CS194-15 Assignment 1: Measuring CPU Performance
Vanessa Matalon
23126281 / cs194-ax

* 1. Measuring Execution Time *

The configurations ordered in ascending order are as follows:

1. with -O2, does not print, argument is hardcoded: 1388 Ticks
2. with -O2, does print, argument is hardcoded: 1413 Ticks
3. with -O2, does not print, argument is not hardcoded: 7556 Ticks
4. with -O2, does print, argument is not hardcoded: 27463 Ticks
5. without -O2, does not print, argument is hardcoded: 66703 Ticks
6. without -O2, does print, argument is hardcoded: 66826 Ticks
7. without -O2, does not print, argument is not hardcoded: 68837 Ticks
8. without -O2, does print, argument is not hardcoded: 68978 Ticks

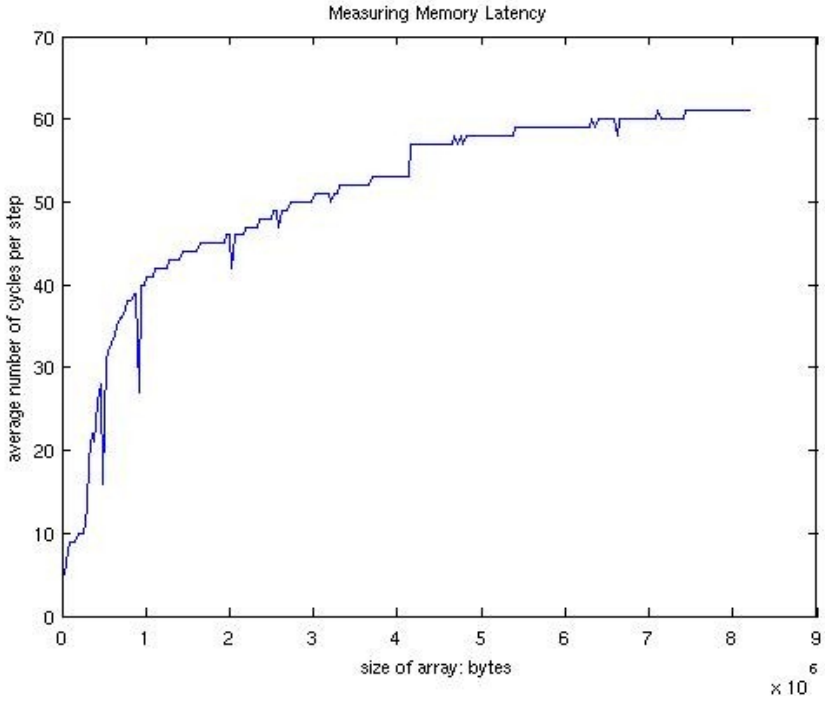
So the most obvious difference is observed when we decide to include or exclude the -O2 optimization: the computation takes longer when we exlude this optimization. Another difference that is observed is that the latency increases when we decide to have an argument inputted by the user because we must parse and convert it into an integer in order to use it, as opposed to having it hardcoded and ready to go. The third discrepancy that is worth mentioning is that the computation takes longer when we decide to print the sum (even if it is after we compute time elapsed) and I think this is because the instruction is fetched by a different stage of the pipeline.

The most accurate way of timing the computation is to have the argument hardcoded and to not print the sum at the end of the computation. This way we eliminate having to parse a command line argument and having the print instruction fetched by a potential pipeline.
Depending on whether we would like to evaluate the performance of the algorithm (implemented in a more optimal way that uses the processors at hand) or the computation executed by this particular program, we should include or discard the -O2 optimization respectively.

`argv[1]` is the first command line argument taken as a character array that the user inputs into the program (in this case, it is the 10000).
`atoi()` parses a string and returns its value as an int.

* 2. Measuring Memory Latency *

The pointer chasing benchmark measures the memory latency by averaging how long it takes to index into a randomly chosen location in an array.



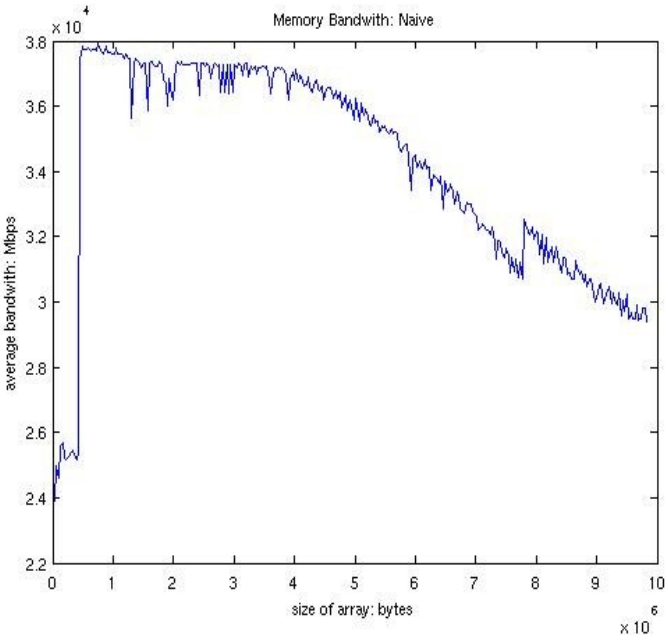
When I call sizeof(int) it is equal to 4 bytes.
An int array of length N on the computer I am running on is 4*N bytes.

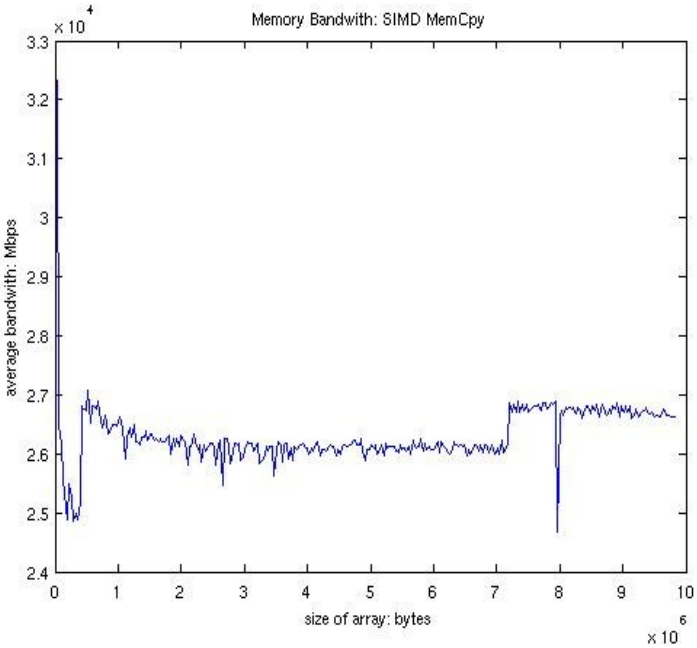
The shape of the curve seems logarithmic with bumps. The changes in slope decrease as the size of the array becomes larger because the caches can only store a limited amount of memory. Once the array is greater than ~4.5 Mb the number of cycles per step stays relatively unchanged, this is likely because the further away we get from the CPU, the amount of memory that we can store increases.

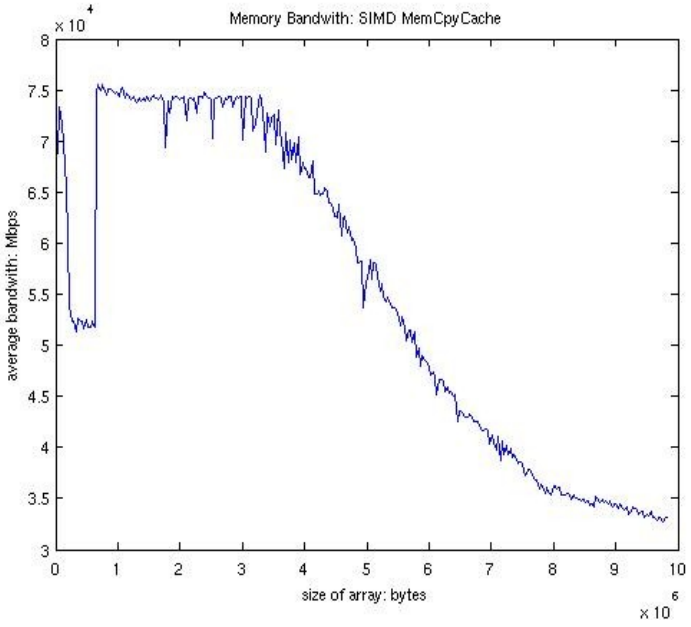
We can be sure that the steps in our benchmark are not being pipelined because the value that is looked up on the current iteration depends on the previous one, and therefore these must be executed in parallel.

BONUS: The first step is to initialize an array Z of 0's that has the same length as array A. Next, the function iterates through all of the indices in A that it can beginning at 0 (moving on to the index whose value is contained at 0, and so on). Each time the program finds itself beginning a new iteration, it checks whether or not `Z[index]=0`: if so, then it sets `Z[index]=1`, otherwise it breaks out of the loop. Once the iteration is broken, we sum the elements of Z in order to get the number of unique values i actually takes on.

* 3. Measuring Memory Bandwith *







Both the naïve and simd-cache implementations share roughly the same shape in their curves, they both start to decrease at ~4Mb because as the array size increases so does the cache miss rate. `simd_memcpy_cache` decreases faster because the optimizations lose their performance improvements, whereas in the naïve implementation there isn't much to lose to begin with. `simd_memcpy` stays roughly constant because the function streams the values without putting them into the cache which will maintain the bandwith relatively low regardles of the array size.

It is necessary to warm up the cache because having it full of relevant information will allow us to maximize bandwith when we are measuring it.

Inefficient array copying yields an inaccurate measure of bandwith because it does not use the processor to its fullest potential. The memory system can transfer more data to the CPU, but the procedure does not take advantage of this and thus limits the throughput.

The SSE intrinsics that were used in `simd_memcpy` & `simd_memcpy_cache` are:

1. `_mm_prefetch`: Fetches the data at the address from memory and loads it into a location in the cache heirarchy.
2. `_mm_load_si128`: Loads 128-bits of integer data from memory into a destination that must be aligned on a 16-byte boundary.
3. `_mm_stream_si128`: Stores 128-bits of integer data from an address into memory without passing through the caches. The destination must be aligned on a 16-byte boundary.
4. `_mm_store_si128`: Stores 128-bits of integer data from an address into memory. The destination must be aligned on a 16-byte boundary.

* 4. Measuring Flops and IPC *

	`opt_simd_sgemm`	`opt_scalar1_sgemm`	`opt_scalar0_sgemm`	`naive_sgemm`
GFlops	8.309650828	1.012909019	0.827211281	0.198091684
IPC	2.352511486	2.595655561	0.731777199	0.32861998

There are $2n^3$ adds and multiplications in a naïve implementation of matrix multiply: in order to get each of the n^2 elements in the matrix we need to do n adds and n multiplies ($a_0b_0+a_1b_1+...+a_{n-1}b_{n-1}$). If we are using the SIMD instructions than we can perform nearly eight times less floating point operations

The processor will complete more than one instruction per cycle if we are taking advantage of the parallel nature of some problems. For instance, when we use SSE Intrinsics we are performing a single instruction on multiple data points and if we have a superscalar CPU we can dynamically check for data dependencies and execute instructions in a data flow order which allows for the CPU to execute multiple instructions per cycle.

`opt_scalar1_sgemm` completes more instructions per cycle by doing out of order processing because waiting to start a computation that is independent is wasting time. `opt_simd_sgemm` has a lower IPC while higher GFlops because the SSE Intrinsics allow the program to execute a single instruction on multiple data points at a time but it doesn't necessarily take full advantage of out of order execution. And therefore, a higher IPC is not always an indicator of a more efficient program.