Sriharsha Ayinala
1378833

Assignment 1

1. The charts below show the runtime (in cycles) to calculate the sum from 0 to 10000 (exclusive). They are bounded by 3 parameters:
   - Is the -O2 flag enabled or not?
   - Is the final sum printed?
   - Is the input number 10000 hard coded or given at runtime?

-O2 Flag Enabled		
	Sum Printed	**Sum Not Printed**
Hard Coded	77274	1302
User Input	94904	1289

-O2 Flag Disabled		
	Sum Printed	**Sum Not Printed**
Hard Coded	137660	62871
User Input	138633	63010

Most of the discrepancy in the timings is mostly attributed to compiler optimizations. When the –O2 flag is enabled, the code is optimized to run faster. In the case where the value to sum to is hardcoded in, the –O2 flag will calculate part of the sum, if not all, during compile time. Otherwise, it will calculate only during runtime. This is why there is a difference between when the –O2 flag is enabled when is hardcoded versus when it is not enabled. When the –O2 flag is not enabled, the sum is calculated during runtime anyways. The –O2 flag also optimizes for the case when the print statement is enabled or not. When the print statement is disabled and optimization is enabled, it optimizes it such that it will not enter the loop because there is no call for the sum.

The most accurate way of timing the computation with or without optimization with the having a user input the number such that no calculation is done during compile time. The sum should also be printed such that the loop runs.

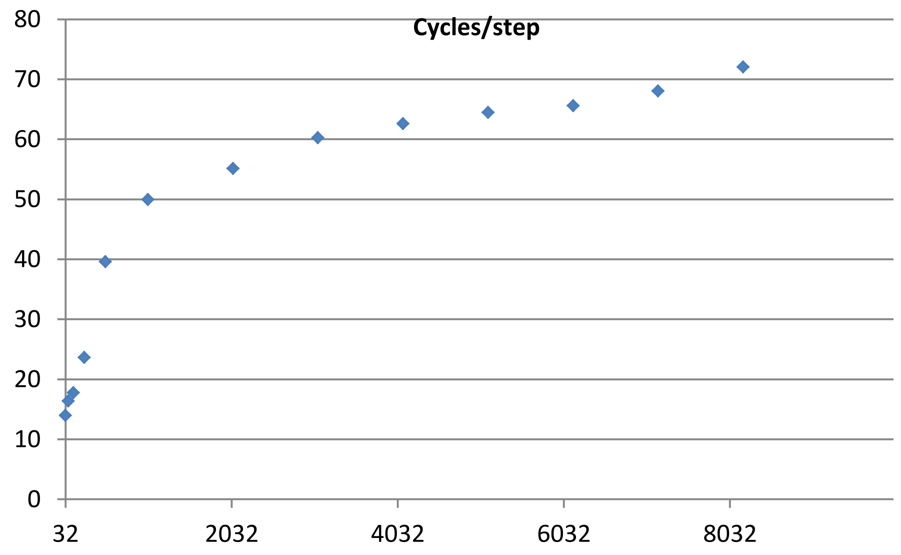
“argv[1]” is the second element of the input given to terminal when calling the program, in this case the number to sum to. “atoi(x)” converts string or char *x* to an integer. Thus “atoi(argv[1])” converts the terminal input for the number to sum until to an integer system it is a proper data type for the program to parse.

Sriharsha Ayinala
1378833

2. The pointer chasing benchmark measures the time it takes to request and retrieve the value of a random index by accessing the entire array at random indices. It will request the value of the index and then moves to the next index using the retrieved value as the next index. Moving from one index to do next is equivalent as retrieving and obtaining a value from memory. Dividing the length by the total cycles taken gives an estimated average of requesting and obtaining a value.

   In an int array of length *N*, it is *4N* bytes as each int is 4 bytes.

Size (kB)	Cycles/step
32	13.96020508
64	16.37298584
128	17.74746704
256	23.60986328
512	39.60473633
1024	49.94857025
2048	55.11474228
4096	62.61709595
8192	72.02857399
3072	60.25797399
5120	64.44828415
6144	65.61485926
7168	68.0408565



There is a sudden spike in the speed after a size of around 256 kB and it plateaus off at that new increased value. This is because of the different memory levels and their respective access times. When the array size got too big, it moved down to do next memory level where do read cycle is much slower. That is why we see such a spike and then a plateau at a higher amount of cycles per step.

The out of order processors will not affect the benchmark as the processor does not know which pointer would be accessed next as they are randomized during execution. As long as pointer access is random during execution, we can be sure that we are getting the true average.

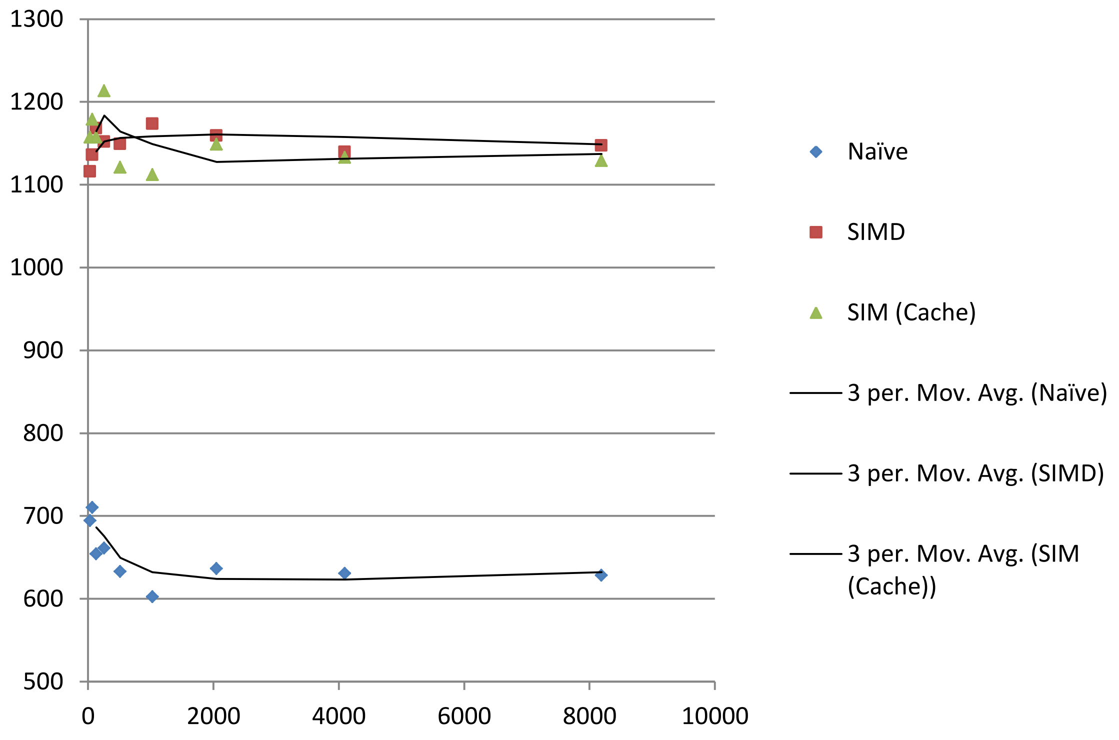
BONUS:

```
#include<set>

int unique(int A[], int N)
{
  std::set<int> unique;
  std::set<int>::iterator it;

  int j = 0;
  unique.insert();
  for(int i=0; i<N; ++i){
    j = arr[0];
    if(unique.find(j) == unique.end())
    {
      return i;
    }
    unique.insert(j);
  }
 return N
}
```

The code uses the C++ set template. It iterates the array from zero to the length of the array and adds whatever the index value is to a set. After each iteration, it will check the set to see if the value exists. If it does, return the number it has iterated through. This is the number of unique values *i* can iterate through the array. Otherwise it would keep inserting new values to the set until it has iterated through the whole array which means that all values of *i* is unique, or it will find a repeat and return the number of unique values *i* can have.

Sriharsha Ayinala
1378833

3.



Size (kB)	Bandwidth (Mbps)		
	Naïve	**SIMD**	**SIMD (Cache)**
32	694.444444	1116.071	1157.40741
64	710.227273	1136.364	1179.24528
128	654.450262	1168.224	1157.40741
256	661.375661	1152.074	1213.59223
512	632.911392	1149.425	1121.07623
1024	602.409639	1173.709	1112.34705
2048	636.739892	1159.42	1148.76508
4096	630.715863	1139.601	1133.14448
8192	628.584898	1147.282	1129.30548

There seems to be dips in the bandwidth after 512 kB for all 3 plots. This is so due to the different cache levels and their respective access times. As array size gets larger, it will not fit into the lower-level caches.

It is necessary to do a warm-up as we need to populate the cache such that there are no existing values in the cash that the program might retrieve during the test.

Sriharsha Ayinala
1378833

And inefficient copying procedure will cause the access of data to be at different memory levels at separate access during, causing the program do not accurately measure the precise bandwidth of a specific memory level that we want to measure.

_mm_prefetch(char const* a, int sel) - this loads one cache line of data from address a and puts it to a location closer to your processor, a lower-level cache.

_mm_load_si128(__m128i const* p) - this loads 128 bits and stores the value in a register.

_mm_stream_si128(__m128i *p, __m128i a) - this stores the data in a to the address p without polluting the caches. If the cache line containing address p is already in the cash, cash will be updated.

_mm_store_si128(__m128 i *p, __m128i a) - stores the data in a to memory in pointer p.

Sriharsha Ayinala
1378833

4.

	Flop/s	**IPC**
opt_simd_sgemm	8489151387	2.2015
opt_scalar1_sgemm	1082863869	2.5525
opt_scalar0_sgemm	698333920	0.5848
naive_sgemm	195524085	0.3275

$2n^3$ operations are done to a n-sized matrix.

IPCs over 1 probably occurred due to optimizations for out of order processors or parallel tasks where instructions are pipelined to multiply and/or add independent colons in rows simultaneously.

This is not always necessarily the case. Although the IPC might be higher, the time it takes to do the operations might be slower asked algorithm compresses more instructions into one cycle. However, the time it takes to complete the cycle might take longer due to inefficient algorithm.