Vy-An Phan
26159888
cs195-15-aii

Assignment 1

1. Measuring Execution Time: Summation

	Sum, print	Sum, no print	Hardcoded, print	Hardcoded, no print
With O2	19282	2258	18976	2175
Without O2	83374	80239	78731	78417

a) Optimization without printing gives a significantly smaller time because the summation isn't even executed. Since the computer "knows" that the output value of the summation isn't used anywhere else it simply sips over that part.
Optimization with printing is also significantly faster than without optimization because it specifically looks for ways to decrease jumping and loading operations.
Hardcoded is slightly faster than argument passing because the computer can just load a direct value instead of looking for a passed argument.

b) The most accurate computation is hardcoded, printing, no -O2 optimization. With O2 we don't know which operations are going in parallel (or skipped altogether, in the case of no printing). Being able to access a hardcoded value directly also allows us to more "purely" access the sum function. Printing, when done outside of the timing operations, ensures that the end result will be used (so any out-of-order operations don't interfere in our timing).

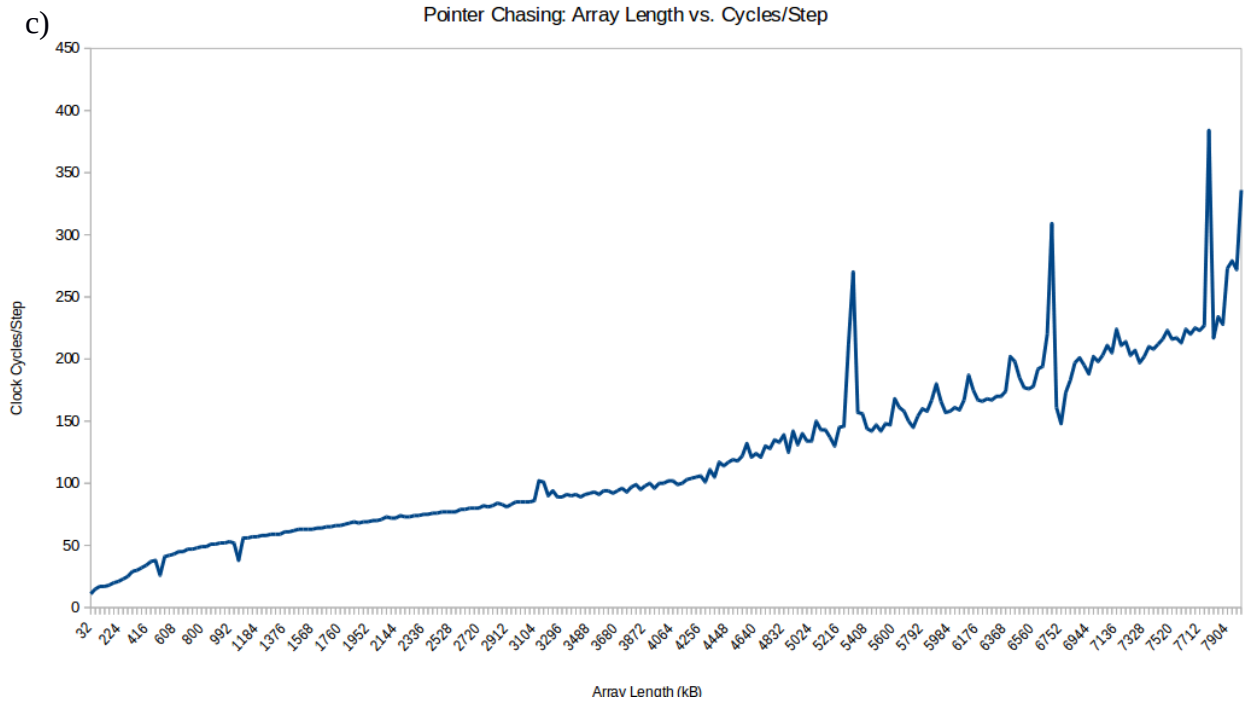
c) `argv[1]` is the value of the argument passed into a function. `atoi()` converts it to an int.

2. Measuring Memory Latency: Pointer Chasing

a) This benchmark measures memory latency by the number of loads (steps).

b) An int array of length N is 4N bytes in memory on this computer (hive20).

c)



d) Since larger arrays require more memory loading, we see a steady increase in cycles per step with array length. Eventually, extremely large arrays will no longer fit in cache and have to be fetched from lower down in the memory hierarchy, to be loaded in the cache in chunks. The extremely sharp peaks happen when excessive loading between main memory and cache is required.

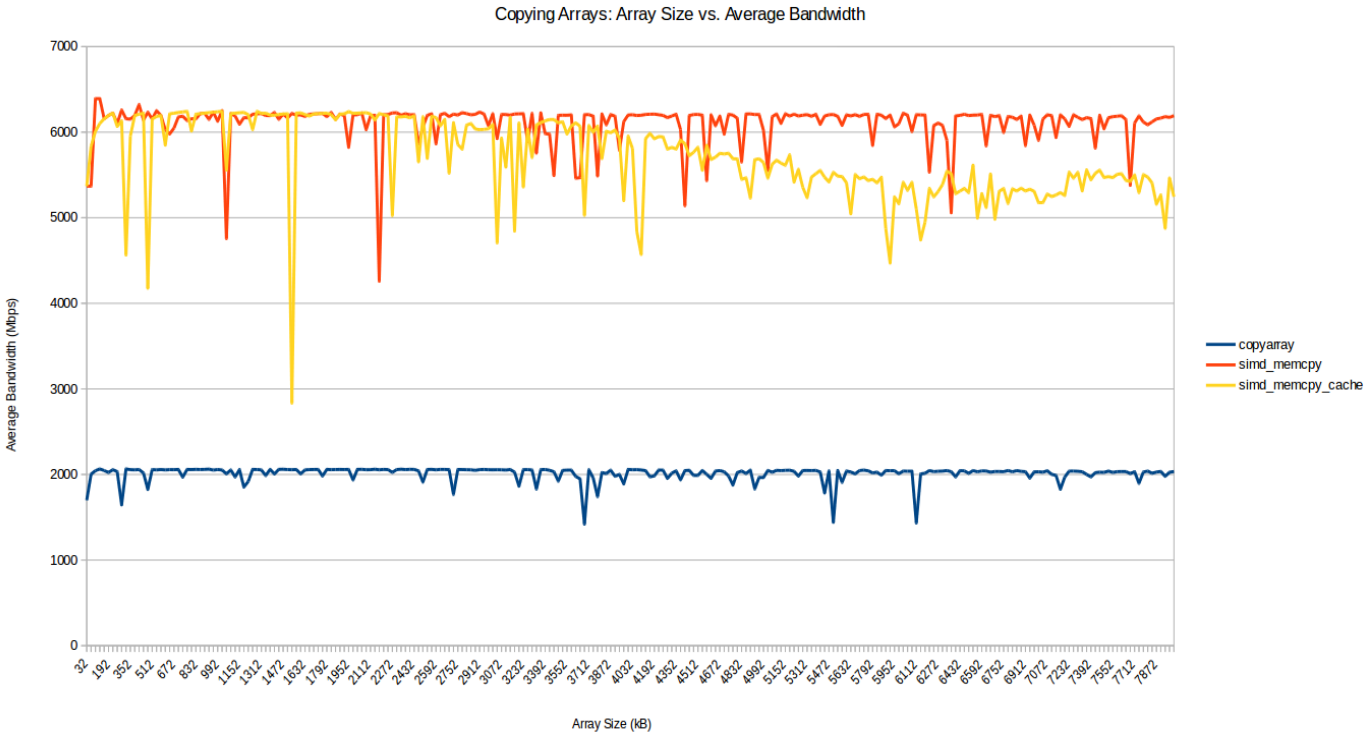
e) No, pipelining shouldn't be affecting our pointer. Due to the way we set up our tests and code, subsequent instructions are dependent on the previous ones. Thus any attempted parallel processing wouldn't work.

f) A modified version of the previous pointer method. Highlighted areas are the new parts pertaining to the unique values problem.
I created another array visited of the same size. When a value in arr is touched, I give that same index in visited a mark. That way, when I am incrementing numDistinct, I know which values I have counted already.

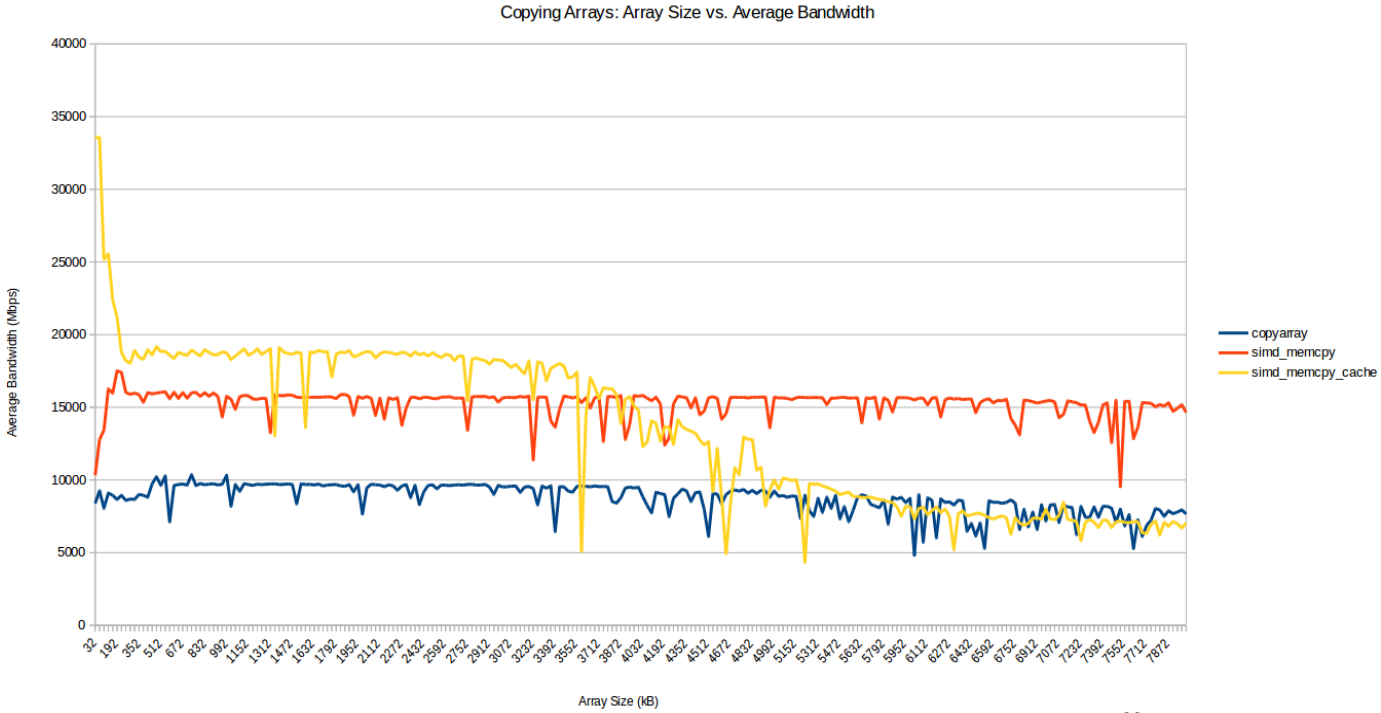
```
int numDistinctValues(int n) {
  int *arr = randIntArray(n);
  int visited[n];                          //to mark already visited values
  int numDistinct = 0;                     //number of unique values
  int p = 0;                                          //same pointer chasing as before
  for(int i=0; i<(1<<20); i++) {
    if(visited[p] != 1) {
      visited[p] = 1;
      numDistinct++;
    }
    p = arr[p];
  }
  return numDistinct;
}
```

3. Measuring Memory Bandwidth: Array Copying

a) Without O2 Optimization



With O2 Optimization



b) My copyArray method and simd_memcpy both have about constant bandwidth, not affected by array size. Naturally O2 optimization makes them faster. simd_memcpy_cache has more dramatic dips because it's entirely dependent on what the cache can hold and thus dependent on the size of the array. All functions display areas of sharp dips in bandwidth. This happens when you run into arrays that require excessive cache loading, or cross page boundaries.

c) Before we warm up the cache, it's full of bad data (junk). Warming it up loads the cache with the type of data we want to use (the array data), making it more likely we won't have to descend through different cache levels.

d) Bad array copying procedures won't utilize the processor to its fullest. There's no point having a very powerful processor if you're only going to end up using small bits and pieces of it.

e) _mm_prefetch fetches specified data line from mem to specified place in cache hierarchy.

_mm_load_si128 loads 128-bits of int data from memory into dst.mem_addr

_mm_stream_si128 stores 128-bits of int data into memory by non-temporal memory hint.

_mm_store_si128 stores 128-bits of int data from specified location into memory.

4. Measuring Flops and IPC: Matrix Multiply

a) Table

	naive_sgemm	opt_scalar0_sgemm	opt_scalar1_sgemm	opt_simd_sgemm
GFLOPS	4.460	8.674	13.037	12.922
IPC	1.149	2.239	3.350	3.372

b) Each value in the resultant matrix is computed from 2 vectors of length n, with n multiplications and $n-1$ additions. This is done n^2 times. Thus the total number of operations is $n^2(n + (n – 1)) = n^2(2n – 1) = 2n^3 – n^2$ operations.

c) See table. $(2n^3 – n^2)/T$

d) Optimization effects within our computers allow for parallel execution of instructions to happen within one clock cycle, using techniques like pipelining, superscalar, etc.

e) Efficiency is usually measured in time or number of cycles, so we can calculate this with #cycles = #instructions / IPC. A higher IPC will reduce the total number of cycles required; however, it can only do so much. If your total number of instructions is too large (courtesy of a bad algorithm) it will eclipse the IPC and result in a huge number of cycles. Thus, high IPCs are not always an indicator of efficient programs.