(c) [5 points] Please describe what needs to be true about array `A` to reach the maximum possible SIMD utilization asked in part (b). (Please cover all cases in your answer.)

For every 32 consecutive elements of `A`, every element should be lower than 33 (`if`), or greater than or equal to 33 (`else`). (NOTE: The solution is correct if both cases are given.)

(d) [13 points] What is the *minimum* possible SIMD utilization of this program?

$\frac{1025}{1568}$.

Explanation:
`Instruction 1` is executed by every active thread ($\frac{1025}{1056}$ utilization).
Then, part of the threads in each warp executes `Instruction 2` and the other part executes `Instruction 3`. We consider that `Instruction 2` is executed by α threads in each warp (except the last warp), where $0 < \alpha \leq 32$, and `Instruction 3` is executed by the remaining $32 - \alpha$ threads. The only active thread in the last warp executes either `Instruction 2` or `Instruction 3`. The other instruction is not issued for this warp.

The minimum SIMD utilization sums to $\frac{1025+\alpha\times 32+(32-\alpha)\times 32+1}{1056+1024+1024+32} = \frac{1025}{1568}$.

(e) [5 points] Please describe what needs to be true about array `A` to reach the minimum possible SIMD utilization asked in part (d). (Please cover all cases in your answer.)

For every 32 consecutive elements of `A`, part of the elements should be lower than 33 (`if`), and the other part should be greater than or equal to 33 (`else`).

(f) [10 points] What is the SIMD utilization of this program if `A[i] = i`? Show your work.

$\frac{1025}{1072}$.

Explanation:
`Instruction 1` is executed by every active thread ($\frac{1025}{1056}$ utilization).
`Instruction 2` is executed by the first 33 threads, i.e., all threads in the first warp and one thread in the second warp.
`Instruction 3` is executed by the remaining active threads.

The SIMD utilization sums to $\frac{1025+32+1+31+960+1}{1056+32+32+32+960+32} = \frac{2050}{2144} = \frac{1025}{1072}$.

10 Memory Hierarchy [40 points]

An enterprising computer architect is building a new machine for high-frequency stock trading and needs to choose a CPU. She will need to optimize her setup for *memory access latency* in order to gain a competitive edge in the market. She is considering two different prototype enthusiast CPUs that advertise high memory performance:

(A) Dragonfire-980 Hyper-Z

(B) Peregrine G-Class XTreme

She needs to characterize these CPUs to select the best one, and she knows from Prof. Mutlu's course that she is capable of reverse-engineering everything she needs to know. Unfortunately, these CPUs are not yet publicly available, and their exact specifications are unavailable. Luckily, important documents were recently leaked, claiming that the two CPUs have:

- Exactly 1 high-performance core
- LRU replacement policies (for any set-associative caches)
- Inclusive caching (i.e., data in a given cache level is present upward throughout the memory hierarchy. For example, if a cache line is present in L1, the cache line is also present in L2 and L3 if available.)
- Constant-latency memory structures (i.e., an access to any part of a given memory structure takes the same amount of time)
- Cache line, size, and associativity are all size aligned to powers of two

Being an ingenious engineer, she devises the following simple application in order to extract all of the information she needs to know. The application uses a high-resolution timer to measure the amount of time it takes to read data from memory with a specific pattern parameterized by `STRIDE` and `MAX_ADDRESS`:

```
start_timer()
repeat N times:
       memory_address <- random_data()
       READ[(memory_address * STRIDE) % MAX_ADDRESS]
end_timer()
```

Assume 1) this code runs for a long time, so all memory structures are fully warmed up, i.e., repeatedly accessed data is already cached, and 2) N is large enough such that the timer captures ***only*** *steady-state information.*

By sweeping `STRIDE` and `MAX_ADDRESS`, the computer architect can glean information about the various memory structures in each CPU.

She produces Figure 1 for CPU A and Figure 2 for CPU B.

Your task: Using the data from the graphs, reverse-engineer the following system parameters. If the parameter does *not make sense* (e.g., L3 cache in a 2-cache system), mark the box with an "X". If the graphs provide *insufficient information* to ascertain a desired parameter, simply mark it as "N/A".

NOTE 1 TO SOLUTION READER:
This analysis provides insufficient information to determine the line size of the cache(s). This is because we are always 'striding' in power-of-two values starting at address 0. This means that either our access pattern entirely fits within the cache (in which case we observe constant latency since the cache is already warmed up), or the access pattern is striding using values larger than the line size, so we never see two accesses to the same cache line.

NOTE 2 TO SOLUTION READER:
This problem is not actually that hard.
The way to think about these plots is that each point is an access pattern. The easiest points to understand are those that result in an access pattern of {0, 0, 0, 0, ...} and randomly from {0, A}, where A is your stride. Just by looking at those you should be able to determine pretty much everything.
The access latencies and sizes are trivial to read off if you understand what the test code is trying to do. The associativities are nuanced, but you can tell from the aforementioned access patterns by simulating carefully.
If you want to go all-in, you can compute probabilities: if I access {0, A} then 50% of the time I'll hit and 50% miss. It's easy to get the cache latencies, so I can just match points from there on :)

(a) [15 points] Fill in the blanks for Dragonfire-980 Hyper-Z.

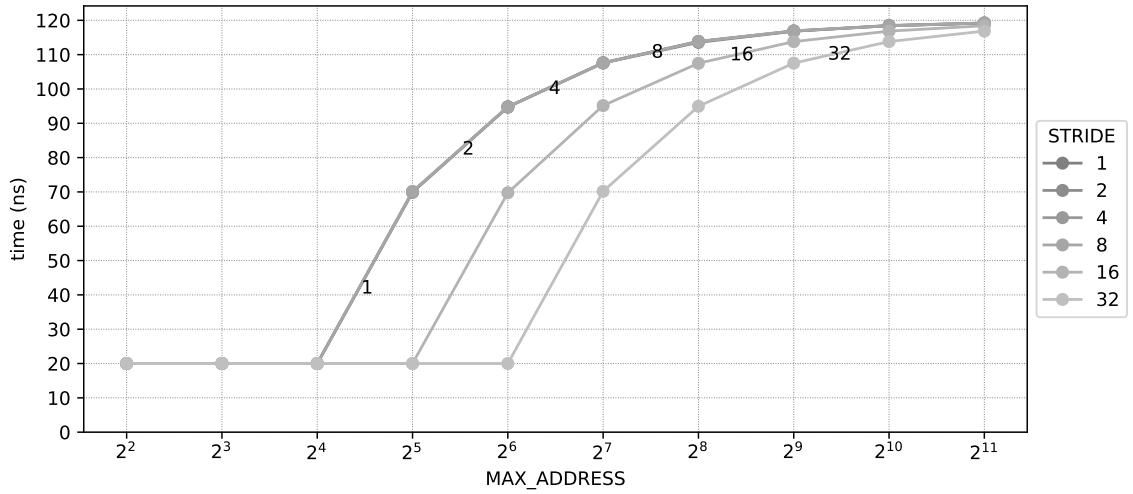

Figure 1: Execution time of the test code on CPU A for various values of *STRIDE* and *MAX_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, and 8 overlap in the figure.

Table 1: Fill in the following table for CPU A (Dragonfire-980 Hyper-Z)

System Parameter	**CPU A: Dragonfire-980 Hyper-Z**			
	L1	L2	L3	DRAM
Cache Line Size (B)	N/A	N/A	N/A	N/A OR X
Cache Associativity	2	X	X	X
Total Cache Size (B)	16	X	X	X
Access Latency from (ns) [1]	20	X	X	100

[1] DRAM access latency means the latency of fetching the data from DRAM to L3, *not* the latency of bringing the data from the DRAM all the way down to the CPU. Similarly, L3 access latency means the latency of fetching the data from L3 to L2. L1 access latency is the latency to bring the data to the CPU from the L1 cache.