CS 194 Assignment 1

Nikhil Unni

Fall 2015

1 Measuring Execution Time

O2 Enabled	Printing	Hardcoded number	Number of Ticks
✓	✓	✓	90,646
X	✓	✓	134,656
✓	X	✓	1,026
X	X	✓	77,027
✓	X	X	11,584
X	X	X	102,817
X	✓	X	164,476
✓	✓	X	75,250

Table 1: Tick Timings for Repeated Addition

Explain the discrepancy in timings. (2 points)

The O2 optimization removes unnecessary instructions (along with other optimizations, like using registers), which is why when its coupled with the "no printing" option, leads to the fastest execution times, because it can cut out the for loop completely, since it doesn't contribute to the output. The hardcoded number just adds additional time since there is necessary stdin and an O(n) conversion of string to int.

Which configuration is the most accurate way of timing the computation of the sum? (2 points)

With O2 disabled, printing disabled, and the number hardcoded in. This is because the atoi conversion and stdout printing are not included in the time, but the compiler won't remove the computation completely either.

How does atoi(argv[1]) work? (2 points)

argv[1] refers to the first argument, a string, supplied by the caller of the program (where argv[0] is the program name itself). atoi() converts a string

to an int ("100" $\rightarrow$ 100, for example) by converting each char to an int, and multiplying by some 10^n.

2 Measuring Memory Latency

Explain how this pointer chasing benchmark measures the time between requesting and obtaining a value from memory (2 points)

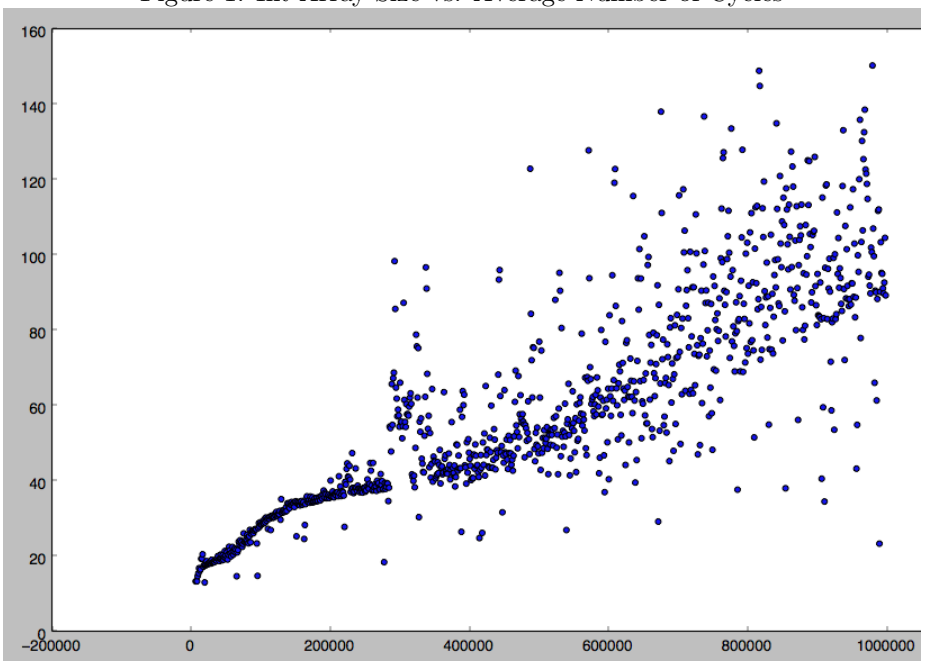
Because the path traveling indices is randomized, ideally we have an equal mix of cache hits and misses when trying to access the next array index, making us truly measure the time to obtain a value from memory.

How many bytes in memory is an int array of length N on your machine? (1 point)

4N

Generate a plot of size in bytes of array vs. average number of cycles to execute a step (32 points).

Figure 1: Int Array Size vs. Average Number of Cycles



As a quick note, the x-axis is measuring the array size, rather than size in bytes,

so the range is [8000,1000000].

Discuss the shape of the curve (4 points)

The point where the spread of the points goes crazy is around 1.2MB (or len(array) = 280,000), which is when the computation blew the cache size. From there, you can sort of see a trend going forward, but the points are more normally distributed around the general trend. The reason for the crazy spread is because you can get very small cycles if you're lucky – for example if A[0] = 2, and A[2] = 0; Worst case, it's a full N-long cycle. When everything could fit on the cache, once you've completed a cycle, every memory access was much faster thereafter.

Do modern out-of-order processors affect our benchmark? (2 points)

No, it does not, since the processor cannot move ahead, since it cannot "guess" what the future value will be. Every instruction is constraint by the immediately previous instruction, and that chain cannot be broken anywhere for parallelization.

BONUS (16 points)

```
int uniqueValues(int *A) {
  int out = 1;
  int fastWalker = A[0], slowWalker = A[A[0]];
  while(slowWalker != fastWalker) {
    slowWalker = A[slowWalker];
    fastWalker = A[A[fastWalker]];
    out++;
  }
  return out;
}
```

Essentially, the randomized array is just a linked-list (where the next value is the value at the current index) with a cycle, and we need to determine the cycle size. So we just have a slow walker, which moves traverses the list one at a time, and a fast walker that traverses the list two at a time, and the two are guaranteed to meet at 0, regardless of whether it's even or odd. If the cycle length is even, the fast runner will be at 0 when the slow runner is halfway and when its finally reached zero, where they'll meet. If the cycle length is odd, then the runner will run "past" 0 twice, the second time, it finally reaching 0, since an odd number times 2 becomes even. Since it's moving twice as fast as the slow runner and takes twice as many cycles to hit 0, the slow and fast runners will yet again meet at 0.

3 Measuring Memory Bandwidth

Plot the relationships between the average bandwidth and the total size of the array (24 points)

Figure 2: my_memcpy

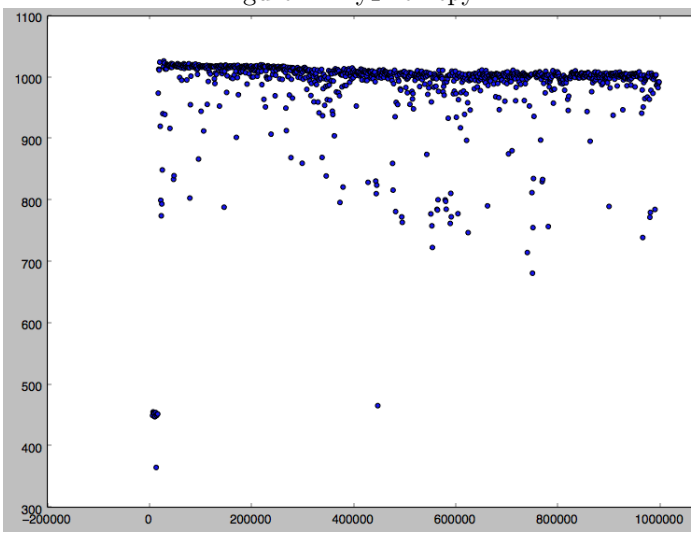


Figure 3: simd_memcpy

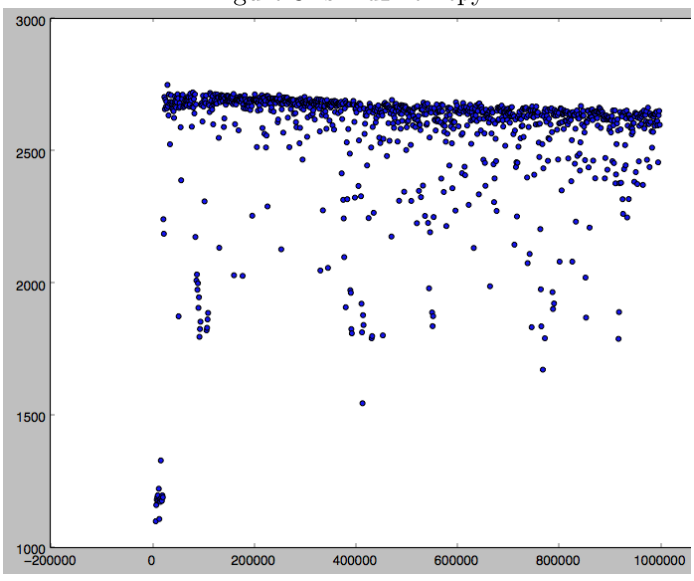
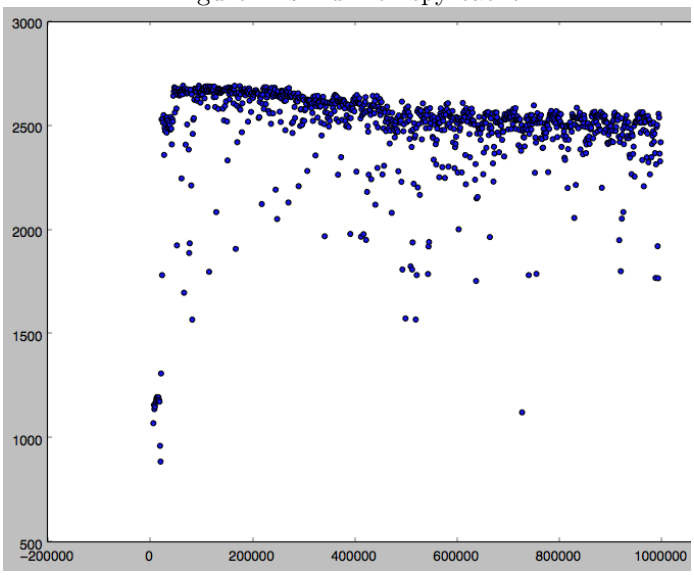


Figure 4: simd_memcpy_cache



Discuss the shape of the curve (4 points).

The three curves generally look the same, except that simd_memcpy_cache seems to decrease in bandwidth as the array size increases slightly more than the other two copy methods. There are no meaningful inflection points that I can see in any of the graphs. However, both simd_memcpy and simd_memcpy_cache perform about 2.7x better than my naive my_copy, that copies over each byte one by one.

Explain why it's necessary to "warm up the cache" (2 points)

By copying over the array a few times, we add each of the elements from the src to our cache, so that our copy method has mostly cache hits when it's trying to find values in memory. This will speed up the copy process.

Explain why an inefficient array copying procedure would yield an inaccurate measure of maximum bandwidth

Because the procedure is inefficient, we're not truly measuring the bandwidth when all resources are used to do the operation. Instead, we get something slower that's not the fastest *possible*.

Find all the different SSE intrinsic calls (6 points)

_mm_prefetch(p, _MM_HINT_NTA) : Prefetches line of data from memory at some address with "non-temporal allignment," which just suggests that we're

not going to go back to that place in memory any time soon, so no need to expect we will.

_mm_load_si128 : Loads 128 bit value

_mm_stream_si128 : Load 128 bit value, and don't go through cache. Just retrieves value.

_mm_store_si128 : Store 128 bit value in memory.

4 Matrix Multiply

Produce a table recording the Flops and IPC of each matrix multiply

	Flops (GFlops)	IPC (Instr/Cycle)
naive_sgemm	1.875	0.9999
opt_scalar0_sgemm	3.557	0.9999
opt_scalar1_sgemm	4.776	0.9999
opt_simd_sgemm	5.006	0.9999

Table 2: Flops and IPC of each matrix multiply

How many floating point operations for a matrix multiplication of two square matrices of size n? (3 points)

Each cell in the output is the result of two inner-products of vectors of size n. This is n multiplications, and $n-1$ additions. Since there are n^2 total cells in the output, this means there are $n^2(2n-1)$ total operations for the matrix multiplication.

How can some implementations have IPC higher than 1? (2 points)

I didn't get any IPC higher than 1, for some reason (running on the lab computers!). But this is possible through parallelism, SSE instructions, and out-of-order execution. All of these techniques lower the number of cycles for the same number of instructions, possibly leading to an IPC > 1.

How can a process with a higher IPC also have lower Flops? (2 points)

Again, this wasn't happening for me, no matter how many times I tried to run it. But the two aren't necessarily coupled – some instructions might take more time. For example, some processors with complicated instruction sets can have high IPC and low Flops, while a more bare-bones architecture can have the opposite.