MIDTERM PREP:

Results: 90/100; in the top 5 in class, including 3 TAs

Part 1: Overview

1.
    a. What is the von Neumann model?
        i. The von Neumann model is a computer architecture and design that shows the basic components of a computer. It has four components:
            1. CPU (connected to memory and I/O by data and address bus, as well as by control bus
            2. Memory
            3. Bus(ses)
            4. I/O
    b. What does the vN model abstract? That is, what information in the target architecture is ignored?
        i. There was no parallelism and a uniform operation time (flat memory and memory references are not much longer than register fetches) in a vN model, so the key to high performance would be:
        - Data fits into memory
        - Minimizing the number of operations
    c. What is a good performance model for the vN model? That is, what is a good way to express potential application performance on an abstract vN machine?
        i. CPS x instructions/cycle * CPI
        ii. A good model for this could be a Roofline model, as this is useful for applications that are memory-intensive. This uses bound and bottleneck analysis, and uses arithmetic intensity against attainable GFLOP/s to find if the code is computation-bound or memory-bound/
        iii. Additionally, you can use basic performance model such as by finding the execution time with instruction count, clock frequency, and CPI to find how performance is affected
2.
    a. We examined performance change over time (1970s to 2010s) and found that it had multiple components, e.g., those performance gains due to improvements in architecture and those not. What improvement(s) is(are) independent of architecture?

i. An improvement that is independent of architecture is developing memory conscious programs. For example, if you are storing data that will be accessed sequentially, and the matrix you are working with is larger than your cache size, you will constantly have cache misses when fetching the data. If you are indexing into the array column-wise for a programming language that follows a row-wise array storage convention, you will have a cache miss on each index and pull the cache entry size of data in that row. However, assuming that your matrix is very large, you may not ever reap the benefits of this attempt at exploiting spatial locality and will constantly see cache misses. The improvement would be to index into the array in the same way that the programming language stores array data.

b. Give an example of an architectural improvement.
   i. An architectural improvement would be the introduction of caches, as well as a memory hierarchy. This exploits temporal and spatial locality by storing data that has a high chance of being referenced again close to the CPU and only accesses memory far from the CPU on rare occasions.

3.

a. What is Moore's Law?
   i. The speed and capabilities of a computer can be expected to double every two years, as result of an increase of transistors that can fit into a microchip. In actuality, it's that we have double the functions per area, which is a more technical understanding of Moore's law.
b. What are the implications of Moore's Law on architecture improvements?
   i. The implication of Moore's Law on architecture improvements is that we are figuring out how to design smaller and accurate transistors. This has the benefit of work per cycle increasing by 1.4x.
c. On non-architecture performance?
   i. Cost of computing has fallen drastically, as we are able to produce more transistors and components at a lower cost.
d. How is it varying with time? (is it dead?)
   i. Moore's Law is now facing the power wall and frequency wall. The energy produced by our chips can only grow so much, and growing at the pace theorized in Moore's law will result in chips needing to dissipate the same amount of energy that is found on the surface of the Sun. This shows that there is no longer much benefit for adding more transistors to a core and rather the efficiency of computing will increase by adding more cores.

4.

a. What is Dennard Scaling?
   i. Dennard Scaling is the physical effects of Moore's Law. It states that as the number of transistors increase (doubling value in area of an IC), and

their size decreases, we should see that the power density stays constant. Dennard observed that voltage and current should be proportional to the linear dimensions of a transistor: as transistors shrank, the necessary voltage and current did as well. Thuis, power is proportional to the area of the transistor. Power = alpha * CFV^2; voltage reduced due to the size of transistors decreasing means that we can operate at higher frequencies for the same power.

b. What are the effects of Dennard Scaling on how computers have evolved (1990s - 2010s)?
    i. This fueled performance growth from the 1990s to early 2000s but broke down in the mid-2010s, leading to major shifts in computer architecture.
        1. 1990s to 2000s was the golden age of frequency scaling. This period had higher clock speeds, going from MHz to GHz. Faster and smaller transistors meant better performance per core, so single-core technology was steadily increasing in performance. This period still had manageable power since the power density remained constant.
        2. From the 2010s, the power and frequency wall became huge problems. Because it was harder to dissipate power, the capacitance increased and then power consumption appeared to grow exponentially. The heat dissipation limit meant that chips couldn't increase clock speeds without consequences in long-time behavior and reliability. This meant for the end of frequency scaling, as CPUs stopped getting faster and the frequency is currently plateaued at 3-4 GHz, and instead of focusing on single-core computers, we started focusing on multi-core architectures.

c. How have the effects of Dennard Scaling varied during that time?
    i. The first period: 1990s - 2000s had the expected frequency scaling. This meant that the frequency increased as the voltage necessary for a transistor decreased. After this, in 2000s - 2010s, we saw that the frequency scaling no longer holds and that we plateaued in increase of frequency.

d. How is Dennard Scaling different from Moore's Law?
    i. Moore's Law moreso focuses on the development of transistor technologies, and said that we should see double the amount of transistors on a chip every 180-24 months. Dennard Scaling instead focuses on the effects of a smaller transistor, stating as the voltage decreases (with capacitance ?) for a smaller transistor, then for the same amount of power, we should be able to increase the frequency as the inverse scale. This is why it is referred to as frequency scaling.

5.

a. In general, performance can be improved by decreasing latency and improving bandwidth. We claimed that this translates into parallelism and locality. Explain. Or dispute if you have a good argument. What does efficiency have to do with this?
    i. I believe that efficiency is the reason/method for decreasing latency and improving bandwidth. Locality is used to keep data close to its home node (horizontal locality) and to also keep data close to the CPU (vertical locality). For vertical locality, it is a more efficient model to keep frequently accessed data closer to the CPU in the L1 cache than for the same data to be further away from the CPU. This means that less time will be spent accessing the data when we need it, because it won't be as far away. For parallelism, this helps increase bandwidth because more work gets done per unit time when multiple tasks execute in parallel. This is more efficient because it will increase resource utilization

b. Give an example (an application) that includes all three components (parallelism, locality, efficiency). That shows how performance of that application can be improved (with parallelism and locality) and how that improvement is limited (by lack of efficiency).
    i. An example that includes all three components is matrix matrix multiplication (MMM). It's performance can be improved with parallelism and locality in the following ways:
        1. Parallelism: each computation of the output matrix is independent, and thus highly parallelizable. One can divide the computation among cores with pthreads and OpenMP.
        2. The memory accesses can be done in a memory-aware manner that has efficient caching and accessing of elements. This can exploit temporal locality and spatial locality because the same elements in the input matrices are used multiple times, and same with storing the matrices in the order that best suits the programming language which you'll be accessing them exploits spatial locality.
    ii. The improvement will be limited in terms of parallelism because of the overhead of synchronization (waiting for other threads). There can also be bottlenecks with memory bandwidth, if the data movement from memory to to cache/registers is slow; Amdahl's Law also shows that there are diminishing returns of parallelism, and there is always the looming power wall that will prevent tasks without degradation.

6.

a. What is the power wall?

i. The power wall is a limitation for increasing the frequency scaling for computation. The current wattage per unit area has a cap, and at the rate before 2010s, it would have been projected to have the same energy per area of a nuclear reactor by 2003. Because the power consumption seems to now increase despite efforts to have a smaller transistor, we are unable to have faster computers without higher energy consumption.

b. How has it affected computer architecture? Give at least two examples of this.

1. Rather than focus on single-core technologies, there is more of a push for studying and developing multi-core architectures for performance in parallelism.
2. There is now a rise in specialized hardware. Before the power wall was hit, general-purpose CPUs were able to handle most workloads. Now, specialized accelerators are developed to offload power-intensive tasks. An example of this is GPUs used for Deep Learning.

7.

a. The most basic way to evaluate performance improvement is to measure wall clock time. But a crucial question is knowing how well you are doing, that is, what fraction of potential improvements you have actually achieved. Explain how you could know this. Include how you account for the characteristics of your computer.

i. A way to determine how well we are doing, we can measure our measured performance against the theoretical best-case performance of the system. We can evaluate a speedup by finding the improvement from the original execution time as compared to our optimized execution time, and additionally we can measure efficiency by seeing our achieved performance against the theoretical peak performance. Having a low efficiency indicates bottlenecks. The way we can account for the computer characteristics is by comparing achieved performance against the hardware limitations. For example, a roofline plot can be used to find the attainable performance to find which optimizations will potentially be beneficial; if a computation-heavy program runs slower than the peak FLOP/s, there is likely a stream bandwidth bottleneck that we can investigate memory access pattern or cache misses after identifying.

8.

a. What applications lend themselves to high performance programming? Why?

i. Things with large data sets, intensive computation, and real-time processing are the main characteristics for applications that need HPP.

1. Scientific computing and simulations:
    a. Require lots of FLOP computations

b. Need to process large datasets efficiently with minimal memory bottlenecks
c. Often run on supercomputers with GPUS and multi-node clusters

2. ML and DL
    a. Relies on massive matrix multiplication
    b. Benefits from GPUs TPUs and tensor cores for acceleration
    c. Need low-latency memory accesses for handling large datasets
3. HFT and Financial Computing
    a. Requires microsecond-latency to get the best trade
    b. Low-latency networking and cache-optimized algorithms
    c. Relies on FPGA acceleration and SIMD instructions for real-time computations

9.

a. What is arithmetic intensity?
    i. Arithmetic intensity is the number of floating point operations per memory access (also referred to as computational intensity)
b. What types of applications have high and low computational intensity?
    i. High intensity (CPU bound), large number of computations per fetch, often a plus of exploiting spatial and temporal locality:
        1. N-body particle methods
        2. Dense Matrix (BLAS3)
    ii. Low intensity (Memory bound), small number of computations per fetch, so often COLD/compulsory misses:
        1. Sparse Matrix (SpMV)
        2. Structured Grids (Stencils, PDEs)
c. Why is knowing the applications' AI critical for obtaining high performance?
    i. By doing roofline plots, we can figure out where optimizations will be potentially beneficial. We plot it as the minimum between FLOP/s and AI*bandwidth. If the secondary is the minimum, then it is stream bandwidth that is holding back the attainable FLOPS/s performance. If we know the ai, then we can figure out how to optimize

10.

a. CPE: cycles per element
b. Measure performance with CPE by finding the time it takes for a code section to run, convert to the effective cycles time. Then, divide by the number of elements iterated upon (usually a function of loop indices).
c. This is useful for evaluating

Part 2a: Review memory hierarchy

15.
CPU generates a memory access request
Cache controller intercepts request, checks to see whether the data is in cache
ON HIT: cache accessed and data set to the CPU (data in cache)
On MISS: memory is accessed, copy of data put into cache, and data is sent to CPU (data not in cache)

16. The "simple" model of memory hierarchy performance is average memory access time:
Hit rate * Hit time + miss rate * miss penalty
Because the miss penalty usually is much larger than the hit time, we want to get a hit rate as close to 1 (inversely, this brings the miss rate as close to 0 as possible). A fraction for hits to target is 100%, but even with an example of hit time = 2ns and miss penalty = 100ns, having a hit rate of 99% results in an average memory access time of 2.98 ns, which is a 50% slowdown from the ideal.

Amdahl's Law: formula that shows how much faster a task can be completed when more resources are added to the system. (overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used).

17: a cache block is a unit of data transfer between the main memory and the cache, it is a fixed-size chunk of data.
Large cache blocks are generally helpful because the blocks are pulled from contiguous memory, so larger cache blocks means that whenever an item is accessed, its surrounding items will be available in the cache. This takes advantage of spatial locality.
The size is limited by the distance for the cache to the CPU. You have a large cache, but that memory takes space and will need to be further away from the CPU to do so (cache has a uniform access time, regardless of where the start and end of the cache is relative to the CPU). The two conflicting goals is that you want to feed data to a CPU at the CPU's operating frequency (small cache near the CPU), and you want to maximize the hit rate (large cache). Thus, conflicting tradeoff problems.

18. We expect only a small amount of data to be accessed at a time (temporal locality) and want this to be supplied to the CPU as fast as possible. However, there is still data that may need to be accessed, and although it might not be as frequently as some other data, it will be faster to access from a higher level in cache rather than fetching from main memory each time.

19. The four types of cache misses are:
1. Capacity miss: happens when the working set of data (active cache blocks) is larger than the cache (a replaced block will be later accessed again);
2. Compulsory miss: happens when first accessing the block of data
3. Conflict (collision) misses: multiple memory locations mapped to the same cache location; occurs when the level of cache is large enough, but multiple data objects map to the same block
4. Coherence misses (invalidation): another process updates memory so the cache is no longer valid to the working program (the cache is filled with another processes' stale data).

Part 2b: Programming with memory

20. Programmer's view of memory (typical programmer model): memory is a large contiguous array of bytes, each with a unique address. Programmers can store and retrieve data by referencing these addresses, which forms the basis of how programs interact with data and instructions.
Arrays are mapped contiguously in memory for this model. It depends on programming language, but 2d arrays are stored either in row-major or column-major format. The memory locations that are accessed are computed as either row*rowlen + col, or col*collen + row.

21. Fully associative cache mode, block size 64B, what is the cache behavior (hit rate) of the two snippets. Describe the memory reference traces

- Fully associative cache: allows each block to map to any portion of memory, most flexible with the replacement policy, as well.
- Assuming size of an int is 4B; block can hold 16 integers
- Arrays stored row-major order in C/C++
-

```
for (i = 0; i < N; i++)
        sum += a[0][i];
```

- The elements here are contiguously, so since each cache block holds 16 integers, it will have a ratio of 1 miss for every 15 hits: hit rate is 15/16 which is 93.75%

```
for (i = 0; i < N; i++)
        sum += a[i][0];
```

- The elements here are not contiguous, so it will depend on if there are hits or not if N is smaller than if the column length has more than 16 elements. If more than 16, then you will have a hit rate of 0%, because it will miss and pull the next 16 integers but then your next access will still be a miss;

22: MMM loop order matters because of how the matrices are stored in memory. MMM takes the row of the first matrix and multiplies it element-wise with the column of the second matrix, summating and storing into an index in the third matrix (ijk). This is the heuristic approach, but it can be approached in different ways as well. You can have a fixed value for your A matrix and multiply this against all of the row values in B, adding this to the existing values in the row for matrix C (ikj). A third method would be to iterate through A column-wise with a fixed B value, and add to the values in C column-wise (jki).
The following performances are associated with misses per inner loop iteration
ijk: A = 0.25; B = 1.0; C = 0.0 – 2 loads and 0 stores, miss/iter = 1.25
ikj: A = 0.0; B = 0.25; C = 0.25 - 2 loads and 1 stores, miss/iter = 0.5
jki: A = 1.0; B = 0.0; C = 1.0 - 2 loads and 1 stores, miss/iter = 2.0

23: blocking is a technique used to increase the performance of a program, it blocks a certain amount of a procedure and does them incrementally. The idea is that this should increase program performance because the data is more likely to be in cache. Because you are improving spatial and temporal locality.

For MMM, it's like doing a BxB mini MMM. you add an additional layer of MMM with 2 sets of ijk. The outer layer will have increments of B while the inner layers increment by 1 and cannot exceed the B outer

boundary of its block. The blocking hopes to reduce the cache misses. If B^2 is << cache size, then the submatrix multiply has no more than O(B^2) cache misses.

This works because it increases the temporal locality; once a small block of A and B are loaded into cache, it is reused multiple times before being evicted. This also keeps data closer together in memory, leveraging spatial locality.

The performance relationship to cache and block size is that if the matrices are too large to fit in the cache, a naive MMM will have cache misses frequently. Blocking ensures that submatrices fit within cache, improving reuse and reducing memory bandwidth overhead.
The optimal block size depends on the cache size. If it is too small, performance benefits are limited; if too large, might not fit into the cache and start thrashing.

24:

Part 3a: Intel Assembly Language

25: there are 16 general-purpose registers, each 8 bytes (64 bits) wide

1. Rax
2. Rbx
3. Rcx
4. Rdx
5. Rsi
6. Rdi
7. Rbp
8. Rsp
9. R8-r15

Can access them at different sizes: 8 bytes (r*), 4 bytes (e*), 2 bytes (*), 8-bytes (*l) low byte and (*h) high byte
Also vector registers

26: the addressing modes: lots of complex ones, rather than only displacement. There are also push, pop, lea, leave (?)
There are several addressing modes: immediate, register, direct (absolute), indirect (register), indexed (register + offset), scaled indexed, and base + scaled indexed + displacement is move rax, [rbx + rcx*4 + 16] (for array accesses)

27: parameters are passed in function calls and are put into the gp-registers. The first 6 integer/pointer arguments get placed in rdi, rsi, rdx, rsx, r8, and r9. Floating point arguments are put into the XMM registers (xmm0-xmm7), and any extra arguments are pushed onto the stack from right to left (so that they will be popped out as expected).

28: flow control is implemented with jump instructions:
Unconditional jumps: jmp
Conditional jumps: je, jne, jl, jg (based on condition codes)

Function calls: call function (use stack for return address)
Returns: ret

Condition codes are flags in the RFLAGS register
Zero flag
Sign flag
Carry flag
Overflow flag

29: what are conditional moves, when are they useful or not useful
Conditional moves are branchless alternatives to if conditions
This would be: cmp eax, ebc
Cmovg eax, ecx ; if (eax > ebx) eax = ecx
You use these because branches are distrubptive to instruction flow through pipelines, and conditional moves do not require control transfer

Bad cases for conditional moves are:

- expensive computations, because both values get computed (for the assignment), thus it makes sense when the computations are very simple.
- Risky computations (if address causes page fault / illegal address)
- Computations with side effects (x*=7 : x*=3)

Useful for assignment if the computation isn't hard and doesn't have side effects

Part 3b: Optimizing program performance

30: Big-O describes how performance scales with input size N. The limitation is that it ignores constants and only provides an upper bound on performance; additionally it will ignore practical hardware effects (cache and parallelism) and might not be meaningful for small N in which overhead dominates.

31. Generally useful optimizations for all programs:

- Code motion: reduce frequency in which computation is performed
    - Especially when it produces the same result, and if you can move code out of a loop
- Strength reduction
    - Replacing a costly operation with a simpler one (shift instead of mul/div of pow2)
    - Machine dependent
- Common subexpression elimination
    - Reuse portions of expressions

32: optimization blockers are codes that make it hard for the compiler to recognize and trust that it can optimize the code. For example, memory aliasing -> two pointers designating the same memory location.

33. Modern CPUs and ILP (instruction-level parallelism):

- Execution limited to how fast we can feed data to the functional units
    - Fetch latency, branch misses, data hazards, and overhead
- Function units are the pipeline: integer/branch, general integer, fp add, fp mult/div, load, store

- Depth is 6? How many stages shown in the pipeline image is 6.
- Latency: not sure
- Control floor implemented as instruction control. There is a retirement unit and inside is the register file, this goes to instruction decode and detch ccontrol, which the fetch control sends an address to the instruction cache and the instruction decode receives instructions from the instruction cache.

34: Loop unrolling in necessary for optimization because of the conditional branch that is taken for each time we increment the iterator and have to go back to the loop. By doing two computations and iterating the incrementer by 2, we get 2x more useful work per iteration, which will have less cmp and jg instructions and less of a delay while waiting
This reduces branch overhead and improves vectorization. This has diminishing returns with modern CPUs, because if a program is memory-bound, unrolling won't help much. Also, if the CPU can already issue instructions in parallel (out-of-order execution) then unrolling too much may provide no extra benefit.
Code scheduling is reordering instructions to maximize CPU efficiency. Goal is to minimize stalls, hide instruction latencies, and increase ILP.
Loop unrolling helps with this because it provides more independent instructions and can allow the CPU to rearrange them for better performance. Additionally, when combined with SIMD/vectorization, it enables multiple operations per cycle

35: accumulator problem: accumulators can have dependencies, sequential dependencies. Having separate accumulators introduces two streams of operation. So the overall performance of N elements with D cycles latency/op should have CPE = D/2. Unrolling with separate parallel accumulator registers will allow a 2x speedup and breaks sequential dependency. This can be utilized in loop unrolling for additive loops to accumulate all at the end.Also, reassociative computation can help.

36: branches are hard because mispredictions stall the CPU and its hard to predict irregular code. Also introduced control flow changes, reduces ILP
Yes it can kill with high misprediction rates having wasted works and pipelines flushed.
Use conditional moves instead of branches, as well as loop unrolling and vectorization and branchless arithmetic.
It might not always be faster if a branch is highly predictable, and because branchless operations always execute both paths,, it could increase power consumption and code complexity.

Part 4a: Intro to SIMD and vector processing

37: The flynn computer architecture taxonomy has four:
1. SIMD: Single Instruction Multiple Data - broadcasting (
2. SISD: single instruction single data - serial code
3. MISD: multiple instruction single data
4. MIMD: multiple instruction multiple data

38. What is standalone SIMD architecture?
- Single instruction sequencer, multiple datapaths

- Single instruction operates on multiple data elements simultaneously enabling parallel processing for data-intensive tasks
- Exploits data-level parallelism by allowing a single instruction to perform the same operation on multiple data elements concurrently
- Can have direct communication with neighboring PEs

What does the controller do?

- Broadcasts instructions to many datapaths

The PEs?

- Element-wise computations with the single instruction (can have activity control where condition computed independently in each PE and only PEs with correct condition store the result)

How is conditional execution done?

- Masks, always used to broadcast

39. What is dataparallel programming? In your favorite data parallel language (or pseudocode) write SOR.

SIMD is an example

The idea of data parallel programming is to operate on collections of data as units.

- Operations perform in parallel on each element of data structure
- Single thread of control
- Elementwise on collections: on each element of a collection
- Scalar-elementwise: on scalar plus collection (single scalar op with each element)
- Within a colleciton of elements: reduction, broadcast (elements are combined into a scalar, distributed to many elements)
- Reordering elements in a collection: shifts rotates permutations

pseudocode , write SOR

```
For iterations from 0 to max_iterations; iterations++:
    For i = 1 < N-1; i++:
        For j = 1 to N-1; j++:
            A[i][j] =a[i][j] - 0.25 * ( a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1] )
```

40. Dataparallel checklist. For each of the following explain what it is

1. Map vector-vector: element-wise operations on two vectors (c[i] = a[i] + b[i])
2. Map scalar-vector: apply scalar operation to all elements of a vector (a[i] *= 2)
3. Reduce combine all elements into a single value: sum = sum [i]
4. Broadcast: copy scalar value to all elements of a vector (a[:] = x
5. Conditional execution: using masks for selective computation (ternary a[i] > 0
6. In-vector data movement: shuffling or permuting within a vector b[i] = a[i+1]
7. Non-local memory references - scatter/gather: load/store non-contiguous memory elements b[i] = a[index[i]]

41. What is a vector architecture?

SIMD-like design where instructions operate on vectors instead of scalar

Different from standalone SIMD architecture such that SIMD has fixed PEs whereas vector architectures operate on register-sized vectors (AVX)

- Different from standard CPU core such that CPUs rely on ILP for parallelism where aws vector processors use data-level parallelism by working on entire vectors at once