095 seconds

To explain the above scenario,

Consider case 1: Matrix Vector Multiply:

```
for (int i = 0; i < n; i++){

for (int j = 0; j < n; j++) {

result[i] += matrix[i][j]*b[j];

}

}
```

Let is start with result[0] = matrix[0][0] * b[0]. This will have a cache miss at matrix [0][0] leading to accessing it in memory. As arrays are laid out as row major in C/C++, therefore, while accessing matrix[0][0], we will get matrix[0][0],matrix[0][1],matrix[0][2],matrix[0][3]. Hence, after the first cache miss, the next three iterations can be executed without cache miss. Same will be the case for vector b. There will be less cache miss leading to lesser runtime.

Now, consider case 2: Matric Vector Multiplication with loop interchange.

```
for (int j = 0; j < n; j++){

for (int i = 0; i < n; i++) {

result[i] += matrix[i][j]*b[j];

}

}
```

Let is start with result[0] = matrix[0][0] * b[0]. Here too, we will get cache miss for matrix[0][0], and we get matrix[0][0],matrix[0][1],matrix[0][2],matrix[0][3] words. But, however, in the second iteration, we will access matrix[1][0] * b[0]. This is a problem as there will be a miss for matrix[1][0] which will again give us 4 words, matrix[1][0],matrix[1][1],matrix[1][2],matrix[1][3]. Therefore, in every iteration, we get a cache miss as we are unable to use the bulk access, which leads to an increase in runtime as seen in the output.

Therefore, strided access is terrible, as we do not make use of bulk access and get worst performance.

Execute the code:

Code will be found inside hw1/question4/   folder

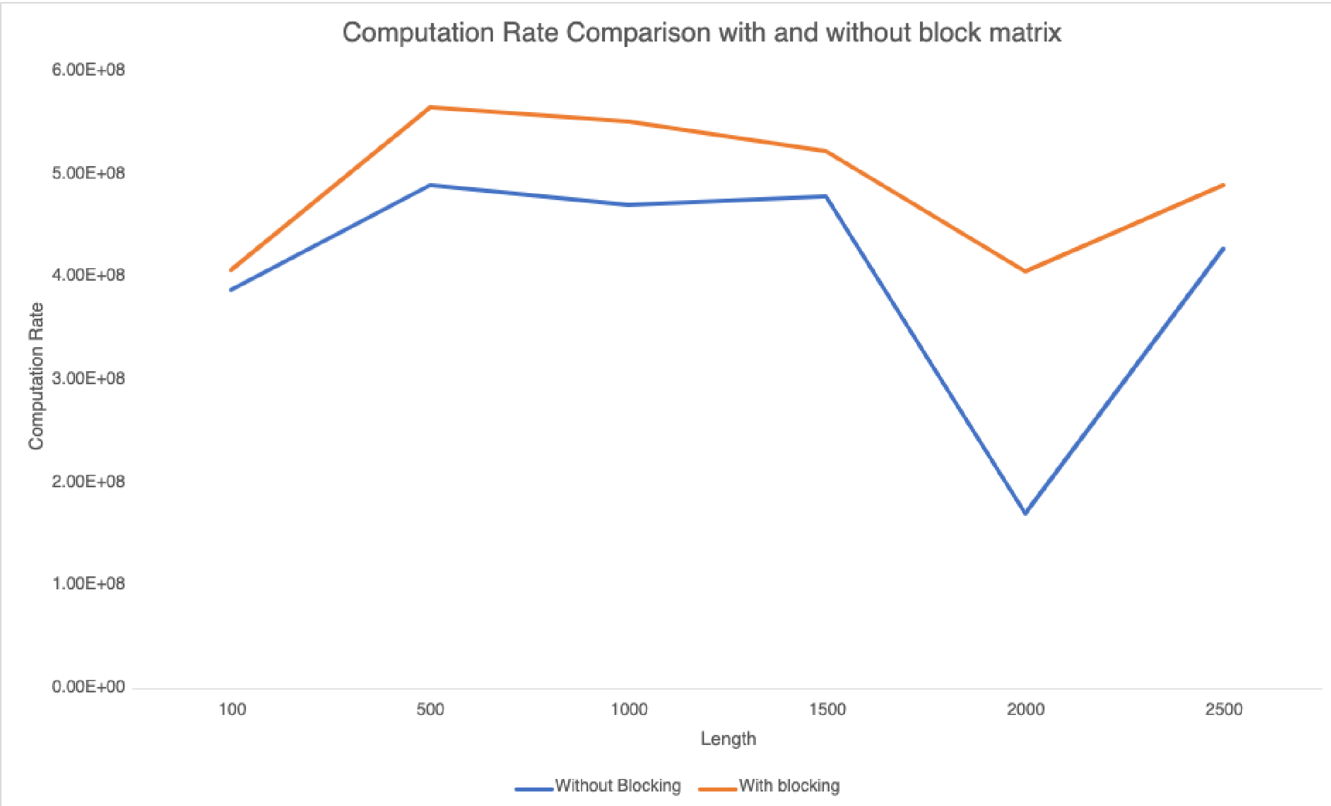
Compile: g++ main.cpp -o ./run

Execute: ./run

Output: output will be saved in output.txt file

Q5. Sol – Following is the Comparison chart between computation rates of matrix matrix multiplication with and without blocking on **Local Machine.**

Matrix Multiplication with length 5000 took a lot of time to execute, hence it has not been taken. Instead I have implemented for length 1500, 2000, 2500



Output of the code:

Length :100 Matrix Mul Computation Rate: 3.13234e+08 block size:25 Block Matrix Mul Computation Rate: 335795836.1

Length :500 Matrix Mul Computation Rate: 489049210.1 block size:25 Block Matrix Mul Computation Rate: 563661000.7

Length :1000 Matrix Mul Computation Rate: 482134622.1 block size:25 Block Matrix Mul Computation Rate: 501433221.5

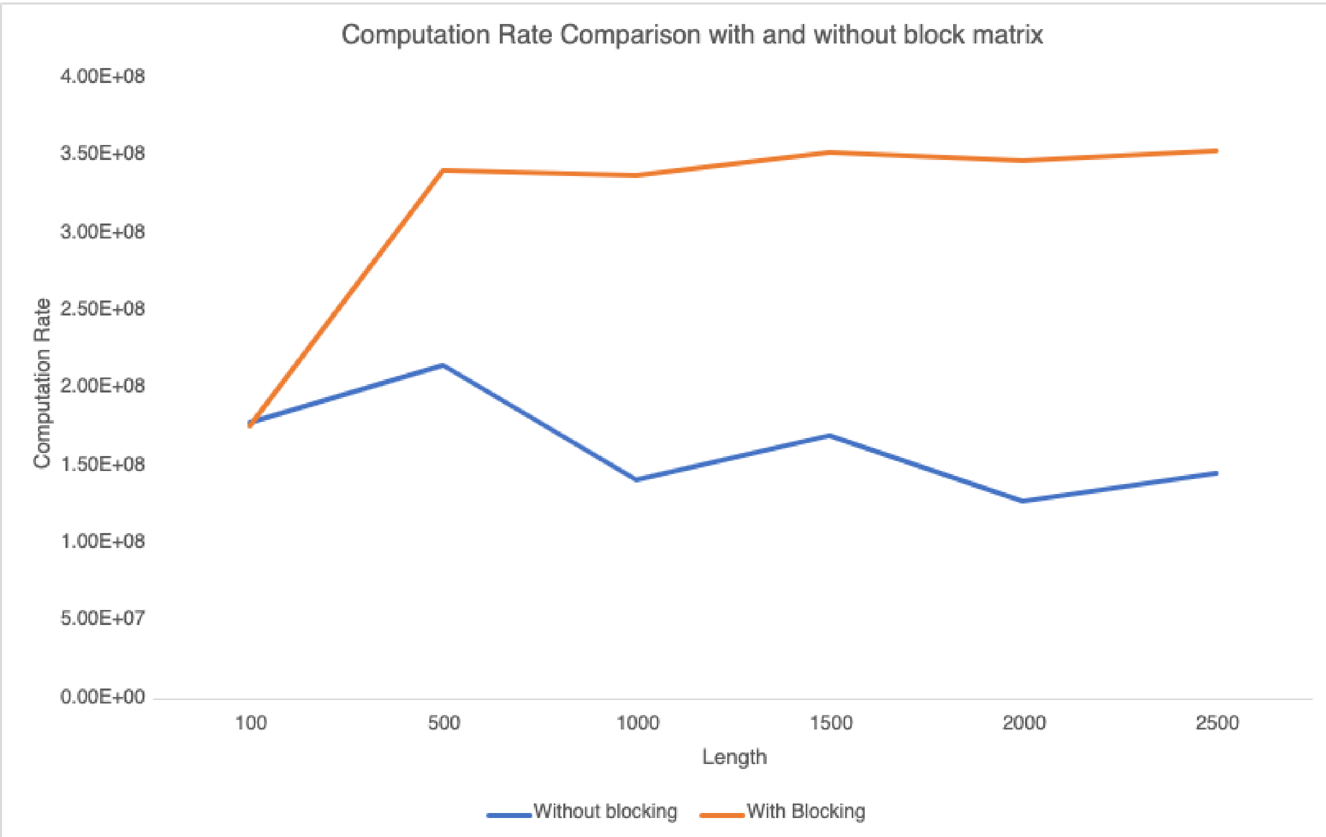
Length :1500 Matrix Mul Computation Rate: 471428005.7 block size:25 Block Matrix Mul Computation Rate: 418927530.6

Length :2000 Matrix Mul Computation Rate: 188793389.6 block size:25 Block Matrix Mul Computation Rate: 424790283

Length :2500 Matrix Mul Computation Rate: 417884353.8 block size:25 Block Matrix Mul Computation Rate: 508297348.3

Following is the Comparison chart between computation rates of matrix matrix multiplication with and without blocking on **mc18 machine**

Matrix Multiplication with length 5000 took a lot of time to execute, hence it has not been taken. Instead I have implemented for length 1500, 2000, 2500



Output of the code:

Length :100 Matrix Mul Computation Rate: 1.78571e+08 block size:25 Block Matrix Mul Computation Rate: 176118351.5

Length :500 Matrix Mul Computation Rate: 215414548.1 block size:25 Block Matrix Mul Computation Rate: 340605951.6

Length :1000 Matrix Mul Computation Rate: 142068429.7 block size:25 Block Matrix Mul Computation Rate: 338242815

Length :1500 Matrix Mul Computation Rate: 170440101.1 block size:25 Block Matrix Mul Computation Rate: 352469889.7

Length :2000 Matrix Mul Computation Rate: 127862425.1 block size:25 Block Matrix Mul Computation Rate: 347065026.2

Length :2500 Matrix Mul Computation Rate: 146227473.9 block size:25 Block Matrix Mul Computation Rate: 353768984.6

Execute the code:

Code will be found inside hw1/question5/ folder

Compile: g++ main.cpp -o ./run

Execute: ./run

Output: output will be saved in output.txt file

From the outputs, we can comment that computation rate for block matrix multiplication is better than normal matrix multiplication as we can store the blocks of matrix in the cache based on the cache size and block size (here, the block size used is 25), leading to data reuse causing less cache misses and better performance as compared to iterating through entire matrix in normal matrix multiplication.