Advanced GPU Assembly Programming Exercise Manual

A Technical Reference for NVIDIA and AMD Architectures

Advanced GPU Assembly Programming Exercise Manual

A Technical Reference for NVIDIA and AMD Architectures

First Edition

Gareth Morgan Thomas
Auckland, New Zealand

Published by Burst Books
Auckland, New Zealand

Preface

This exercise manual accompanies the main text and is intended to reinforce understanding through applied practice. Each chapter corresponds directly to the material in the primary volume, offering exercises that focus on problem-solving, technical reasoning, and conceptual depth.

No answers are provided. The intent is to encourage independent thought and mastery of the material through careful analysis and experimentation. Readers are encouraged to verify results through coding, simulation, or additional research where relevant.

By working through these exercises, students and professionals alike will deepen their understanding of GPU architectures, assembly-level programming, and performance optimization—transforming theoretical insight into practical expertise.

Welcome to *Advanced GPU Assembly Programming Exercise Manual*.

About the Author

Gareth Morgan Thomas is a qualified expert with extensive expertise across multiple STEM fields. Holding six university diplomas in electronics, software development, web development, and project management, along with qualifications in computer networking, CAD, diesel engineering, well drilling, and welding, he has built a robust foundation of technical knowledge. Educated in Auckland, New Zealand, Gareth Morgan Thomas also spent three years serving in the New Zealand Army, where he honed his discipline and problem-solving skills. With years of technical training, Gareth Morgan Thomas is now dedicated to sharing his deep understanding of science, technology, engineering, and mathematics through a series of specialized books aimed at both beginners and advanced learners.

Table of Contents

About the Author **vii**

1 Introduction to NVIDIA GPUs **1**
1.1 History of NVIDIA GPUs . . . 1
1.2 Applications of NVIDIA GPUs . . . 2

2 Understanding GPU Architecture **5**
2.1 GPU vs. CPU: Architectural Comparison . . . 5
2.2 Basics of Instruction Set Architecture (ISA) . . . 6

3 Key NVIDIA GPU Architectures **9**
3.1 Overview of Major NVIDIA Architectures . . . 9
3.2 Evolution of Design Goals . . . 12

4 Deep Dive into NVIDIA Microarchitectures **15**
4.1 Streaming Multiprocessors (SMs) . . . 15
4.2 Memory Hierarchy . . . 16
4.3 Threading and Warp Scheduling . . . 17

5 CUDA and Its Role in NVIDIA GPUs **19**
5.1 Introduction to CUDA . . . 19
5.2 How CUDA Integrates with Hardware . . . 20
5.3 Advantages and Limitations of CUDA . . . 21

6 Performance Optimization in NVIDIA GPUs **23**
6.1 Profiling and Debugging Tools . . . 23
6.2 Common Bottlenecks and Solutions . . . 24
6.3 Writing Efficient GPU Code . . . 25

7 Future Trends in NVIDIA GPUs **27**
7.1 AI and Deep Learning Integration . . . 27
7.2 New Architectural Directions . . . 27

8 Introduction to AMD GPUs **29**
8.1 History of AMD in Graphics Computing . . . 29
8.2 Applications of AMD GPUs . . . 30

9 Understanding AMD GPU Architecture **33**
9.1 GPU vs. CPU: Architectural Comparison . . . 33
9.2 Basics of AMD's ISA . . . 33

10 Key AMD GPU Architectures **35**
10.1 Overview of Major AMD Architectures . . . 35
10.2 Evolution of AMD Design Philosophy . . . 37

11 Deep Dive into AMD Microarchitectures **39**
11.1 Compute Units (CUs) and Shaders . . . 39
11.2 Memory Architecture . . . 40
11.3 Command Processors and Pipelines . . . 41

12 Programming for AMD GPUs **43**
12.1 ROCm (Radeon Open Compute) Ecosystem . . . 43
12.2 AMD GPUs with OpenCL . . . 44

13 Performance Optimization in AMD GPUs **45**
13.1 Profiling and Debugging Tools . . . 45
13.2 Identifying Bottlenecks . . . 46
13.3 Writing High-Performance GPU Code . . . 47

14 Future Trends in AMD GPUs **49**
14.1 RDNA 4 and Beyond . . . 49
14.2 AMD in Machine Learning and AI . . . 49

15 Comparatison of AMD and NVIDIA Architectures **51**
15.1 Introduction . . . 51
15.2 Architectural Fundamentals . . . 53
15.3 Memory System Comparisons . . . 57
15.4 Threading and Execution Models . . . 60
15.5 Performance Optimization Analogies . . . 63
15.6 Development Ecosystems . . . 67
15.7 Cross-Vendor Programming and Trends . . . 69
15.8 Conclusion . . . 71

16 GPU Assembly Fundamentals **73**
16.1 GPU ISA Architecture Deep Dive . . . 73
16.2 Memory System Architecture . . . 76
16.3 Execution Model Implementation . . . 78

17 Assembly Language Specifics **83**
17.1 Instruction Set Deep Dive . . . 83
17.2 Register Architecture . . . 85
17.3 Memory Access Patterns . . . 89

18 AMD GPU Assembly Architecture **93**
18.1 GCN/RDNA ISA Technical Details . . . 93
18.1.1 Item 4: Wave32/Wave64 execution models . . . 94
18.2 AMD Memory System . . . 96
18.3 AMD Performance Optimization . . . 99
18.4 NVIDIA Memory Architecture . . . 105
18.5 NVIDIA Performance Engineering . . . 108

19 Cross-Vendor Techniques **113**
19.1 Comparative Analysis . . . 113
19.2 Portable Assembly Code . . . 114
19.3 Cross-Vendor Debugging and Profiling . . . 116

20 Low-Level Optimization Strategies **119**
20.1 Memory System Optimization . . . 119
20.2 Instruction Scheduling . . . 122
20.3 Register Optimization . . . 125

21 Practical Applications **129**
21.1 Scientific Computing 129
21.2 Real-Time Graphics 131
21.3 Machine Learning 133

22 Performance Analysis Techniques **135**
22.1 Performance Counters 135
22.2 Optimization Methodology 138

23 Emerging Trends in GPU Assembly **143**
23.1 Next-Generation Architectures 143
23.2 Future of Low-Level Programming 144

24 Advanced Development Tools **147**
24.1 Assembly Development Tools 147
24.2 Profiling Implementation 150

Chapter 1

Introduction to NVIDIA GPUs

1.1 History of NVIDIA GPUs

Matrix Multiplication Exercise: Implement matrix multiplication for two given matrices `A` and `B` in Python. Assume both matrices are represented as nested lists.

1. Verify the dimensions of `A` and `B` are compatible for multiplication.
2. Compute the product `C = A * B` using nested loops.
3. Print the resulting matrix `C` with proper formatting.

Linear Regression Exercise: Fit a linear regression model to the following dataset using least squares:

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 5, 4, 5]
```

1. Compute the slope and intercept of the best-fit line.
2. Plot the data points and the regression line using `matplotlib`.
3. Calculate the coefficient of determination (`R^2`).

Sorting Algorithm Exercise: Implement the bubble sort algorithm to sort a list of integers in ascending order.

1. Write a function `bubble_sort(arr)` that modifies the input list in-place.
2. Add a flag to optimize the algorithm by detecting early completion.
3. Test your implementation on `[5, 3, 8, 4, 2]` and print the sorted result.

GPU Architecture Comparison Exercise:

1. Compare the memory hierarchy of NVIDIA's Fermi (2010) and Ampere (2020) architectures.
2. Calculate the theoretical peak FLOPS for an NVIDIA A100 GPU with 6912 CUDA cores running at 1.41 GHz.
3. Explain how AMD's Infinity Cache in RDNA 2 GPUs reduces memory bandwidth requirements.

CUDA Programming Exercise:

1. Write a CUDA kernel to perform element-wise addition of two arrays `A` and `B`.
2. Modify the kernel to include a check for array index bounds using `threadIdx.x` and `blockDim.x`.
3. Measure the execution time of your kernel using `cudaEventRecord` for arrays of size 1 million elements.

GPU Performance Analysis Exercise:

1. Profile a matrix multiplication kernel using NVIDIA Nsight Compute and identify the bottleneck.
2. Calculate the achieved memory bandwidth percentage compared to the theoretical maximum for your GPU.
3. Suggest three optimizations based on the occupancy calculator results for a kernel with 64 registers per thread.

1.2 Applications of NVIDIA GPUs

[Game Physics] Exercise: Implement a simple 2D physics engine for a platformer game. Your task:

1. Write a function `update_velocity(velocity, acceleration, dt)` that updates velocity using Euler integration.
2. Implement a collision detection function `check_collision(aabb1, aabb2)` for axis-aligned bounding boxes.
3. Create a response system that prevents objects from overlapping by adjusting their positions.
4. Test your system with a falling box that lands on a static platform.

[AI Behavior] Exercise: Create a finite state machine for an NPC enemy in a first-person shooter:

1. Define states `IDLE`, `CHASE`, and `ATTACK` with transitions between them.
2. Implement a distance check to trigger the `CHASE` state when the player is within 10 units.
3. Add a cooldown timer for the `ATTACK` state that lasts 2 seconds.
4. Visualize the state transitions using `print()` statements.

[Rendering] Exercise: Optimize a sprite rendering system for a 2D game:

1. Create a sprite batch class that stores vertex data in a single `std::vector`.
2. Implement texture atlas support using `glTexCoord2f` for UV coordinates.
3. Add a sorting mechanism to draw sprites from back to front based on their `z_index`.
4. Benchmark your system by rendering 1000 sprites and comparing frame times.

Linear Regression Implementation Exercise: Implement a simple linear regression model from scratch using Python.

1. Generate a synthetic dataset with 100 samples using `np.random.randn()` for features `X` and a linear relationship `y = 2*X + 1 + noise`.
2. Write a function to compute the mean squared error (MSE) between predictions and ground truth.
3. Implement gradient descent to optimize the parameters `w` (weight) and `b` (bias).
4. Plot the loss curve over iterations and the final fitted line against the data.

Decision Tree Classification Exercise: Train a decision tree classifier on the Iris dataset and evaluate its performance.

1. Load the Iris dataset using `sklearn.datasets.load_iris()` and split it into 70% training and 30% test sets.
2. Train a `DecisionTreeClassifier` with `max_depth=3` and `random_state=42`.
3. Compute and print the accuracy, precision, and recall scores on the test set.
4. Visualize the decision tree using `sklearn.tree.plot_tree()`.

Neural Network Forward Pass Exercise: Implement the forward pass of a 2-layer neural network with ReLU activation.

1. Define a network with input size 4, hidden layer size 5, and output size 3.
2. Initialize weights `W1` and `W2` using He initialization and biases as zeros.
3. Write the forward pass logic using ReLU for the hidden layer and softmax for the output.
4. Test the implementation on a random input tensor of shape `(10, 4)`.

Numerical Integration with Simpson's Rule Exercise: Implement Simpson's rule to approximate the integral of a function `f(x)` over the interval `[a, b]`.

1. Define the function `f(x)= exp(-x^2)` in Python using `numpy`.
2. Write a function `simpson(f, a, b, n)` that computes the integral using `n` subintervals.
3. Test your implementation by approximating $\int_0^1 f(x)\,dx$ with `n = 10` and compare the result to `scipy.integrate.quad`.

Matrix Condition Number Analysis Exercise: Investigate the condition number of a Hilbert matrix using `numpy.linalg`.

1. Generate a Hilbert matrix `H` of size `5x5` using `scipy.linalg.hilbert`.
2. Compute its condition number using `numpy.linalg.cond` with the 2-norm.
3. Solve the linear system `Hx = b` where `b` is a vector of ones, and compute the residual `norm(Hx - b)`.

ODE Solver Implementation Exercise: Implement the Euler method to solve the ODE `dy/dt = -2y` with initial condition `y(0)= 1`.

1. Write a function `euler(f, y0, t_span, h)` that applies the Euler method with step size `h`.
2. Use it to approximate `y(t)` over `t_span = [0, 2]` with `h = 0.1`.
3. Plot the solution alongside the exact solution `y(t)= exp(-2t)` using `matplotlib`.

Chapter 2

Understanding GPU Architecture

2.1 GPU vs. CPU: Architectural Comparison

Memory Coalescing Exercise:

1. Write a CUDA kernel that performs vector addition `C[i] = A[i] + B[i]` where each thread handles one element.
2. Modify the kernel to use 128-byte memory transactions by ensuring `A`, `B`, and `C` are aligned and accessed contiguously.
3. Measure the runtime difference between the original and optimized kernels using `cudaEvent_t` for a vector size of 1 million elements.

Warp Divergence Exercise:

1. Implement a CUDA kernel where threads in odd-numbered positions compute `x[i] * y[i]` and even-numbered threads compute `x[i] + y[i]`.
2. Rewrite the kernel to eliminate warp divergence using `threadIdx.x % 2` and measure the performance improvement.
3. Explain why the rewritten version avoids branch divergence within a warp.

Shared Memory Bank Conflict Exercise:

1. Write a CUDA kernel that computes a per-block sum of an array using shared memory with 32 banks.
2. Identify and fix bank conflicts in your implementation by adjusting the shared memory access stride.
3. Compare the execution time of the original and conflict-free versions using `nvprof`.

[Thermal efficiency comparison] Exercise:

1. Calculate the thermal efficiency of a Carnot engine operating between 500°C and 50°C.
2. A real heat engine operates between the same temperatures but achieves only 65% of the Carnot efficiency. Compute its actual efficiency.
3. If the real engine rejects 1200 kJ/min of waste heat, determine its power output in kW.

[Algorithm runtime analysis] Exercise:

1. Implement a Python function `linear_search(arr, x)` that returns the index of element `x` in list `arr`.
2. The same list is searched using binary search with `O(log n)` complexity instead of `O(n)`. Write the modified function `binary_search(arr, x)`.
3. Given a sorted list of 1,000,000 elements, calculate the maximum number of comparisons required by each algorithm when searching for a non-existent element.

[LED vs incandescent power consumption] Exercise:

1. A 60W incandescent bulb produces 800 lumens. Calculate its efficacy in lumens per watt.
2. An equivalent LED bulb consumes 9W to produce the same luminous flux. Compute its efficacy.
3. Determine the annual energy savings (in kWh) from using the LED bulb instead of the incandescent for 5 hours daily, assuming electricity costs $0.12/kWh.

2.2 Basics of Instruction Set Architecture (ISA)

Matrix Multiplication Exercise:

1. Given matrices `A = [[1, 2], [3, 4]]` and `B = [[5, 6], [7, 8]]`, compute the product `C = A * B` manually.
2. Implement a Python function `matrix_multiply(A, B)` that returns the product of two 2x2 matrices.
3. Verify your function by comparing its output with your manual calculation from step 1.

SQL Query Exercise:

1. Given a table `employees` with columns `id`, `name`, `salary`, and `department_id`, write a SQL query to find the average salary per department.
2. Modify the query to include only departments with an average salary greater than 50000.
3. Add a column showing the difference between each employee's salary and their department's average salary.

Ohm's Law Exercise:

1. Calculate the current through a resistor with `R = 220` ohms and `V = 5` volts using Ohm's Law.
2. Design a Python function `calculate_current(V, R)` that returns the current in amperes.
3. Test your function with the values from step 1 and a second case where `V = 12` and `R = 470`.

[Warp Divergence Analysis] Exercise:

1. Given the following CUDA kernel code, identify all potential warp divergence points:

```
__global__ void divergence_kernel(int* data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        if (data[idx] > 0) {
            data[idx] *= 2;
        } else {
            data[idx] -= 1;
        }
    }
}
```

2. For each divergence point, calculate the maximum possible threads affected when `blockDim.x` = 32
3. Rewrite the kernel to minimize warp divergence while maintaining identical functionality

[Shared Memory Bank Conflict] Exercise:

1. Analyze the following shared memory access pattern for bank conflicts:

```
__shared__ float smem[128];
float val = smem[threadIdx.x * 4];
```

2. Calculate the number of bank conflicts when `threadIdx.x` ranges from 0 to 31

3. Modify the access pattern to eliminate all bank conflicts while keeping the same logical data layout
4. Specify the required `__shared__` memory declaration for your solution

[PTX Register Usage] Exercise:

1. Compile the following CUDA kernel to PTX using `nvcc --ptx` and locate the register declarations

```
__global__ void reg_test(float* out, float a, float b) {
    float t1 = a * b;
    float t2 = a / b;
    float t3 = t1 + t2;
    out[threadIdx.x] = t3;
}
```

2. Count the total number of 32-bit registers used per thread
3. Modify the kernel to reduce register pressure by reusing variables, then verify the reduction in PTX
4. Determine the maximum possible block size (in threads) for your optimized kernel on a device with 64K registers per SM

Chapter 3

Key NVIDIA GPU Architectures

3.1 Overview of Major NVIDIA Architectures

[Fermi Energy Calculation] Exercise:

1. Calculate the Fermi energy E_F for copper at $T = 0$ K, given its electron density $n = 8.5 \times 10^{28}$ electrons/m^3. Use the formula $E_F = \frac{\hbar^2}{2m}(3\pi^2 n)^{2/3}$, where $\hbar = 1.05 \times 10^{-34}$ J·s and $m = 9.11 \times 10^{-31}$ kg.
2. Express the result in electron volts (eV) using 1 eV $= 1.6 \times 10^{-19}$ J.
3. Compare your result with the experimental value of 7.0 eV for copper and discuss possible reasons for discrepancies.

[Fermi-Dirac Distribution] Exercise:

1. Plot the Fermi-Dirac distribution $f(E) = \frac{1}{e^{(E-E_F)/kT}+1}$ for $E_F = 5$ eV at temperatures $T = 0$ K, 300 K, and 1000 K. Use `matplotlib` in Python.
2. Label the axes and include a legend. Use `numpy.linspace` for energy values from 0 to 10 eV.
3. Explain the physical significance of the broadening of the distribution at higher temperatures.

[Fermi Velocity Estimation] Exercise:

1. Estimate the Fermi velocity v_F for aluminum, given its Fermi energy $E_F = 11.7$ eV. Use the relation $E_F = \frac{1}{2}mv_F^2$.
2. Convert the result to m/s and compare it with the speed of light c. What fraction of c is v_F?
3. Write a Python function `fermi_velocity(ef)` that takes E_F in eV and returns v_F in m/s. Test it with the given value.

Orbital Period Calculation Exercise: Given a satellite in a circular orbit around Earth with an altitude of 800 km, calculate its orbital period using Kepler's third law. Assume Earth's mass is 5.972×10^{24} kg and the gravitational constant $G = 6.67430 \times 10^{-11}$ m^{3}kg^{-1}s^{-2}.

1. Compute the orbital radius by adding Earth's radius (6,371 km) to the altitude.
2. Apply Kepler's third law: $T^2 = \frac{4\pi^2 r^3}{GM}$.
3. Solve for T and express the result in minutes.

Elliptical Orbit Eccentricity Exercise: A spacecraft has an elliptical orbit with a perigee of 7,000 km and an apogee of 42,000 km from Earth's center.

1. Calculate the semi-major axis a of the orbit.
2. Determine the eccentricity e using the formula $e = \frac{r_a - r_p}{r_a + r_p}$, where r_a is apogee and r_p is perigee.

3. Verify that the perigee and apogee distances satisfy $r_p = a(1-e)$ and $r_a = a(1+e)$.

Orbital Velocity Simulation Exercise: Write a Python function to compute the orbital velocity at any point in an elliptical orbit using the vis-viva equation:

```
def orbital_velocity(r, a, mu):
    # r: current distance from central body
    # a: semi-major axis
    # mu: standard gravitational parameter (G*M)
    return (mu * (2/r - 1/a))**0.5
```

1. Test the function for a satellite with $a = 26,571$ km and $\mu = 3.986 \times 10^{14}$ m^{3}s^{-2} at perigee ($r = 7,000$ km) and apogee ($r = 42,000$ km).
2. Compare the results with the circular orbit velocity formula $v = \sqrt{\mu/r}$ at $r = a$.

[Faraday's Law Application] Exercise: A circular loop of radius `r = 0.1 m` lies in the xy-plane. A time-varying magnetic field is given by `B(t)= 0.5 * sin(2 * pi * 50 * t)` Tesla.

1. Calculate the induced electromotive force (EMF) in the loop.
2. Determine the direction of the induced current at `t = 0.005 s`.
3. Plot the EMF as a function of time for one full period.

[Wave Equation Derivation] Exercise: Starting from Maxwell's equations in free space (no charges or currents):

1. Derive the wave equation for the electric field `E`.
2. Show that the speed of the wave is `c = 1 / sqrt(mu_0 * epsilon_0)`.
3. Verify the units of `mu_0` and `epsilon_0` to confirm dimensional consistency.

[Poynting Vector Calculation] Exercise: An electromagnetic wave in vacuum has an electric field `E = E_0 * cos(kz - omega t) i_hat`.

1. Find the corresponding magnetic field `B`.
2. Compute the Poynting vector `S`.
3. Calculate the time-averaged power per unit area carried by the wave.

[Array Manipulation] Exercise: Write a Pascal program that declares an array of 10 integers and initializes it with the first 10 even numbers. Then, perform the following tasks:

- Print the original array elements in order.
- Calculate and print the sum of all elements.
- Replace every element at an odd index (1-based) with its square.
- Print the modified array elements in reverse order.

[File Handling] Exercise: Create a Pascal program that reads a text file named `input.txt` and performs the following operations:

- Count and display the total number of lines in the file.
- Write all lines containing the word "error" (case-insensitive) to a new file `errors.txt`.
- Append a summary line to `errors.txt` with the total error count.
- Handle file opening errors by displaying "File not found" if `input.txt` doesn't exist.

[Record Sorting] Exercise: Implement a Pascal program that manages student records with fields: `student_id` (integer), `name` (string), and `gpa` (real). Complete these tasks:

- Declare an array of 5 student records and populate them with user input.
- Sort the records in descending order of `gpa` using bubble sort.
- Display all records in tabular format after sorting.
- Calculate and print the average GPA of all students.

State Transition Diagram Exercise: Exercise: Design a Turing machine that recognizes the language $L = \{a^n b^n \mid n \geq 1\}$.

- Draw the state transition diagram with labeled transitions.
- Specify the tape alphabet $\Gamma = \{a, b, \sqcup\}$, where `\sqcup` is the blank symbol.
- Ensure the machine halts in an accepting state if the input string is in L.
- Include a rejection path for strings not in L.

Turing Machine Simulation Exercise: Exercise: Simulate the execution of a Turing machine for the input `aabb`.

- Use the machine from the previous exercise.
- Show the tape configuration and state after each step.
- Label the head position with an arrow (e.g., `a a b b`).
- Stop when the machine halts or after 10 steps, whichever comes first.

Programming a Universal Turing Machine Exercise: Exercise: Implement a universal Turing machine simulator in Python.

- Define a class `TuringMachine` with methods for step and run.
- Use a dictionary to encode the transition function.
- Represent the tape as a list with dynamic expansion.
- Test your simulator with the machine from the first exercise and input `aaabbb`.

Magnetic Field Around a Wire Exercise: A long straight wire carries a current of `I = 5 A`. Calculate the magnetic field magnitude at a perpendicular distance of `r = 0.1 m` from the wire. Use Ampere's Law and provide the steps:

1. Write the integral form of Ampere's Law for this scenario.
2. Simplify the equation for a long straight wire.
3. Substitute the given values and solve for the magnetic field `B`.

Current Loop Force Exercise: A square loop of side length `L = 0.2 m` carries a current `I = 3 A` and is placed in a uniform magnetic field `B = 0.5 T` directed perpendicular to the loop's plane. Determine the force on each side of the loop:

1. Sketch the loop and label the current direction and magnetic field.
2. Use the Lorentz force law to calculate the force on one side.
3. Repeat for all four sides and state the net force on the loop.

Solenoid Inductance Exercise: A solenoid has `N = 500` turns, length `l = 0.3 m`, and cross-sectional area `A = 0.01 m^2`. Calculate its self-inductance `L`:

1. Derive the formula for the magnetic field inside the solenoid.
2. Write the expression for the magnetic flux through one turn.

3. Compute the total flux linkage and solve for `L`.

Implementing the Analytical Engine's Division Algorithm Exercise:

1. Write a Python function `analytical_divide(dividend, divisor)` that implements Lovelace's division algorithm for the Analytical Engine.
2. The function must return a tuple `(quotient, remainder)` using only subtraction and comparison operations.
3. Include error handling for division by zero using `try-except` blocks.
4. Test your function with `dividend = 17` and `divisor = 3`, then verify against Python's built-in division.

Optimizing Bernoulli Number Calculation Exercise:

1. Implement Lovelace's Bernoulli number algorithm as a recursive function `bernoulli(n)` in Python.
2. The function should cache intermediate results using the `functools.lru_cache` decorator.
3. Calculate `bernoulli(4)` and compare the result with the known value `-1/30`.
4. Measure execution time for `n=8` with and without caching using `timeit`.

Simulating Punched Card Operations Exercise:

1. Create a class `PunchedCard` with methods `read_hole(position)` and `punch_hole(position)`.
2. Store card data as a bit array using `bytearray` where each bit represents a hole position.
3. Implement a method `to_binary_string()` that returns a 24-character string of 0s and 1s.
4. Simulate reading a card with holes at positions `[3, 7, 12]` and verify the binary output.

3.2 Evolution of Design Goals

Power dissipation in CMOS circuits Exercise

- Calculate the dynamic power consumption of a CMOS inverter with `V_DD` = 1.8V, switching frequency `f_sw` = 500MHz, load capacitance `C_L` = 15fF, and activity factor $\alpha = 0.3$.
- Derive the leakage current equation for a MOSFET in subthreshold region using the parameters `I_0`, `V_GS`, `V_TH`, and `n`.
- Implement a Verilog model of a clock gating cell that reduces dynamic power in a 4-bit counter when the enable signal `EN` is low:

```
module clock_gating (
    input clk,
    input EN,
    output reg gated_clk
);
    // Your code here
endmodule
```

Battery capacity calculations Exercise:

1. A IoT device draws 15mA continuously with 3.3V operation. Calculate the battery life in hours using a 1200mAh Li-ion battery.
2. Derive the Peukert's equation for battery capacity considering discharge rate `I` and Peukert's constant `n`.
3. Design a Python function that estimates remaining battery capacity given initial capacity `C_init`, current draw `I`, and elapsed time `t`:

```
def battery_capacity(C_init, I, t, n=1.05):
    # Your code here
```

Low-power digital design Exercise:

1. Compare the power consumption of a circuit using DVFS (Dynamic Voltage Frequency Scaling) between 1.2V/800MHz and 0.9V/500MHz operation with `C_eff` = 20nF.
2. List three techniques for reducing static power in nanometer CMOS designs.
3. Write a SystemVerilog assertion that checks if power-down mode is properly entered when `PWR_DOWN` is high for 5 cycles:

```
property check_power_down;
    @(posedge clk) // Your assertion here
endproperty
```

Memory Access Optimization Exercise: Exercise: Given the following C code snippet:

```
float sum_array(float* array, int size) {
    float sum = 0.0f;
    for (int i = 0; i < size; ++i) {
        sum += array[i];
    }
    return sum;
}
```

1. Identify two potential performance bottlenecks related to memory access patterns
2. Rewrite the function using loop unrolling with a factor of 4
3. Explain how your modification improves cache utilization
4. Measure the execution time before and after optimization using `clock_gettime()`

Parallel Processing Exercise: Exercise: A matrix multiplication algorithm is implemented as:

```
void matmul(double** A, double** B, double** C, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            C[i][j] = 0;
            for (int k = 0; k < n; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

1. Parallelize the outermost loop using OpenMP pragmas
2. Add appropriate reduction clauses for thread safety
3. Compare the speedup achieved with 2, 4, and 8 threads on a 1024x1024 matrix
4. Explain why the middle loop is not suitable for parallelization in this case

Algorithmic Complexity Exercise: Exercise: Consider a function that checks for duplicate elements:

```
bool has_duplicates(int* arr, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (arr[i] == arr[j]) return true;
        }
    }
    return false;
}
```

1. Calculate the time complexity of the current implementation
2. Propose an optimized version with O(n log n) complexity
3. Implement your solution using `qsort` from the C standard library
4. Verify the correctness of both implementations with test cases

Pipelined Processor Analysis Exercise:

1. Draw a 5-stage RISC pipeline diagram for the following MIPS instructions, showing stalls and forwarding paths:

```
lw $t0, 0($s1)
add $t1, $t0, $s2
sw $t1, 4($s1)
```

2. Calculate the total cycles required with and without forwarding.
3. Modify the code to eliminate stalls while maintaining the same functionality.

Cache Optimization Exercise:

1. Given a 4-way set-associative cache with 64-byte blocks and 16KB total size, compute the number of index bits and tag bits for a 32-bit address.
2. Analyze the cache behavior for this matrix multiplication kernel:

```
for (i=0; i<1024; i++)
    for (j=0; j<1024; j++)
        for (k=0; k<1024; k++)
            C[i][j] += A[i][k] * B[k][j];
```

3. Rewrite the kernel to improve spatial locality and reduce cache misses.

Branch Prediction Exercise:

1. Design a 2-bit saturating counter predictor for a branch at `0x4000a4` with this history: NT, T, NT, T, T (NT=Not Taken, T=Taken).
2. Calculate the prediction accuracy for both always-taken and 2-bit predictor strategies.
3. Implement a simple correlating predictor using Python that tracks the last two branch outcomes.

Chapter 4

Deep Dive into NVIDIA Microarchitectures

4.1 Streaming Multiprocessors (SMs)

[Matrix-Vector Multiplication] Exercise: Implement a parallel matrix-vector multiplication algorithm using OpenMP. The matrix `A` (size `n x n`) and vector `x` (size `n`) should be initialized with random values. Follow these steps:

- Use `#pragma omp parallel for` to parallelize the outer loop.
- Ensure thread safety by declaring private variables where necessary.
- Print the resulting vector `y = A * x`.
- Measure and report the execution time for `n = 1000` with 1, 2, 4, and 8 threads.

[Parallel Reduction] Exercise: Write a parallel reduction algorithm to compute the sum of an array `arr` of size `N = 1e6` using MPI. Follow these steps:

- Initialize `arr` with random values on the root process (rank 0).
- Scatter equal portions of `arr` to all processes.
- Compute local sums on each process.
- Use `MPI_Reduce` to aggregate the results to the root process.
- Print the final sum and verify correctness against a sequential sum.

[Load Balancing] Exercise: Simulate dynamic load balancing for a task queue using POSIX threads. Follow these steps:

- Create a shared task queue of 1000 integers (tasks).
- Spawn 4 worker threads to process tasks (e.g., compute `task_i * task_i`).
- Protect the queue with a mutex and use a condition variable to signal task availability.
- Ensure threads terminate when all tasks are completed.
- Report the total processing time and per-thread workload distribution.

[Thermal analysis of heat sinks] Exercise:

1. Calculate the thermal resistance `R_th` of an aluminum heat sink with base dimensions `100x100x10_mm` and 20 fins of height `50_mm`, thickness `2_mm`, and spacing `5_mm`. Use thermal conductivity `k=200_W/(m*K)`.
2. Modify the heat sink design to reduce `R_th` by 20% while keeping the base dimensions constant. Provide new fin dimensions.

3. Write a Python function using `scipy.optimize` to minimize `R_th` given constraints on total mass and base dimensions.

[PCB layout optimization] Exercise:

1. Design a 4-layer PCB stackup for a mixed-signal system with `3.3_V` digital and `5_V` analog sections. Specify layer purposes and materials.
2. Place components to minimize crosstalk between `I2C` and `ADC` traces. Provide a sketch using `tikz` coordinates.
3. Calculate the characteristic impedance of a `0.2_mm` wide microstrip on `1.6_mm` FR4 with `35_um` copper using the IPC-2141 formula.

[Embedded firmware debugging] Exercise:

1. Identify three bugs in the following STM32 HAL code snippet:

```
void UART_Transmit(uint8_t* data) {
  HAL_UART_Transmit(&huart2, data, strlen(data), 100);
  HAL_Delay(50);
  HAL_UART_Transmit(&huart2, "\r\n", 2, 100);
}
```

2. Rewrite the function to use DMA with circular buffer and implement error handling for `HAL_UART_ERROR_DMA`.
3. Add FreeRTOS task synchronization using `xQueueSend` to prevent UART overflows at `115200_bps`.

4.2 Memory Hierarchy

[Memory Allocation Analysis] Exercise: Consider the following CUDA kernel that uses global, shared, and local memory:

```
__global__ void compute(int* global_data, int N) {
    __shared__ int shared_data[128];
    int local_data = threadIdx.x;
    shared_data[threadIdx.x] = global_data[threadIdx.x % N] + local_data;
    __syncthreads();
    global_data[threadIdx.x] = shared_data[127 - threadIdx.x] * 2;
}
```

1. Identify all global, shared, and local memory variables in the kernel.
2. Explain the purpose of `__syncthreads()` in this context.
3. Calculate the maximum number of threads per block this kernel can support without modification.
4. Rewrite the kernel to avoid bank conflicts when accessing `shared_data`.

[Shared Memory Reduction] Exercise: Implement a CUDA kernel that performs a sum reduction using shared memory:

1. Allocate a shared memory array of 512 floats named `partial_sums`.
2. Each thread should load one element from global memory `input_data` to `partial_sums`.
3. Perform a tree-based reduction in shared memory to compute the sum of all elements.
4. Store the final result in `output_data[0]` in global memory.
5. Include necessary synchronization points and handle cases where the input size isn't a power of two.

[Memory Access Patterns] Exercise: Analyze the following memory access scenarios:

1. A warp reads 32 consecutive 4-byte words from global memory starting at address `0x1000`.
2. A warp reads 32 4-byte words spaced 128 bytes apart starting at `0x2000`.
3. A warp reads from `shared_data[threadIdx.x * 2]` where `shared_data` is 4-byte aligned.
4. For each case, determine whether the access is coalesced (global) or conflict-free (shared).
5. Calculate the total memory transactions required for each case on a GPU with 128-byte cache lines.

[Register Allocation Basics] Exercise:

1. Given the following C code snippet, identify all variables that must be stored in registers during execution:

```
int compute(int a, int b) {
    int c = a + b;
    int d = c * 2;
    return d;
}
```

2. For the function above, how many registers would be minimally required assuming a calling convention where arguments are passed in registers?
3. Rewrite the function in x86 assembly using at most 3 general-purpose registers.

[Register Spilling] Exercise:

1. Compile the following function with `-O0` and `-O2` optimization flags:

```
float dot_product(float* a, float* b, int n) {
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

2. Compare the generated assembly for both cases and identify where register spilling occurs in the `-O0` version.
3. Modify the C code to reduce register pressure and recompile to verify fewer spills.

[Calling Convention Practice] Exercise:

1. Write an x86-64 assembly function that follows the System V ABI to compute `(a * b)+ (c / d)` where all parameters are 64-bit integers.
2. Annotate your assembly to show which registers are used for parameter passing, computation, and return value.
3. Handle the case where `d = 0` by returning `0xDEADBEEF` without causing a division fault.

4.3 Threading and Warp Scheduling

[Warp Scheduling Policies] Exercise: Compare and contrast the following warp scheduling policies in GPUs:

- Describe how the `Greedy_Then_Oldest` policy prioritizes warps.
- Explain the trade-offs of the `Round_Robin` policy for latency-sensitive workloads.
- Write a pseudocode snippet using `if-else` logic to implement `Oldest_First` scheduling.

[Warp Divergence Analysis] Exercise: Analyze the following CUDA kernel for warp divergence:

```
__global__ void divergence_kernel(int* data, int threshold) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (data[idx] > threshold) {
        data[idx] *= 2;
    } else {
        data[idx] /= 2;
    }
}
```

- Calculate the maximum number of divergent warps for a block size of 128 threads.
- Modify the kernel to use `__shfl_sync` for reduced divergence.
- Estimate the performance penalty if 50% of warps diverge on an NVIDIA Volta GPU.

[Occupancy Calculation] Exercise: Given a GPU with the following specifications:

- Compute Capability 7.0 (Volta)
- 32 warps per SM maximum
- 64 registers per thread
- 96 KB shared memory per SM
- Calculate the register-limited occupancy for a kernel using 128 threads per block.
- Determine if shared memory will be a limiting factor when using 4 KB per block.
- Propose block dimensions to achieve at least 75% occupancy.

[Thread indexing] Exercise:

1. Write a CUDA kernel that computes the 1D global thread index using `blockIdx.x`, `blockDim.x`, and `threadIdx.x`.
2. Launch the kernel with 256 threads per block and 16 blocks.
3. Print the computed global index for each thread using `printf` inside the kernel.
4. Verify that the printed indices range from 0 to 4095 (inclusive).

[Matrix addressing] Exercise:

1. Implement a CUDA kernel that computes 2D matrix indices for each thread using `blockIdx.y`, `threadIdx.y`, `blockIdx.x`, and `threadIdx.x`.
2. Configure the kernel launch to process a 64×64 matrix using 16×16 thread blocks.
3. Each thread should compute its global row index as `blockIdx.y * blockDim.y + threadIdx.y` and column index similarly.
4. Store the computed row and column indices in a 2D output array in device memory.

[Block synchronization] Exercise:

1. Create a CUDA kernel with two separate computation phases in each thread.
2. Use `__syncthreads()` to synchronize all threads in a block after the first phase.
3. In the first phase, have each thread compute the square of its `threadIdx.x` value.
4. In the second phase, have each thread add the results from all threads in its block using shared memory.
5. Launch the kernel with 128 threads per block and 8 blocks.

Chapter 5

CUDA and Its Role in NVIDIA GPUs

5.1 Introduction to CUDA

[Speedup calculation] Exercise: A parallel program takes 8 seconds to execute on a single processor and 2 seconds on 4 processors.

1. Calculate the speedup achieved by using 4 processors.
2. Compute the parallel efficiency for this case.
3. If the program's serial fraction is 0.1, calculate the maximum possible speedup according to Amdahl's Law.

[MPI collective operations] Exercise: Consider the following MPI code snippet:

```
int rank, data[4];
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
for (int i=0; i<4; i++) data[i] = rank*10 + i;
```

1. Write the MPI call to gather all processes' `data` arrays to rank 0.
2. Modify the code to perform an all-to-all broadcast of `data` arrays.
3. Explain what happens if we replace `MPI_COMM_WORLD` with `MPI_COMM_SELF`.

[CUDA thread organization] Exercise: A CUDA kernel is launched with `<<<16, 32>>>` configuration:

1. Calculate the total number of threads in this grid.
2. Determine the `blockIdx.x` and `threadIdx.x` values for thread 127.
3. Write the formula to compute a unique global thread ID from `blockIdx` and `threadIdx`.

[CUDA Thread Indexing] Exercise: Write a CUDA kernel that initializes an array `output_array` of size `N` (where `N` is a multiple of 1024) such that each element at index `i` is set to the value of its thread ID. Use the following steps:

1. Declare the kernel with the signature `__global__ void init_array(int* output_array, int N)`.
2. Compute the global thread ID using `blockIdx.x * blockDim.x + threadIdx.x`.
3. Add a boundary check to ensure the thread ID does not exceed `N`.
4. Launch the kernel with 256 threads per block and enough blocks to cover `N`.

[GPU Matrix Transpose] Exercise: Implement a matrix transpose operation in CUDA for a square matrix of size `M x M`. Follow these steps:

1. Allocate device memory for input matrix `A` and output matrix `B`.
2. Write a kernel where each thread transposes one element from `A` to `B`, i.e., `B[j][i] = A[i][j]`.

3. Use shared memory to optimize memory access patterns.
4. Launch the kernel with a 2D grid and block configuration matching the matrix dimensions.
5. Verify correctness by copying `B` back to the host and comparing with a CPU-based transpose.

[Reduction on GPU] Exercise: Implement a parallel reduction kernel in CUDA to compute the sum of an array `data` of size `N` (power of 2). Follow these steps:

1. Allocate device memory for `data` and a temporary array `partial_sums`.
2. Write a kernel that performs reduction in shared memory, halving the active threads in each iteration.
3. Use `__syncthreads()` to synchronize threads within a block.
4. Launch multiple blocks and combine their partial sums in a final kernel call.
5. Compare the result with a sequential CPU reduction for validation.

5.2 How CUDA Integrates with Hardware

Memory access patterns Exercise:

1. Write a CUDA kernel that performs element-wise addition of two arrays `A` and `B`, storing the result in array `C`.
2. Modify the kernel to use shared memory for coalesced global memory access when the array size exceeds 1024 elements.
3. Add a check for thread index bounds to prevent out-of-range memory accesses.

Thread divergence Exercise:

1. Implement a kernel that processes an array of integers, where each thread checks if its element is even or odd using `if-else`.
2. Measure the performance impact of this divergence using CUDA events.
3. Rewrite the kernel using arithmetic operations instead of branching and compare the execution times.

Occupancy optimization Exercise:

1. Write a matrix multiplication kernel using `tile_width=16` without shared memory.
2. Profile the kernel to determine register usage and occupancy using NVIDIA's profiler.
3. Adjust the kernel to maximize occupancy by limiting register usage via `__launch_bounds__` or compiler flags.

[CUDA thread block mapping] Exercise:

1. Write a CUDA kernel that launches with `dim3 grid(4, 2, 1)` and `dim3 block(32, 8, 1)`.
2. Calculate the total number of threads per block and total blocks in the grid.
3. Print the `blockIdx.x`, `blockIdx.y`, `threadIdx.x`, and `threadIdx.y` for each thread.
4. Modify the kernel to map each thread to a unique global index using `blockDim` and `gridDim`.

[Memory coalescing optimization] Exercise:

1. Create a CUDA kernel that performs matrix transposition of a 1024x1024 float matrix.
2. Implement both a naive version and an optimized version using shared memory.
3. Measure the bandwidth of both versions using `cudaEventRecord`.

4. Explain how thread block mapping affects memory coalescing in both cases.

[Occupancy calculation] Exercise:

1. Given a GPU with 48KB shared memory per SM and 2048 threads per SM.
2. Calculate maximum occupancy for kernels using: (a) 128 threads/block and 16KB shared memory, (b) 256 threads/block and 32KB shared memory.
3. Determine which configuration achieves higher occupancy and why.
4. Verify your calculation using the CUDA Occupancy Calculator API.

5.3 Advantages and Limitations of CUDA

Chapter 6

Performance Optimization in NVIDIA GPUs

6.1 Profiling and Debugging Tools

Profiling CUDA Kernels Exercise: Analyze the performance of a matrix multiplication kernel using NVIDIA Nsight Compute.

1. Write a CUDA kernel for matrix multiplication using shared memory (tiling) and compile it with `--generate-line-info`.
2. Launch the kernel with `Nsight Compute` and capture the profile using `--launch-skip 2 --launch-count 5`.
3. Identify the bottleneck by examining the `GPU Speed Of Light` section in the report.
4. Modify the kernel to reduce shared memory bank conflicts and rerun the profiling.
5. Compare the achieved occupancy and execution time before and after optimization.

Debugging Warp Divergence Exercise: Detect and fix warp divergence in a parallel reduction kernel.

1. Implement a reduction kernel that sums an array of 1024 elements using `__shfl_down_sync`.
2. Run the kernel under `Nsight Compute` with the `--set full` flag.
3. Locate the `Warp State Statistics` metric to identify divergent warps.
4. Rewrite the reduction to use a divergence-free strategy (e.g., sequential addressing).
5. Verify the improvement by comparing the `Stall Reasons` section in both profiles.

Memory Access Pattern Analysis Exercise: Optimize global memory access in a stencil computation.

1. Write a 3D stencil kernel that accesses a 32x32x32 grid with `float` data.
2. Profile the kernel with `Nsight Compute` and note the `L1/TEX Cache Hit Rate`.
3. Use the `Memory Workload Analysis` section to identify non-coalesced accesses.
4. Restructure the kernel to use `__ldg` for read-only data and pad shared memory.
5. Re-profile and check the `DRAM Utilization` metric for improvement.

Profiling Matrix Multiplication Exercise: Given the following CUDA matrix multiplication kernel:

```
__global__ void matMulKernel(float* A, float* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; ++k) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

1. Compile the kernel with `nvcc -Xptxas -O3` and profile it using `nvprof` for `N=1024`.
2. Record the achieved occupancy percentage and global memory throughput.
3. Modify the kernel to use shared memory and profile again.
4. Compare the original and optimized kernel's `gld_transactions` metric.

Concurrent Kernel Analysis Exercise:

1. Write two simple CUDA kernels: one that writes to global memory and one that performs floating-point computations.
2. Use `nvprof --concurrent-kernels` to profile their simultaneous execution.
3. Record the timeline showing kernel overlap in `nsight-sys`.
4. Increase the grid size of both kernels and observe how it affects overlap.

Memory Transfer Optimization Exercise:

1. Create a CUDA program that copies `float` data between host and device using `cudaMemcpy`.
2. Profile with `nvprof --print-gpu-trace` to measure transfer times.
3. Replace with pinned host memory using `cudaMallocHost` and re-profile.
4. Calculate the bandwidth improvement for `1MB`, `10MB`, and `100MB` data sizes.

6.2 Common Bottlenecks and Solutions

Measuring Load-to-Use Delay Exercise:

1. Write a C function `measure_load_latency()` that allocates a 1MB buffer and measures the time difference between loading a value from memory and using it in a computation.
2. Use `rdtsc` for cycle-accurate timing and ensure the compiler doesn't optimize away your memory access by declaring pointers with `volatile`.
3. Plot the latency distribution over 10,000 trials using a histogram with 100ns bins.

Cache Line Alignment Exercise:

1. Create two C structures: one with `int` fields aligned to cache lines (`__attribute__((aligned(64)))`) and one without explicit alignment.
2. Measure the access time difference for reading 10,000 sequential elements from each structure.
3. Calculate the effective bandwidth for both cases assuming a 64-byte cache line and 2.4GHz clock speed.

Prefetching Strategy Exercise:

1. Implement a matrix multiplication kernel in C that uses `__builtin_prefetch` to fetch the next cache line while computing the current one.
2. Compare the performance against a naive implementation using 1024x1024 float matrices.
3. Determine the optimal prefetch distance by testing offsets from 1 to 10 cache lines ahead.

[Pipeline Throughput Analysis] Exercise:

1. Calculate the throughput (instructions per cycle) of a 5-stage RISC pipeline when executing a loop with no data hazards.
2. Modify your calculation for a scenario where 30% of instructions are `LOAD`s with 1-cycle latency.
3. Determine the throughput impact if branch mispredictions occur every 8 instructions with a 3-cycle penalty.

[SIMD Optimization] Exercise:

1. Write an AVX2 intrinsic function to multiply four 64-bit floating-point numbers in `__m256d` registers.
2. Measure the throughput difference between your AVX2 implementation and a scalar version using `rdtsc`.
3. Identify three common bottlenecks in SIMD instruction throughput on x86 processors.

[Branch Prediction] Exercise:

1. Implement a branchless version of this code snippet:

```
if (x > threshold) {
    result += x * factor;
}
```

2. Calculate the theoretical throughput improvement assuming:
   - 90% prediction accuracy
   - 5-cycle branch misprediction penalty
   - 1-cycle branchless execution
3. Validate your calculation with performance counters using `perf stat`.

6.3 Writing Efficient GPU Code

[Memory Coalescing] Exercise:

1. Write a CUDA kernel that performs element-wise addition of two arrays `A` and `B`, storing the result in array `C`.
2. Ensure memory accesses are coalesced by having each thread read and write contiguous elements.
3. Launch the kernel with 256 threads per block and grid dimensions to cover the entire array size `N`.
4. Use `cudaMallocManaged` for unified memory allocation.
5. Verify correctness by comparing with a CPU implementation.

[Occupancy Optimization] Exercise:

1. Profile a given CUDA kernel using `nvprof` and note the achieved occupancy.
2. Modify the kernel to increase occupancy by adjusting the thread block size.
3. Experiment with block sizes of 128, 256, and 512 threads.
4. Measure the runtime and occupancy for each configuration using `nvprof`.

5. Explain which block size yields the best performance and why.

[Shared Memory Reduction] Exercise:

1. Implement a parallel reduction kernel to compute the sum of an array using shared memory.
2. Use the `__syncthreads()` barrier to synchronize threads within a block.
3. Avoid bank conflicts by ensuring adjacent threads access non-conflicting shared memory addresses.
4. Compare the performance of your kernel with a naive reduction without shared memory.
5. Measure the speedup using `nvprof`.

Memory Coalescing Optimization Exercise:

1. Write a CUDA kernel that performs element-wise addition of two arrays `A` and `B`, storing the result in array `C`.
2. Modify the kernel to ensure memory coalescing by reordering the thread-to-data mapping.
3. Measure the execution time before and after optimization using `cudaEvent_t` and report the speedup.
4. Explain how your changes improved memory access patterns.

Shared Memory Reduction Exercise:

1. Implement a parallel reduction kernel in CUDA that sums all elements of an array using global memory only.
2. Rewrite the kernel to use shared memory for intermediate results.
3. Compare the performance of both versions for array sizes of `1e6` and `1e7` elements.
4. Add a barrier synchronization (`__syncthreads()`) where needed and justify its placement.

Occupancy Analysis Exercise:

1. Write a matrix multiplication kernel using `tile_width=16` and `tile_height=16`.
2. Use the CUDA Occupancy Calculator to determine the theoretical occupancy for your kernel on an NVIDIA A100 GPU.
3. Adjust the thread block dimensions to maximize occupancy while maintaining the same tile size.
4. Validate the actual occupancy using `nvprof` or Nsight Compute.

Chapter 7

Future Trends in NVIDIA GPUs

7.1 AI and Deep Learning Integration

Matrix Multiplication Optimization Exercise:

1. Implement a naive matrix multiplication function in Python using nested loops for two 1024x1024 matrices.
2. Profile the execution time of your implementation using `time.time()`.
3. Modify the function to use NumPy's `np.dot()` and compare the speedup.
4. Calculate the theoretical FLOPs for both implementations assuming a 2.5 GHz CPU.

GPU Memory Hierarchy Exercise:

1. Write a CUDA kernel to perform vector addition using global memory only.
2. Modify the kernel to use shared memory for tiles of size `BLOCK_SIZE=32`.
3. Measure the bandwidth improvement using `cudaEventRecord()` timing.
4. Explain how the L2 cache affects performance in both cases.

Quantization Aware Training Exercise:

1. Create a simple CNN with 2 convolutional layers in PyTorch for MNIST classification.
2. Insert `torch.quantization.FakeQuantize` layers after each ReLU activation.
3. Train the model for 5 epochs with standard floating-point precision.
4. Convert the model to INT8 using `torch.quantization.convert` and measure accuracy drop.

7.2 New Architectural Directions

Memory Hierarchy Analysis Exercise: Exercise:

1. Calculate the effective access time for a Hopper architecture with the following parameters: L1 cache hit time = 1 cycle (95% hit rate), L2 cache hit time = 5 cycles (85% hit rate), main memory access time = 100 cycles.
2. Modify the calculation if Grace introduces a new L0 cache with 0.5 cycle access time and 98% hit rate.
3. Compare the results and explain which architecture performs better for this memory hierarchy configuration.

Instruction Pipeline Design Exercise: Exercise:

1. Design a 5-stage pipeline for a Hopper processor with stages: Fetch, Decode, Execute, Memory, Writeback.
2. Identify potential hazards when executing `ADD R1, R2, R3` followed by `SUB R4, R1, R5`.

3. Propose two solutions Grace could implement to mitigate these hazards without adding pipeline stages.

Vector Unit Implementation Exercise: Exercise:

1. Write a C code snippet using `#pragma omp simd` to vectorize a dot product calculation.
2. Convert this to Hopper assembly using `VLOAD`, `VMUL`, and `VREDUCE` instructions.
3. Calculate the theoretical speedup when Grace introduces 512-bit vectors compared to Hopper's 256-bit vectors for this operation.

Chapter 8

Introduction to AMD GPUs

8.1 History of AMD in Graphics Computing

GPU Architecture Comparison Exercise:

1. Research the key architectural differences between ATI's R600 (Radeon HD 2000 series) and AMD's GCN (Graphics Core Next) architectures.
2. Create a table comparing stream processors, memory bandwidth, and instruction set features between these architectures.
3. Explain how AMD integrated ATI's technologies into their own GPU roadmap after the 2006 acquisition.

Driver Compatibility Analysis Exercise:

1. Write a Python script using `subprocess` to test OpenGL version support on an AMD GPU using `glxinfo` (Linux) or `dxdiag` (Windows).
2. Compare the output with legacy ATI Catalyst driver documentation from 2004-2006.
3. Identify three API functions that were deprecated during the transition from ATI to AMD driver stacks.

Market Share Calculation Exercise:

1. Using the quarterly reports from AMD (2006-Q2 to 2007-Q4), calculate the percentage change in GPU market share post-acquisition.
2. Plot the data using

```
\documentclass{standalone}
\usepackage{pgfplots}
\begin{document}
\begin{tikzpicture}
% Plotting code here
\end{tikzpicture}
\end{document}
```

3. Correlate the market share changes with the release dates of Radeon HD 3000 series GPUs.

GPU architecture evolution Exercise:

1. Compare the execution model of a `SIMT` (Single Instruction Multiple Thread) architecture with traditional `SIMD`. List two advantages of `SIMT` for irregular workloads.
2. Calculate the theoretical peak FLOPS for an NVIDIA A100 GPU with 6912 CUDA cores running at 1.41 GHz, assuming each core can perform two floating-point operations per cycle.
3. Implement a CUDA kernel to perform element-wise addition of two arrays using `blockDim.x` and `threadIdx.x` for thread indexing.

Memory hierarchy optimization Exercise:

1. Explain how the L1 cache and shared memory interact in modern GPU architectures. What is the typical size range for each in current GPUs?
2. Write a CUDA kernel that uses shared memory to compute the sum of a 256-element array in a single thread block.
3. Analyze the memory access pattern of a stencil computation (3x3 neighborhood) and suggest optimal tile dimensions for shared memory utilization.

Parallel primitives implementation Exercise:

1. Implement a parallel reduction kernel in CUDA that finds the maximum value in an array using warp shuffle operations.
2. Design a GPU-optimized histogram algorithm that uses atomic operations on shared memory before global memory writes.
3. Compare the performance implications of using `atomicAdd` versus a reduction-based approach for computing dot products on GPUs.

8.2 Applications of AMD GPUs

[Collision Detection] Exercise: Implement a 2D axis-aligned bounding box (AABB) collision check between two rectangles. Given:

- Rectangle A with position `(x1, y1)`, width `w1`, and height `h1`
- Rectangle B with position `(x2, y2)`, width `w2`, and height `h2`

Write a function `bool check_collision(float x1, y1, w1, h1, x2, y2, w2, h2)` that returns `true` if the rectangles overlap. Use this formula:

```
return (x1 < x2 + w2) && (x1 + w1 > x2) &&
       (y1 < y2 + h2) && (y1 + h1 > y2);
```

[Pathfinding] Exercise: Implement Dijkstra's algorithm for a 5x5 grid with weighted nodes. Given:

- A 2D array `grid[5][5]` where each cell contains a movement cost (1-9)
- Start position `(0, 0)` and goal position `(4, 4)`
- Movement allowed in 4 directions (no diagonals)

Write pseudocode to:

```
1. Initialize distance matrix with infinity
2. Create a priority queue with start node (cost=0)
3. While queue not empty:
   a. Dequeue node with lowest cost
   b. If node is goal, return path
   c. For each neighbor:
      i. Calculate new_cost = current_cost + grid[neighbor]
      ii. If new_cost < stored_cost, update and enqueue
```

[Rendering Optimization] Exercise: Optimize a sprite batch renderer by implementing a texture atlas. Given:

- 100 sprites with individual 256x256 textures
- Current draw calls: 100 (one per sprite)
- GPU supports textures up to 2048x2048

Calculate:

```
1. Minimum atlas size to fit all sprites (2048x2048)
2. New UV coordinates for sprite at (row=2, col=3) in atlas
3. Vertex shader code to sample from atlas using:
   vec2 uv = (sprite_pos + sprite_size * frag_coord) / atlas_size;
```

Parallel Matrix Multiplication Exercise: Implement a parallel matrix multiplication algorithm using MPI. Assume square matrices of size `N x N`.

1. Write a function to initialize two matrices `A` and `B` with random values.
2. Distribute the rows of `A` across MPI processes using `MPI_Scatter`.
3. Broadcast matrix `B` to all processes using `MPI_Bcast`.
4. Compute the local product of the assigned rows of `A` with `B`.
5. Gather the results using `MPI_Gather` to assemble the final product matrix.
6. Validate correctness by comparing with a sequential implementation.

OpenMP Loop Optimization Exercise: Optimize a computationally intensive loop using OpenMP directives.

1. Write a loop to compute the sum `sum = sum + A[i] * B[i]` for arrays of size `N`.
2. Add OpenMP pragmas to parallelize the loop across threads.
3. Use `reduction` to handle the `sum` variable correctly.
4. Measure execution time with `omp_get_wtime()` for `N = 1e8`.
5. Compare speedup against the sequential version for 2, 4, and 8 threads.

CUDA Vector Addition Exercise: Implement a vector addition kernel in CUDA.

1. Allocate host arrays `A`, `B`, and `C` of size `N`.
2. Allocate device memory for these arrays using `cudaMalloc`.
3. Write a CUDA kernel to compute `C[i] = A[i] + B[i]`.
4. Launch the kernel with appropriate grid and block dimensions.
5. Copy the result back to the host and verify correctness.
6. Measure kernel execution time using `cudaEvent`.

Linear regression implementation Exercise:

1. Load the diabetes dataset from `sklearn.datasets`.
2. Split the data into training and test sets using `train_test_split` with 30% test size.
3. Implement linear regression using the normal equation: $\theta = (X^T X)^{-1} X^T y$.
4. Compare your results with `sklearn.linear_model.LinearRegression`.
5. Calculate and print the mean squared error for both models on the test set.

Decision tree hyperparameter tuning Exercise:

1. Load the breast cancer dataset from `sklearn.datasets`.
2. Create a decision tree classifier with default parameters.
3. Use grid search with 5-fold cross-validation to find optimal `max_depth` and `min_samples_split`.
4. Evaluate the best model on a held-out test set (20% of data).
5. Plot the decision tree using `sklearn.tree.plot_tree` with feature names.

Neural network for MNIST Exercise:

1. Load the MNIST dataset using `keras.datasets.mnist.load_data`.
2. Normalize pixel values to [0,1] and one-hot encode labels.
3. Build a sequential model with two dense layers (128 and 10 units) using ReLU and softmax activations.
4. Compile the model with Adam optimizer and categorical crossentropy loss.
5. Train for 10 epochs with batch size 32 and plot the training/validation accuracy.

Chapter 9

Understanding AMD GPU Architecture

9.1 GPU vs. CPU: Architectural Comparison

GPU Memory Bandwidth Calculation Exercise:

1. A GPU has a memory clock speed of 1.75 GHz and a 384-bit memory interface. Calculate the theoretical memory bandwidth in GB/s assuming GDDR5 memory (effective data rate = 4 transfers per clock cycle).
2. If this GPU achieves 85% of its theoretical bandwidth in practice, what would be the actual bandwidth available for a CUDA kernel?
3. Given a matrix multiplication kernel that processes 4096x4096 single-precision matrices, estimate the minimum time required just for memory transfers (assuming no computation overhead).

CUDA Thread Hierarchy Exercise:

1. Write a CUDA kernel that adds two vectors using `blockDim.x = 256` and `gridDim.x` calculated for vectors of length 1,048,576.
2. Modify the kernel to handle cases where the vector length isn't evenly divisible by `blockDim.x`.
3. Add a shared memory optimization to compute partial sums within each block before writing to global memory.

Heterogeneous Pipeline Analysis Exercise:

1. A video processing pipeline spends 15ms on CPU preprocessing, 8ms on PCIe transfer to GPU, and 5ms on GPU processing. Calculate the theoretical speedup if GPU processing time is reduced to 3ms.
2. Identify which component becomes the bottleneck if the PCIe transfer speed is doubled.
3. Propose a modification to overlap CPU-GPU transfers with computation to hide transfer latency.

9.2 Basics of AMD's ISA

[Compute Unit Structure] Exercise:

1. Identify the key components of a Compute Unit (CU) in GCN architecture and list their functions.
2. Compare the number of stream processors per CU in GCN (e.g., GCN 1.0) and RDNA (e.g., RDNA 2).
3. Write a simple OpenCL kernel using `__attribute__((reqd_work_group_size(X, Y, Z)))` to match a GCN CU's wavefront size.

[Memory Hierarchy] Exercise:

1. Calculate the effective bandwidth of GDDR6 memory in an RDNA 2 GPU with a 256-bit bus and 14 Gbps pin speed.

2. Explain the role of the L2 cache in RDNA architecture using a diagram (ASCII art in `verbatim` environment).
3. Write a ROCm HIP code snippet to prefetch data into `LDS` (Local Data Share) using `__builtin_amdgcn_ds_prefetch`.

[Instruction Set] Exercise:

1. Disassemble the following GCN ISA instruction: `v_mac_f32 v0, v1, v2` and explain its operation.
2. List three differences between scalar and vector ALU pipelines in RDNA.
3. Write an inline assembly block for RDNA to perform a packed FP16 multiply using `v_pk_mul_f16`.

Chapter 10

Key AMD GPU Architectures

10.1 Overview of Major AMD Architectures

[GCN Instruction Encoding] Exercise:

1. Write a GCN assembly instruction to perform a 32-bit floating-point multiply-add operation on registers `v0`, `v1`, and `v2`, storing the result in `v3`.
2. Encode the above instruction as a 32-bit hexadecimal value using the GCN instruction format.
3. Identify which bits in the encoded instruction correspond to the opcode field.

[Wavefront Scheduling] Exercise:

1. Calculate the maximum number of wavefronts that can concurrently execute on a GCN GPU with 40 compute units, given each compute unit can hold 10 wavefronts.
2. A kernel launches with 65,536 work-items divided into wavefronts of 64 work-items each. How many wavefronts does this kernel require?
3. Explain what happens when the number of wavefronts exceeds the GPU's capacity.

[Memory Coalescing] Exercise:

1. Given a GCN GPU with a 128-byte cache line size, determine whether the following access pattern is coalesced: 16 work-items reading consecutive 32-bit words starting at address `0x1000`.
2. Rewrite the following OpenCL kernel to improve memory coalescing:

```
__kernel void bad_coalesce(__global float* input, __global float* output) {
    int tid = get_global_id(0);
    output[tid * 4] = input[tid * 4];
}
```

3. Calculate the memory bandwidth wasted per cache line if only 32 bytes out of 128 are utilized.

[Black-Scholes Vega Calculation] Exercise: Compute the Vega of a European call option using the Black-Scholes model. Given:

- Stock price `S = 100`
- Strike price `K = 105`
- Risk-free rate `r = 0.03`
- Volatility `sigma = 0.25`
- Time to maturity `T = 1`

Use the formula:

```
Vega = S * sqrt(T) * N'(d1)
```

where `N'(x)` is the standard normal PDF and `d1` is the standard Black-Scholes parameter.

[Vega Hedging Implementation] Exercise: Write Python code to hedge Vega exposure for a portfolio containing:

- 100 call options with Vega `0.35` per option
- 50 put options with Vega `0.28` per option

Calculate how many units of a hedging instrument with Vega `0.42` are needed to make the portfolio Vega-neutral. Provide the complete calculation and code:

```
# Your Python implementation here
```

[Volatility Surface Vega Analysis] Exercise: Given the following volatility surface data for a stock:

- At `T=0.5`: Vega `0.18` for `K=90`, Vega `0.22` for `K=100`
- At `T=1.0`: Vega `0.25` for `K=90`, Vega `0.30` for `K=100`

Create a matrix showing the total Vega exposure for:

- A long position of 200 options at `T=0.5, K=100`
- A short position of 150 options at `T=1.0, K=90`

Present the results in a clear tabular format. **[Shader Core Utilization] Exercise:** Compute the theoretical maximum floating-point operations per second (FLOPS) for an RDNA 2 GPU with the following specifications:

- 40 compute units (CUs)
- 64 stream processors per CU
- 2 FMA (fused multiply-add) operations per cycle per stream processor
- GPU clock speed of 2.4 GHz

Show your calculations using the formula: `FLOPS = CUs × SPs_per_CU × Ops_per_cycle × Clock_speed`.

[Wavefront Scheduling] Exercise: Given an RDNA 3 GPU with the following workload:

- 10,240 threads divided into wavefronts of 32 threads each
- 48 compute units available
- Each CU can execute 4 wavefronts concurrently

Calculate:

- Total number of wavefronts
- Minimum number of cycles to execute all wavefronts (assuming no stalls)
- Occupancy percentage if only 32 CUs are active

[Memory Bandwidth Analysis] Exercise: An RDNA-based GPU has the following memory configuration:

- 256-bit GDDR6 memory interface
- Memory clock speed of 1.75 GHz (effective 14 Gbps due to QDR)
- 512 MB L2 cache with 2 TB/s bandwidth

Perform these tasks:

- Calculate peak memory bandwidth using `Bandwidth = Interface_width × Effective_speed / 8`.
- Compare this to the L2 cache bandwidth in percentage terms.

- Identify which component (memory or cache) becomes the bottleneck when 80% of memory accesses hit the L2 cache.

[Compute Unit Comparison] Exercise:

1. Compare the number of Compute Units (CUs) in AMD's RDNA 2 (e.g., `Navi 21`) and RDNA 3 (e.g., `Navi 31`) architectures.
2. Calculate the theoretical FP32 (single-precision) throughput for both architectures, assuming a base clock speed of `2.0 GHz`.
3. Explain how the dual-issue SIMD capability in RDNA 3 affects throughput compared to RDNA 2.

[Infinity Cache Analysis] Exercise:

1. Identify the Infinity Cache sizes for RDNA 2's `RX 6900 XT` and RDNA 3's `RX 7900 XT`.
2. Using the memory bandwidth and cache hit rate data, estimate the effective bandwidth improvement from RDNA 2 to RDNA 3.
3. Discuss how the partitioning of Infinity Cache in RDNA 3 reduces latency for high-resolution workloads.

[Ray Tracing Implementation] Exercise:

1. Write a simplified HLSL code snippet to demonstrate ray traversal using RDNA 2's `Ray Accelerator` units.
2. Modify the code to leverage RDNA 3's improved `Ray Accelerator` with dual issue capability.
3. Compare the expected ray intersection throughput for both architectures, assuming identical scene complexity.

10.2 Evolution of AMD Design Philosophy

[Frame Time Analysis] Exercise: A game's frame times (in milliseconds) are recorded as follows: `frame_times = [16.7, 17.2, 33.3, 16.9, 16.8, 50.1, 16.6]`.

1. Calculate the average frame time.
2. Identify all frames exceeding 1.5x the average and count them as "stutters".
3. Compute the percentage of stutter frames relative to total frames.
4. Convert the average frame time to FPS (frames per second).

[GPU Memory Bandwidth] Exercise: A GPU with 256-bit bus width runs at 1750 MHz memory clock speed.

1. Calculate the peak memory bandwidth in GB/s using the formula: `(bus_width * memory_clock * 2)/ 8`.
2. Given a game texture streaming at 120 GB/s, determine if this GPU can handle it without bottlenecking.
3. If GDDR6X with 19 Gbps/pin is used instead, recalculate the bandwidth.

[Latency Optimization] Exercise: A multiplayer game has this network pipeline:

```
input_sample() -> process_input() -> send_packet() ->
network_transit(28ms) -> receive_packet() ->
apply_state_update() -> render_frame()
```

1. Identify all non-network latency sources assuming `process_input()` takes 3ms and `apply_state_update()` takes 2ms.
2. Propose one optimization to reduce latency in `send_packet()`.
3. Calculate total latency if `render_frame()` adds 12ms and all other steps remain unchanged.

[Ray tracing power consumption] Exercise:

1. Calculate the power consumption of a ray tracing operation that processes 10 million rays per second, given each ray requires 50 pJ of energy.
2. A GPU performs ray tracing at 80% utilization with a TDP of 250W. What percentage of power is consumed by ray tracing if other operations consume 120W?
3. Using the results from (1) and (2), determine how many rays per second could be processed if all available ray tracing power was used.

[BVH traversal optimization] Exercise:

1. Implement a simplified BVH traversal function in C++ that counts ray-box intersections:

```
int countIntersections(Ray ray, BVHNode node) {
    // Your implementation here
}
```

2. Given a scene with 1000 triangles organized in a BVH with average 5% empty volume, calculate the expected number of ray-box tests for one ray.
3. Modify the function from (1) to early-terminate when the ray's maximum distance is exceeded.

[Power-aware rendering] Exercise:

1. A mobile GPU switches between two ray tracing modes: Quality (10 rays/pixel, 15W) and Performance (4 rays/pixel, 8W). Calculate the energy savings for rendering a 1080p frame in Performance mode.
2. Derive the equation for energy-per-pixel as a function of rays-per-pixel, assuming power scales linearly with ray count.
3. Using `power_monitor.cpp`, measure the actual power draw difference between these modes on your system:

```
// power_monitor.cpp snippet
void measure_mode(Mode m) {
    start_power_log();
    render_scene(m);
    return end_power_log();
}
```

Chapter 11

Deep Dive into AMD Microarchitectures

11.1 Compute Units (CUs) and Shaders

[ALU Control Signals] Exercise: Design the control unit for a simple ALU that supports 4 operations: add, subtract, AND, and OR. The ALU takes two 8-bit inputs `data_a` and `data_b`, and produces an 8-bit output `result`. Complete the following tasks:

1. List the required control signals and their bit widths
2. Write the truth table for the control signals
3. Implement the control logic in VHDL using a `case` statement
4. Simulate the design for all operations with `data_a = 0x55` and `data_b = 0x0F`

[Pipeline Hazard Detection] Exercise: A 5-stage MIPS pipeline has the following hazard cases:

1. Identify all RAW hazards in the instruction sequence: `ADD R1, R2, R3`; `SUB R4, R1, R5`; `AND R6, R1, R7`
2. Calculate the required stall cycles for each hazard without forwarding
3. Modify the control unit to implement forwarding paths from EX/MEM and MEM/WB stages
4. Draw the pipeline diagram showing the resolved hazards

[Cache Controller FSM] Exercise: Design a finite state machine for a direct-mapped cache controller with the following specifications:

1. 4KB cache with 32-byte blocks
2. Write-back policy with dirty bit tracking
3. Support for hit/miss detection
4. Implement the FSM in Verilog with states for `IDLE`, `TAG_CHECK`, `MEM_READ`, and `MEM_WRITE`
5. Generate test cases for cache hits, clean misses, and dirty misses

[Vertex Shader Transformation] Exercise:

1. Write a GLSL vertex shader that applies a perspective projection matrix to vertex positions.
2. The shader must receive `model_view_matrix` and `projection_matrix` as uniforms.
3. Include proper transformation of normals using the inverse transpose of `model_view_matrix`.
4. Pass texture coordinates unchanged to the fragment shader.

```
// Your solution here
```

[Fragment Shader Lighting] Exercise:

1. Implement a Phong lighting model in a GLSL fragment shader.
2. Use `light_position`, `light_color`, and `view_pos` as uniforms.
3. Calculate diffuse using `max(dot(normal, light_dir), 0.0)`.
4. Add specular highlights with exponent 32.0.
5. Combine ambient (0.1), diffuse and specular components.

```
// Your solution here
```

[Compute Shader Parallel Processing] Exercise:

1. Create a compute shader that doubles all values in a `buffer_data` storage buffer.
2. Use `layout(local_size_x = 64)` for workgroup size.
3. Each invocation should process one element using `gl_GlobalInvocationID.x`.
4. Include memory barrier with `memoryBarrierBuffer()` after writes.

```
// Your solution here
```

11.2 Memory Architecture

HBM Bandwidth Calculation Exercise:

1. A GPU uses HBM2 with a `1024-bit` wide bus and `1.6_GHz` clock speed.
2. The memory uses `4_stack` configuration with `2_channels` per stack.
3. Calculate the total theoretical bandwidth in `GB/s`.
4. Assume double data rate (DDR) is applied. Show all steps.

HBM Power Efficiency Analysis Exercise:

1. An HBM2E module consumes `3.5_W` at `307_GB/s` bandwidth.
2. A GDDR6 module consumes `5.2_W` at `448_GB/s` bandwidth.
3. Calculate the power efficiency (bandwidth per watt) for both.
4. Compare the results and state which is more efficient.

HBM Address Mapping Exercise:

1. A system uses HBM with `8_bank_groups`, `4_banks` per group, and `32K_rows` per bank.
2. The memory controller receives a request for address `0x1F3A5C7E`.
3. Decode the address into bank group, bank, row, and column fields.
4. Assume a `32-bit` address with `3_bits` for bank group, `2_bits` for bank, and `15_bits` for row.

[Cache Hit Rate Calculation] Exercise: A GPU with Infinity Cache has the following characteristics:

- Total cache size: 128 MB
- Cache line size: 64 bytes
- Miss rate: 12% for a given workload
- Access latency: 10 cycles for a hit, 150 cycles for a miss

Calculate:

- The total number of cache lines in the Infinity Cache.
- The effective access time in cycles.
- The hit rate percentage.

[Bandwidth Improvement Analysis] Exercise: A system with Infinity Cache achieves a bandwidth of 1.5 TB/s under these conditions:

- Base bandwidth without cache: 800 GB/s
- Cache hit rate: 85%
- Cache miss penalty: 50 ns

Determine:

- The effective bandwidth improvement factor.
- The theoretical maximum bandwidth if hit rate reaches 100%.
- The miss rate at which the system would break even with base bandwidth.

[Cache Replacement Policy] Exercise: Implement a simplified LRU (Least Recently Used) replacement policy for Infinity Cache in Python:

- Create a class `InfinityCache` with methods `access(line)` and `evict()`.
- Use a dictionary to track cache lines and their access timestamps.
- When evicting, remove the line with the oldest timestamp.
- Handle cache misses by evicting if the cache is full.

```
class InfinityCache:
    def __init__(self, size):
        self.size = size
        self.cache = {}
        self.timestamp = 0

    def access(self, line):
        # Your implementation here
        pass

    def evict(self):
        # Your implementation here
        pass
```

11.3 Command Processors and Pipelines

[Rasterization Pipeline] Exercise: Implement a basic rasterization pipeline for a single triangle in software.

1. Define vertex positions for a 2D triangle in homogeneous coordinates using `float vertices[3][3]`.
2. Write a function `void transform_to_screen(float vertices[3][3], int width, int height)` that converts NDC to screen coordinates.
3. Implement edge function rasterization to determine pixel coverage.
4. Fill covered pixels with a solid color using `void set_pixel(int x, int y, uint8_t r, uint8_t g, uint8_t b)`.
5. Output a PPM image file showing the rendered triangle.

[Compute Shader Optimization] Exercise: Optimize a parallel reduction operation in a compute shader.

1. Create a compute shader that sums 1024 random float values stored in `RWStructuredBuffer input`.
2. First implement a naive version using one thread per element.
3. Then optimize using shared memory and sequential addressing patterns.
4. Compare performance between versions using GPU timestamps.
5. Modify the shader to handle input sizes that aren't power-of-two.

[Vulkan Pipeline Setup] Exercise: Create a minimal graphics pipeline in Vulkan.

1. Write vertex and fragment shaders in SPIR-V using `#version 450`.
2. Define a `VkPipelineVertexInputStateCreateInfo` for position and color attributes.
3. Configure `VkPipelineRasterizationStateCreateInfo` with backface culling enabled.
4. Set up a `VkPipelineColorBlendAttachmentState` for alpha blending.
5. Record commands to draw a triangle using this pipeline.

[Wavefront parallelism] Exercise: Consider a 3x3 matrix multiplication using wavefront parallelism. The computation follows the wavefront pattern where each element `C_i_j` depends on `A_i_k` and `B_k_j` for `k = 0,1,2`.

1. Draw the wavefront execution diagram for the 3x3 matrix multiplication, labeling the time steps and dependencies.
2. Identify the earliest time step at which `C_2_2` can be computed.
3. Write pseudocode using `#pragma omp parallel for` to parallelize the wavefront computation.

[GPU wavefront scheduling] Exercise: A GPU executes a wavefront of 64 threads (1 workgroup) for the following kernel:

```
__kernel void saxpy(float a, __global float* x, __global float* y) {
    int i = get_global_id(0);
    y[i] = a * x[i] + y[i];
}
```

1. Assuming no memory latency, calculate the minimum cycles needed to execute one wavefront on a GPU with 32 SIMD lanes.
2. If `x` and `y` are uncached, and memory latency is 200 cycles, estimate the total execution time for one wavefront.
3. Modify the kernel to use local memory for coalesced accesses, assuming `get_local_size(0)= 64`.

[Wavefront dependency resolution] Exercise: A stencil computation updates a 2D grid using a 5-point stencil:

```
for (int t = 0; t < T; ++t) {
    for (int i = 1; i < N-1; ++i) {
        for (int j = 1; j < N-1; ++j) {
            grid_new[i][j] = (grid_old[i-1][j] + grid_old[i+1][j]
                            + grid_old[i][j-1] + grid_old[i][j+1]) / 4;
        }
    }
    swap(grid_old, grid_new);
}
```

1. Identify the wavefront dependencies by drawing the anti-diagonal lines of parallelism for `N=5`.
2. Rewrite the inner loops to exploit wavefront parallelism using OpenMP directives.
3. Calculate the theoretical speedup for `N=1024` assuming infinite resources and no overhead.

Chapter 12

Programming for AMD GPUs

12.1 ROCm (Radeon Open Compute) Ecosystem

ROCm Profiler Setup Exercise:

1. Install ROCm Profiler on a Linux system using the official ROCm repository.
2. Launch the profiler and record GPU metrics for a simple `vector_add` kernel.
3. Export the profiling data as a CSV file and identify the kernel execution time.
4. Modify the kernel to reduce global memory accesses and reprofile.

ROCm Libraries Matrix Multiplication Exercise:

1. Write a C++ program using `rocBLAS` to multiply two 1024x1024 matrices.
2. Compare the performance with a CPU-only implementation using `std::vector`.
3. Use `rocBLAS` functions to transpose one matrix before multiplication.
4. Measure the speedup achieved by enabling `rocBLAS`'s tensor cores.

HIP Kernel Debugging Exercise:

1. Write a HIP kernel with intentional memory access errors (e.g., out-of-bounds).
2. Compile with `--amdgpu-target=gfx906` and debug using `rocgdb`.
3. Identify the faulty line using the debugger's backtrace feature.
4. Fix the kernel and validate correctness with `hipDeviceSynchronize`.

[GPU Memory Management] Exercise:

1. Write a CUDA kernel that copies elements from `input_array` to `output_array` using GPU global memory.
2. Modify the kernel to use shared memory for caching a 32-element tile before writing to `output_array`.
3. Add error checking for CUDA API calls and kernel launches using `cudaGetLastError`.

[OpenCL Device Query] Exercise:

1. Write an OpenCL program that queries and prints all available platforms.
2. For each platform, list all devices and their `CL_DEVICE_TYPE`.
3. Extract and print the `CL_DEVICE_MAX_COMPUTE_UNITS` for each device.

[SYCL Unified Memory] Exercise:

1. Create a SYCL buffer for a 1024-element array using unified shared memory (USM).
2. Write a SYCL parallel_for that squares each element in the array.
3. Add host-side verification code to check the results after kernel execution.

12.2 AMD GPUs with OpenCL

[Host-device communication] Exercise: Write an OpenCL host program that performs the following tasks:

1. Create an OpenCL context for the first available GPU device
2. Allocate two input buffers `input_buffer1` and `input_buffer2` each of size 1024 floats
3. Create an output buffer `result_buffer` of the same size
4. Write a kernel that adds corresponding elements from both input buffers and stores the result
5. Execute the kernel with 128 work-items per work-group
6. Read back and print the first 10 results from `result_buffer`

[Work-item synchronization] Exercise: Implement an OpenCL kernel that computes a prefix sum (scan) of an input array:

1. Declare a kernel with input buffer `input_array` and output buffer `prefix_sum`
2. Each work-item should process one element of the array
3. Use local memory and work-group barriers for synchronization
4. Implement the Hillis-Steele inclusive scan algorithm
5. Handle arrays of arbitrary size up to 4096 elements
6. Write the host code to verify the correctness with a test array of 16 elements

[Memory optimization] Exercise: Optimize matrix multiplication in OpenCL for a 64×64 matrix:

1. Create a kernel that multiplies matrices `matrixA` and `matrixB`
2. Use tiling with 16×16 local work-group size
3. Copy tiles to local memory before computation
4. Employ vector data types for global memory access
5. Compare performance with a naive implementation
6. Measure execution time using OpenCL events

File Path Handling Exercise:

1. Write a Python function `normalize_path(path)` that converts all backslashes to forward slashes and removes duplicate slashes.
2. Modify the function to handle both `C:\Users\file.txt` and `/home/user/file.txt` inputs correctly.
3. Add platform detection using `os.name` to automatically choose the correct path separator for the current OS.

Endianness Conversion Exercise:

1. Implement a C function `uint32_t swap_endian(uint32_t value)` that converts between little-endian and big-endian representations.
2. Write a test program that verifies the function works on both little-endian (x86) and big-endian (PowerPC) systems.
3. Create a preprocessor macro `IS_LITTLE_ENDIAN` that detects the system's endianness at compile time.

Line Ending Normalization Exercise:

1. Create a Java method `String normalizeLineEndings(String text)` that converts all line endings (CR, LF, CRLF) to the system's native format.
2. Write unit tests that verify correct operation on Windows (`\r\n`), Unix (`\n`), and legacy Mac (`\r`) inputs.
3. Add a parameter to force a specific line ending style regardless of the host OS.

Chapter 13

Performance Optimization in AMD GPUs

13.1 Profiling and Debugging Tools

GPU Timeline Analysis Exercise: Exercise:

1. Capture a GPU trace using RGP for a simple Vulkan application rendering a rotating cube.
2. Identify the longest-running `vkCmdDraw` call in the timeline and note its duration.
3. Locate the corresponding pipeline state in the RGP pipeline view and list its key parameters.
4. Export the shader ISA for the vertex shader used in this draw call.
5. Calculate the arithmetic intensity (FLOPs/byte) for this shader using the ISA statistics.

Memory Access Pattern Exercise: Exercise:

1. Profile a compute shader performing matrix multiplication using RGP.
2. In the memory view, identify the cache hit rates for `L1` and `L2`.
3. Modify the shader's workgroup size to improve cache utilization.
4. Re-profile and compare the new cache hit rates with the original values.
5. Explain how the changed workgroup size affected memory access patterns.

Wavefront Occupancy Exercise: Exercise:

1. Load an RGP trace of a fragment shader with low occupancy.
2. Identify the limiting factor (register pressure, memory latency, etc.) from the occupancy view.
3. Modify the shader to reduce register usage by splitting computations.
4. Re-capture the trace and verify improved occupancy in the wavefront view.
5. Measure the performance change in cycles per pixel before and after optimization.

CPU Performance Counters Exercise: Exercise:

1. Install AMD uProf on a Linux system with an AMD Ryzen processor.
2. Use `sudo apt-get install amduprof` to install the tool.
3. Run `amduprof-cli --list-events` to view available hardware performance counters.
4. Profile a simple C program (e.g., matrix multiplication) using the `amduprof-cli` command with `--event` flags for `RETIRED_INSTRUCTIONS` and `CPU_CLK_UNHALTED`.
5. Calculate the IPC (Instructions Per Cycle) from the collected data.

Memory Bandwidth Analysis Exercise: Exercise:

1. Write a C program that performs sequential and random memory accesses on a large array.
2. Compile the program with `gcc -O3` and run it under AMD uProf.
3. Use `amduprof-cli` with `--event MEMORY_CONTROLLER_READS` and `MEMORY_CONTROLLER_WRITES`.
4. Compare the bandwidth measurements between sequential and random access patterns.
5. Identify which memory access pattern causes higher DRAM controller activity.

Power Profiling Exercise: Exercise:

1. Download the AMD uProf Power Profiler add-on for your processor family.
2. Run a parallel OpenMP workload (e.g., `parallel for` loop) on 4 cores.
3. Use `amduprof-cli --power --cores 0-3` to monitor power consumption.
4. Record the average package power during workload execution.
5. Correlate power spikes with thread synchronization points in your code.

13.2 Identifying Bottlenecks

Buffer Overflow Prevention Exercise:

1. Write a C program that reads user input into a fixed-size buffer `char buf[64]` using `fgets`.
2. Modify the program to reject inputs longer than 63 characters with an error message.
3. Add a function that safely concatenates two strings without exceeding the buffer size.
4. Demonstrate how your solution prevents buffer overflow using `strcpy` as a counterexample.

Memory Fragmentation Analysis Exercise:

1. Implement a C program that allocates 1000 blocks of random sizes between 1-100 bytes using `malloc`.
2. Free every alternate block to create fragmentation.
3. Measure the remaining contiguous memory using `malloc_usable_size`.
4. Visualize the fragmentation pattern by printing allocated/free blocks as a bitmap.

Cache Optimization Exercise:

1. Create two versions of a matrix multiplication algorithm in C: one with `A[i][j]` access and one with `A[j][i]`.
2. Measure their execution times for 1000x1000 matrices using `clock_gettime`.
3. Explain the performance difference in terms of cache lines.
4. Rewrite the slower version using blocking (tiling) with 32x32 tiles.

Memory Access Patterns Exercise: Identify and fix execution inefficiencies in the following C code snippet that processes a 2D array:

```
void process_matrix(int matrix[100][100]) {
    for (int j = 0; j < 100; j++) {
        for (int i = 0; i < 100; i++) {
            matrix[i][j] *= 2;
        }
    }
}
```

item Explain why the current implementation causes poor cache utilization item Rewrite the code to improve memory access patterns item Calculate the theoretical cache miss reduction for a 64-byte cache line size

Branch Prediction Exercise: Analyze and optimize this sorting function for execution efficiency:

```
void sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

item Identify the main source of branch prediction misses item Modify the code to reduce branch mispredictions item Measure the performance improvement using `__builtin_expect` item Compare with a branch-free implementation using bitwise operations

Instruction-Level Parallelism Exercise: Optimize this matrix multiplication kernel for modern CPUs:

```
void matmul(int A[100][100], int B[100][100], int C[100][100]) {
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 100; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

item Reorder the loops to maximize instruction-level parallelism item Implement loop unrolling with a factor of 4 item Add SIMD intrinsics using `__m256i` item Measure the speedup from each optimization

13.3 Writing High-Performance GPU Code

[Memory Coalescing Optimization] Exercise:

1. Write a CUDA kernel to compute the sum of all elements in a 1D array `float A[N]`. Use shared memory for partial sums.
2. Modify the kernel to ensure memory coalescing by having thread `i` access `A[i]` instead of `A[blockIdx.x * blockDim.x + threadIdx.x]`.
3. Measure the performance difference between the original and coalesced versions using `cudaEvent_t` timers for `N = 1<<20`.

[Warp Divergence Analysis] Exercise:

1. Implement a CUDA kernel that processes an array `int flags[N]` where each thread checks `if (flags[i] > 0)` and performs `sqrt((float)flags[i])`.
2. Generate test data where `flags` alternates between 0 and non-zero values every 32 elements to simulate worst-case warp divergence.
3. Compare execution time with uniformly distributed non-zero values using `nvprof --metrics branch_efficiency`.

[Occupancy Calculator] Exercise:

1. Compute the theoretical occupancy for a kernel with 64 threads per block using `cudaOccupancyMaxPotentialBlockSize` on an NVIDIA A100 GPU.

2. Write a kernel using 32 registers per thread and 48KB shared memory per block. Calculate the occupancy limit due to register pressure.

3. Modify the kernel to reduce register usage via `__launch_bounds__` and verify the new occupancy with `nvcc --ptxas-options=-v`.

Chapter 14

Future Trends in AMD GPUs

14.1 RDNA 4 and Beyond

Modular facade system design Exercise:

1. Calculate the thermal transmittance (U-value) of a triple-glazed facade panel with the following layers: 6mm outer glass, 12mm argon gap, 6mm low-E coating glass, 16mm air gap, 8mm inner glass. Use conductivity values of 1.0 W/mK for glass and 0.016 W/mK for argon.
2. Design a parametric joint detail in `Grasshopper` that allows for ±15mm thermal movement between aluminum mullions and glass panels.
3. Write a `Python` script to analyze solar heat gain coefficients (SHGC) for three different glass types at 30°, 45°, and 60° sun angles.

Cross-laminated timber connection Exercise:

1. Determine the required number of `M20` steel dowels to transfer 85kN shear force in a CLT wall-to-floor connection, assuming a characteristic load-carrying capacity of 12.3kN per dowel.
2. Model a moment-resisting beam-column connection in `Revit` using standard steel brackets and concealed screws.
3. Create a fabrication drawing for a notched CLT beam splice connection showing all critical dimensions and fastener locations.

Adaptive reuse structural assessment Exercise:

1. Evaluate the load-bearing capacity of an existing 1950s concrete column (300×300mm, C20/25 concrete) for new office live loads of 4kN/m² using the reduced stress method.
2. Propose three retrofit solutions for a masonry bearing wall that needs to accommodate new `HVAC` ductwork penetrations.
3. Develop a `MATLAB` script to compare deflection profiles of original vs. reinforced concrete beams under point loads.

14.2 AMD in Machine Learning and AI

Memory Bandwidth Analysis Exercise: Exercise:

1. Calculate the theoretical memory bandwidth of an MI250X GPU given its 4096-bit memory interface and HBM2e memory speed of 1.6 Gbps.
2. Compare this with the bandwidth of an A100 GPU (5120-bit bus, 2.4 Gbps HBM2) and express the difference as a percentage.
3. Write a `rocm_smi` command to measure actual memory bandwidth utilization during kernel execution.

ROCm Kernel Optimization Exercise: Exercise:

1. Identify three key differences between CUDA and HIP programming models relevant to MI-series GPUs.
2. Convert the following CUDA kernel to HIP for MI200 series:

```
__global__ void add(float* a, float* b, float* c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}
```

3. Add appropriate error checking for HIP API calls in your converted code.

AI Workload Benchmarking Exercise: Exercise:

1. Install ROCm-supported version of PyTorch and verify MI250 GPU detection using `torch.cuda.is_available()`.
2. Benchmark ResNet-50 training on synthetic data with batch sizes 64, 128, and 256 using `torch.utils.benchmark.Timer`.
3. Plot the throughput (images/sec) versus batch size and explain any observed trends specific to MI-series architecture.

Chapter 15

Comparatison of AMD and NVIDIA Architectures

15.1 Introduction

GPU Architecture Comparison Exercise:

1. Identify three key architectural differences between AMD's RDNA 3 and NVIDIA's Ada Lovelace microarchitectures.
2. Calculate the theoretical FP32 throughput (in TFLOPS) for an NVIDIA RTX 4090 with 16,384 CUDA cores running at 2.52 GHz.
3. Using `rocm-smi` or `nvidia-smi`, write a Bash command to monitor real-time GPU utilization for an AMD or NVIDIA GPU respectively.

DLSS vs FSR Implementation Exercise:

1. Create a Unity script using `UnityEngine.Rendering` that toggles between NVIDIA DLSS and AMD FSR 2.0 at runtime.
2. Measure and compare the VRAM usage difference when running FSR 2.1 versus DLSS 3.0 at 4K resolution using `vkMemAlloc`.
3. Explain how to modify `CMakeLists.txt` to conditionally compile DLSS or FSR based on detected GPU vendor.

CUDA vs ROCm Porting Exercise:

1. Convert the following CUDA kernel to ROCm HIP:

```
__global__ void vecAdd(float* A, float* B, float* C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}
```

2. Benchmark the converted HIP kernel against the original CUDA version using `nvprof` and `rocprof`.
3. List three modifications required when porting CUDA's `thrust::device_vector` to ROCm's `hip::device_vector`.

Memory Allocation Comparison Exercise: Compare stack and heap memory allocation in C++ by:

1. Listing two advantages of stack allocation over heap allocation.
2. Writing a C++ code snippet using `new` to allocate an integer on the heap.
3. Writing a C++ code snippet showing automatic stack allocation of an integer.

4. Explaining when you must use heap allocation instead of stack.

File I/O Methods Exercise: Compare Python's `open()` and `pathlib` file handling:

1. Write Python code to create a file using `open()` with context manager.
2. Write equivalent code using `pathlib.Path`.
3. List one case where `open()` is preferable to `pathlib`.
4. List one operation that's simpler in `pathlib` than with `open()`.

Database Query Languages Exercise: Compare SQL and MongoDB query syntax:

1. Write a SQL query to find users with `age > 25` in a `users` table.
2. Write the equivalent MongoDB query for a `users` collection.
3. State one advantage of SQL's JOIN over MongoDB's `$lookup`.
4. State one advantage of MongoDB's document structure over SQL tables.

Historical context of modular design Exercise:

1. Identify three early programming languages (pre-1980) that pioneered modular design principles.
2. Compare the implementation of modules in `ALGOL_60` versus `Modula-2` using two concrete syntax examples.
3. Write a `Pascal` procedure demonstrating information hiding through nested scopes.

Object-oriented paradigm shift Exercise:

1. Convert the following `C` structure into a `C++` class with encapsulation:

```
struct Point {
    float x_coord;
    float y_coord;
};
```

2. Explain how the `Smalltalk` messaging model differs from `Java` method invocation.
3. Implement a `Python` class demonstrating polymorphism through operator overloading.

Modern functional design patterns Exercise:

1. Rewrite this imperative `JavaScript` loop using `Array.prototype.map`:

```
const numbers = [1, 2, 3];
const squared = [];
for (let i = 0; i < numbers.length; i++) {
    squared.push(numbers[i] * numbers[i]);
}
```

2. Contrast lazy evaluation in `Haskell` with eager evaluation in `Scheme` using list processing examples.
3. Implement a closure in `Rust` that maintains state between invocations.

15.2 Architectural Fundamentals

Matrix Multiplication Verification Exercise: Given two matrices A and B:

$$A = \begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix}$$

1. Compute the product $C = A \times B$ manually.
2. Write a Python function using `numpy` to verify your result.
3. Explain why the order of multiplication matters by computing $D = B \times A$.

Ohm's Law Circuit Analysis Exercise: A circuit consists of a voltage source $V = 12\,\text{V}$ and resistors $R_1 = 4\,\Omega$, $R_2 = 6\,\Omega$ in series.

1. Calculate the total resistance and current through the circuit.
2. Simulate the circuit in `LTspice` and compare the measured current with your calculation.
3. Modify the circuit to include $R_3 = 3\,\Omega$ in parallel with R_2 and repeat steps 1–2.

SQL Query Optimization Exercise: Given a database table `employees` with columns `employee_id`, `name`, `salary`, and `department_id`:

1. Write a SQL query to find the highest-paid employee in each department.
2. Explain how adding an index on `department_id` would improve performance.
3. Rewrite the query using a Common Table Expression (CTE) instead of a subquery.

[Data Parallelism Analysis] Exercise: Given the following C code snippet using SIMD intrinsics:

```
#include
void add_arrays(float* a, float* b, float* c, int n) {
    for (int i = 0; i < n; i += 8) {
        __m256 va = _mm256_load_ps(&a[i]);
        __m256 vb = _mm256_load_ps(&b[i]);
        __m256 vc = _mm256_add_ps(va, vb);
        _mm256_store_ps(&c[i], vc);
    }
}
```

1. Identify the SIMD vector width in bits and the number of floating-point elements processed per instruction.
2. Explain why the loop increment is `i += 8` instead of `i++`.
3. Rewrite the function to handle cases where `n` is not a multiple of 8.

[Thread Divergence Identification] Exercise: Consider the following CUDA kernel implementing a conditional operation:

```
__global__ void process_data(float* input, float* output, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        if (input[idx] > 0.5f) {
            output[idx] = sqrt(input[idx]);
        } else {
            output[idx] = input[idx] * input[idx];
        }
    }
}
```

1. Identify where thread divergence occurs in this kernel.
2. Calculate the warp execution efficiency if 60% of threads take the `sqrt` branch.
3. Propose an alternative implementation that reduces thread divergence.

[Memory Access Pattern Optimization] Exercise: The following OpenCL kernel performs matrix transposition:

```
__kernel void transpose(__global float* input,
                        __global float* output,
                        int width, int height) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    output[y * width + x] = input[x * height + y];
}
```

1. Analyze the memory access patterns for both input and output matrices.
2. Identify potential performance bottlenecks due to memory access.
3. Modify the kernel to use local memory for better memory access coalescing.

[Thread Pool Configuration] Exercise: Design a thread pool with a hierarchical structure for a web server application. The thread pool should have one master thread and three worker threads per CPU core.

1. Write pseudocode to initialize the thread pool using `pthread_create`.
2. Implement a task queue using `std::queue` protected by a mutex `std::mutex`.
3. Add a condition variable `std::condition_variable` to signal available tasks.
4. Modify the worker threads to process HTTP requests from the queue.
5. Include cleanup code to join all threads during shutdown.

[Pipeline Throughput Analysis] Exercise: Analyze a 4-stage image processing pipeline where each stage has different throughput:

1. Calculate the theoretical maximum throughput given stage latencies of 2ms, 5ms, 3ms, and 4ms.
2. Determine the bottleneck stage and compute the actual pipeline throughput.
3. Propose two solutions to balance the pipeline using thread scaling or batch processing.
4. Implement one solution using `OpenMP` parallel sections.
5. Measure the speedup using `std::chrono` high-resolution clock.

[Hierarchical Task Scheduling] Exercise: Implement a two-level scheduler for a scientific computing application:

1. Create 4 parent threads using `std::thread` for major computation domains.
2. Each parent thread should manage 8 child threads for sub-tasks.
3. Use `std::atomic` to implement a work-stealing queue between child threads.
4. Add a load-balancing mechanism that redistributes tasks when queues differ by more than 5 tasks.
5. Validate the implementation using a matrix multiplication benchmark with `1000x1000` matrices.

Memory Access Latency Analysis Exercise:

1. A GPU has the following memory hierarchy latencies: `global_memory` (400 cycles), `shared_memory` (40 cycles), and `L1_cache` (10 cycles). Compute the effective latency for a kernel where 70% of accesses hit in `L1_cache`, 20% in `shared_memory`, and 10% in `global_memory`.

2. If the `shared_memory` size is doubled, reducing its miss rate to 5%, recompute the effective latency assuming the `L1_cache` hit rate increases to 75%.
3. Write a CUDA code snippet that declares a `__shared__` float array of size 1024 and initializes all elements to zero.

Shared Memory Bank Conflict Exercise:

1. Identify potential bank conflicts in the following CUDA code:

```
__shared__ int s_data[32];
int idx = threadIdx.x * 2;
s_data[idx] = threadIdx.x;
```

2. Rewrite the code to avoid bank conflicts while maintaining the same functionality.
3. Calculate the maximum possible bank conflicts per warp for a `shared_memory` with 32 banks and a stride of 8 bytes.

Memory Coalescing Optimization Exercise:

1. Analyze the memory access pattern of the following CUDA kernel:

```
__global__ void copy(float* out, float* in) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    out[tid * 4] = in[tid * 4];
}
```

2. Modify the kernel to achieve perfect coalescing for 128-byte memory transactions.
3. Assuming a GPU with 128-bit wide memory buses, compute the theoretical speedup from your optimized version.

Memory Allocation Exercise:

1. Write a C function `void* allocate_matrix(int rows, int cols)` that dynamically allocates a 2D matrix of `rows x cols` integers using a single `malloc` call.
2. Modify the function to return `NULL` if either `rows` or `cols` is zero or negative.
3. Add error handling to check if `malloc` fails and return `NULL` in that case.
4. Write a corresponding `void free_matrix(void* matrix)` function to deallocate the memory.

Database Schema Exercise:

1. Design a SQL schema for a library management system with tables `books`, `authors`, and `borrowers`.
2. Include appropriate primary keys, foreign keys, and constraints (e.g., `NOT NULL`, `UNIQUE`).
3. Write a query to find all books borrowed by a specific borrower in the last 30 days.
4. Add an index to optimize the query from step 3 and explain why you chose that index.

Circuit Analysis Exercise:

1. Given a parallel RC circuit with `R = 1Ωk` and `C = µ10F`, calculate the impedance at `f = 50Hz`.
2. Plot the magnitude of the impedance versus frequency from `10Hz` to `10kHz` using Python's `matplotlib`.
3. Modify the circuit to include a series inductor `L = 100mH` and recalculate the impedance at `50Hz`.
4. Determine the resonant frequency of the modified RLC circuit.

[Wavefront Divergence Analysis] Exercise:

1. Write a CUDA kernel that simulates a divergent warp by conditionally executing different arithmetic operations based on `threadIdx.x \% 2`.
2. Measure the execution time of your divergent kernel vs. a uniform kernel using `cudaEventRecord`.
3. Calculate the performance penalty as a percentage difference between the two kernels.
4. Repeat the measurement for 64, 128, and 256 threads per block and tabulate the results.

[Wavefront Scheduling] Exercise:

1. Implement an OpenCL kernel that performs 40 independent `sqrt` operations per work-item.
2. Launch the kernel with 160 work-items on AMD hardware (report the specific GPU model used).
3. Use ROCm's `rocprof` to count the number of wavefronts executed.
4. Vary the work-group size from 64 to 256 in powers of 2 and plot wavefronts vs. work-group size.

[Warp Efficiency Optimization] Exercise:

1. Given a CUDA kernel with `if (threadIdx.x < 32){ ... } else { ... }`, rewrite it to eliminate warp divergence.
2. Verify your solution using NVIDIA's `nvprof --metrics achieved_occupancy`.
3. Compare the original and optimized versions using the metric `stall_inst_dependency`.
4. Calculate the L1 cache hit rate improvement (if any) using `nvprof --metrics l1_cache_hit_rate`.

[Instruction Encoding] Exercise:

1. Compare the instruction encoding formats for AMD GCN and NVIDIA PTX. List the key differences in operand fields and opcode structure.
2. Given a PTX instruction `add.s32 %r1, %r2, %r3;`, write the equivalent GCN assembly using `v_add_i32`.
3. Identify which ISA (GCN or PTX) uses a fixed 32-bit instruction width and which supports variable-length encodings.

[Memory Hierarchy] Exercise:

1. Explain how AMD RDNA's L0/L1 cache hierarchy differs from NVIDIA's unified L1/texture cache in a CUDA core.
2. Write a PTX code snippet to load a 32-bit value from global memory using `ld.global.u32` with an offset of 0x100.
3. Convert the PTX code from step 2 to GCN assembly, assuming the address is in `vaddr[0:1]` and the result goes to `vdata`.

[SIMD Execution] Exercise:

1. Describe how GCN's wavefronts (64 threads) differ from NVIDIA's warps (32 threads) in terms of SIMD execution.
2. Write a GCN assembly loop that performs `v_mul_f32` on 4 consecutive registers (`v0` to `v3`) with a scalar operand `s0`.
3. List two PTX instructions that would be required to achieve the same operation as step 2, using `mul.f32` and predicate registers.

[Compute Unit Resource Allocation] Exercise:

1. An AMD GPU has 40 Compute Units (CUs), each with 64 SIMD lanes. Calculate the total number of concurrent threads if each SIMD lane executes one thread.

2. A CU dispatches wavefronts of 64 threads. If a kernel requires 12,800 threads, how many wavefronts must be scheduled?

3. Modify the following OpenCL kernel to use `__attribute__((reqd_work_group_size(64, 1, 1)))` to enforce wavefront alignment:

```
__kernel void vec_add(__global float* A, __global float* B, __global float* C) {
    int id = get_global_id(0);
    C[id] = A[id] + B[id];
}
```

[SM Warp Scheduling Analysis] Exercise:

1. An NVIDIA SM processes 32-thread warps. Given a block with 256 threads, calculate the number of warps per block.

2. If an SM has 64 warps in flight and 4 warp schedulers, what is the maximum number of warps that can be issued per clock cycle?

3. Write a CUDA kernel with `<<<128, 256>>>` grid/block dimensions that uses `__syncthreads()` after a shared memory operation:

```
__global__ void reduce_sum(float* input, float* output) {
    __shared__ float partial_sum[256];
    int tid = threadIdx.x;
    partial_sum[tid] = input[blockIdx.x * 256 + tid];
    __syncthreads();
    // ... (rest of reduction omitted)
}
```

[Memory Hierarchy Comparison] Exercise:

1. An AMD CU has 32 KiB of local data share (LDS) per workgroup. If 4 workgroups share a CU, what is the maximum LDS per workgroup?

2. An NVIDIA SM has 96 KiB of shared memory. Convert this to bytes and calculate how many 4-byte floats it can store.

3. Write a HIP memory copy operation that transfers data between `host_ptr` and `device_ptr` using `hipMemcpyDeviceToHost`:

```
float* host_ptr = malloc(1024 * sizeof(float));
float* device_ptr;
hipMalloc(&device_ptr, 1024 * sizeof(float));
hipMemcpy(host_ptr, device_ptr, 1024 * sizeof(float), hipMemcpyDeviceToHost);
```

15.3 Memory System Comparisons

Matrix Multiplication Verification Exercise:

1. Implement matrix multiplication for two matrices `A` (3x2) and `B` (2x4) in Python using nested loops.

2. Validate your result by comparing with NumPy's `np.dot(A,B)` function.

3. Calculate the theoretical FLOP count for this operation and compare with your implementation's actual floating-point operations.

SQL Query Optimization Exercise:

1. Given a table `employees(id, name, department_id, salary)` and `departments(id, name)`, write a SQL query to find the highest-paid employee in each department.

2. Explain how adding an index on `department_id` would affect query performance.
3. Rewrite the query using a Common Table Expression (CTE) instead of a subquery.

Ohm's Law Circuit Analysis Exercise:

1. Design a series circuit with three resistors (`R1Ω=100`, `R2Ω=220`, `R3Ω=330`) and a 9V battery. Calculate total resistance and current.
2. Modify the circuit to parallel configuration and recalculate total resistance and individual branch currents.
3. Simulate both circuits using `LTspice` or `CircuitJS` and verify your calculations.

Cache hierarchy analysis Exercise:

1. Calculate the average memory access time for a system with L1 cache hit time of 2 cycles, L2 hit time of 8 cycles, and main memory access time of 100 cycles.
2. The L1 miss rate is 10% and L2 miss rate is 5%. Show all steps of your calculation.
3. Determine if adding an L3 cache with 30 cycle hit time and 2% miss rate would improve performance, assuming the L2 miss rate becomes 20% with this change.

Prefetching implementation Exercise:

1. Write a C function that implements stride prefetching for an array processing loop.
2. The function should take parameters for array base address, array size, and stride length.
3. Use `__builtin_prefetch` for prefetching and ensure proper prefetch distance.
4. Test your function with a sample array and measure cache misses using `perf stat`.

Memory system optimization Exercise:

1. Analyze the following memory access pattern: `0x1000, 0x1100, 0x1200, 0x1300, 0x1400`.
2. Determine if spatial locality is being exploited with a 64-byte cache line size.
3. Suggest two modifications to improve cache utilization for this pattern.
4. Calculate the potential reduction in memory accesses for a 1MB array processed with your optimized pattern.

Memory coalescing analysis Exercise:

1. Consider a CUDA kernel accessing a global memory array `float A[1024]` with thread ID `tid`. The access pattern is `A[tid * 2]`.
2. Calculate the number of memory transactions required for 32 threads in a warp under CUDA's 128-byte cache line size.
3. Rewrite the access pattern to achieve perfect coalescing while maintaining the same logical data access.
4. Explain how your modified pattern improves bandwidth utilization.

Bandwidth optimization for matrix transpose Exercise:

1. Given a 64x64 matrix stored in row-major order, implement a naive GPU kernel to transpose it.
2. Profile the kernel's memory bandwidth using `nvprof` and note the achieved bandwidth percentage.
3. Modify the kernel to use shared memory tiles of size `16x16`.
4. Compare the bandwidth utilization before and after optimization.
5. Identify the main source of bandwidth waste in the naive implementation.

Stride access optimization Exercise:

1. A kernel processes a 3D array `float data[256][256][256]` with access pattern `data[z][y][x]` where `x` varies fastest.
2. Analyze the memory access pattern when threads access elements with `z=threadIdx.x`, `y=threadIdx.y`, and `x=blockIdx.x`.
3. Propose an alternative data layout that improves coalescing for this access pattern.
4. Implement both versions and measure the runtime difference on a GPU with compute capability 7.0.

[Memory Allocation Strategies] Exercise: Compare the memory allocation behavior of `malloc` in C versus `new` in C++.

1. Write a C program that allocates an array of 100 integers using `malloc` and initializes them to zero.
2. Write an equivalent C++ program using `new` instead.
3. Measure the execution time of both programs using `clock()` in C and `std::chrono` in C++.
4. Explain why one might be faster than the other based on implementation differences.

[Thread Synchronization] Exercise: Implement a producer-consumer scenario in Python and Java to compare thread synchronization mechanisms.

1. Create a Python program using `threading.Lock` to synchronize access to a shared queue.
2. Create a Java equivalent using `synchronized` blocks.
3. Run both programs with 5 producer and 5 consumer threads for 1000 iterations each.
4. Compare the throughput (operations per second) and explain any performance differences.

|[File I/O Performance] Exercise: Benchmark binary file writing performance in Rust and Go.

1. Write a Rust program that creates a 1GB file filled with random bytes using `std::fs::File`.
2. Write a Go equivalent using `os.Create` and `rand.Read`.
3. Time both programs and compare their memory usage during execution.
4. Suggest reasons for any observed differences in speed or resource utilization.

Memory Bandwidth Calculation Exercise: Exercise:

1. AMD's Infinity Cache has a bandwidth of 2 TB/s and a hit rate of 80%. NVIDIA's texture cache has a bandwidth of 1.5 TB/s and a hit rate of 75%. Calculate the effective bandwidth for both caches.
2. If the Infinity Cache size is increased by 50%, the hit rate improves to 85%. Recalculate its effective bandwidth under this new condition.
3. Using `roofline_model.py`, plot the effective bandwidth vs. cache size for both architectures, assuming a linear hit rate improvement with cache size.

Cache Latency Comparison Exercise: Exercise:

1. Measure the latency of `textureFetch` in CUDA and `amd\_texture\_fetch` in HIP for a 1024x1024 image. Use `cudaEventRecord` and `hipEventRecord` for timing.
2. Repeat the measurement with cache-bypassing modes enabled (`cudaReadModeElementType` for NVIDIA, `AMD\_TEX\_CACHE\_BYPASS` for AMD).
3. Calculate the percentage latency difference between cached and uncached accesses for both architectures.

GPU Cache Simulation Exercise: Exercise:

1. Implement a simplified cache simulator in C++ that models both Infinity Cache (`infinity\_cache\_t`) and texture cache (`texture\_cache\_t`) with LRU replacement.

2. Feed the simulator with a trace of memory addresses from `gpu\_trace.log` and record hit/miss rates.
3. Modify the simulator to include a prefetcher for the Infinity Cache case (`infinity\_prefetch` flag) and compare the results.

[LDS vs. Global Memory Access] Exercise:

1. Write a CUDA kernel that computes the sum of squares of an input array using `__shared__` memory for intermediate results.
2. Modify the kernel to use Local Data Share (LDS) instead of global memory for the same computation.
3. Compare the execution times of both kernels for an array of size 4096 and explain the performance difference.

[LDS Bank Conflict Analysis] Exercise:

1. Write a CUDA kernel that demonstrates bank conflicts in LDS by forcing simultaneous accesses to the same memory bank.
2. Modify the kernel to avoid bank conflicts by adjusting the memory access pattern.
3. Measure the bandwidth improvement using NVIDIA's `nvprof` tool for both versions.

[Shared Memory Synchronization] Exercise:

1. Implement a CUDA kernel that performs a parallel reduction using `__syncthreads()` for synchronization.
2. Rewrite the kernel using LDS and compare the synchronization overhead with the shared memory version.
3. Analyze the impact of thread divergence on both implementations using the `--ptxas-options=-v` compiler flag.

15.4 Threading and Execution Models

[Thread Creation and Management] Exercise:

1. Write a Java program that creates three threads using the `Thread` class.
2. Each thread should print its name and a counter from 1 to 5 with a 500ms delay between iterations.
3. Use `Thread.sleep()` for the delay and handle `InterruptedException`.
4. Ensure the main thread waits for all three threads to complete using `join()`.

[Thread Synchronization] Exercise:

1. Implement a Python program with two threads incrementing a shared counter 1000 times each.
2. Use `threading.Lock()` to prevent race conditions.
3. Print the final counter value and verify it equals 2000.
4. Measure the execution time with and without the lock using `time.time()`.

[Thread Pool Execution] Exercise:

1. Create a C# program that uses `ThreadPool.QueueUserWorkItem` to process 10 tasks.
2. Each task should compute the square of its task ID (0-9) and print the result.
3. Use `ManualResetEvent` to block the main thread until all tasks complete.
4. Ensure the output shows non-sequential task completion due to thread pooling.

Wavefront vs. Warp Execution Analysis Exercise:

1. Compare the execution behavior of AMD's `Wave32` and `Wave64` wavefronts with NVIDIA's `warp` (32 threads).
2. Explain how divergent branches impact performance differently in a `Wave32` versus a `warp`.
3. Write a CUDA kernel where a `warp` diverges due to a conditional branch, and measure its IPC (instructions per cycle).
4. Write an equivalent HIP kernel for AMD GPUs using `Wave32` and compare the IPC with the CUDA result.

Occupancy Calculation Exercise:

1. Calculate the theoretical occupancy for an NVIDIA GPU with 64K registers per SM and a warp size of 32, given a kernel using 40 registers per thread and a block size of 128 threads.
2. Repeat the calculation for an AMD GPU with `Wave64`, assuming 32K registers per CU and the same kernel parameters.
3. Modify the kernel to reduce register usage to 20 per thread and recompute occupancy for both architectures.
4. Discuss how `Wave32` would affect register pressure compared to `Wave64`.

Memory Coalescing Comparison Exercise:

1. Write a CUDA kernel where threads in a `warp` access global memory with a stride of 2, and measure the effective bandwidth.
2. Rewrite the kernel for AMD's `Wave32`, ensuring the same access pattern, and measure bandwidth.
3. Adjust the stride to 1 (contiguous access) and repeat measurements for both architectures.
4. Explain how the wider `Wave64` might further impact memory coalescing efficiency.

Divergence in Vector Fields Exercise: Given the vector field $\mathbf{F} = x^2y\mathbf{i} + yz^2\mathbf{j} + zx^2\mathbf{k}$:

1. Compute the divergence $\nabla \cdot \mathbf{F}$.
2. Evaluate $\nabla \cdot \mathbf{F}$ at the point $(1, -1, 2)$.
3. Determine if the field is compressible (i.e., $\nabla \cdot \mathbf{F} = 0$) at $(1, -1, 2)$.

Finite Difference Divergence Exercise: A 2D velocity field is discretized on a grid with spacing $\Delta x = \Delta y = 0.1$:

```
u[i,j] = i * Delta_x * j * Delta_y  # x-component
v[i,j] = sin(i * Delta_x) + cos(j * Delta_y)  # y-component
```

1. Write the finite difference expression for $\nabla \cdot \mathbf{v}$ at point (i, j).
2. Compute $\nabla \cdot \mathbf{v}$ at $(i, j) = (2, 3)$ using central differences.
3. State whether the flow is locally divergent or convergent at this point.

Divergence-Free Field Construction Exercise:

1. Verify if $\mathbf{G} = \nabla \times (yz\mathbf{i} + zx\mathbf{j} + xy\mathbf{k})$ is divergence-free.
2. Modify the z-component of $\mathbf{G}$ to $\mathbf{G}_z = xy + f(x, y)$ such that $\nabla \cdot \mathbf{G} = 0$.
3. Propose a function $f(x, y)$ that satisfies the divergence-free condition.

Wavefront Divergence Analysis Exercise:

1. Write a CUDA kernel that demonstrates warp divergence by branching on `threadIdx.x % 2`.
2. Modify the kernel to use NVIDIA's `__shfl_xor_sync` to mitigate the divergence.

3. Write an equivalent HIP kernel for AMD GPUs using `__shfl_xor` and compare the assembly output using `rocobjdump`.

Occupancy Calculator Exercise:

1. Calculate the theoretical occupancy for an NVIDIA V100 GPU given: 64 warps per SM, 2048 threads per SM, and a kernel using 32 registers per thread.
2. Repeat the calculation for an AMD MI100 GPU with 40 wavefronts per CU, 2560 threads per CU, and the same register usage.
3. Implement a CUDA `__global__ void occupancy_test()` kernel that empirically verifies your calculations using `cudaOccupancyMaxActiveBlocksPerMultiprocessor`.

Masked Wavefront Operation Exercise:

1. Write an OpenCL kernel that performs a conditional reduction using AMD's `__ballot` instruction with a `wavefront_mask`.
2. Port the kernel to CUDA using `__ballot_sync` with appropriate warp masks.
3. Benchmark both versions on respective hardware using `clGetEventProfilingInfo` and CUDA events.

Preemptive Scheduling Analysis Exercise: Consider a system with three processes having the following properties:

- Process `P1`: Arrival time = 0, Burst time = 8, Priority = 2
- Process `P2`: Arrival time = 2, Burst time = 4, Priority = 1
- Process `P3`: Arrival time = 4, Burst time = 6, Priority = 3

Calculate the average waiting time for:

- Priority scheduling (higher number = higher priority)
- Round Robin scheduling with time quantum = 3

Real-Time System Feasibility Exercise: A real-time system has four periodic tasks:

- Task `T1`: Execution time = 2ms, Period = 6ms
- Task `T2`: Execution time = 3ms, Period = 12ms
- Task `T3`: Execution time = 1ms, Period = 24ms
- Task `T4`: Execution time = 4ms, Period = 48ms

Perform the following:

- Verify schedulability using Rate Monotonic Scheduling
- Calculate the total processor utilization
- Determine if all deadlines can be met

Multilevel Queue Implementation Exercise: Implement a Python class `MultilevelQueue` with:

- Three queues: `system` (highest priority), `interactive`, `batch` (lowest)
- Method `enqueue(process, queue_type)` to add processes
- Method `dequeue()` that always takes from highest non-empty queue

Write the class definition and test it with:

```
p1 = {"pid": 1, "burst": 3}
p2 = {"pid": 2, "burst": 2}
p3 = {"pid": 3, "burst": 4}
```

[AMD GCN Architecture] Exercise:

1. Explain how AMD's Asynchronous Compute Engines (ACEs) enable concurrent execution of compute and graphics workloads. List the key hardware components involved.
2. Write a ROCm HIP kernel that launches two independent compute tasks (e.g., vector addition and matrix multiplication) to demonstrate ACE utilization. Use `hipStreamCreate` and `hipStreamNonBlocking`.
3. Compare the latency hiding mechanisms in AMD's ACEs versus NVIDIA's warp schedulers when handling memory-bound workloads.

[Warp Scheduling Analysis] Exercise:

1. Describe how NVIDIA's SIMT architecture schedules warps on SM cores during stall conditions (e.g., memory latency). Include the role of the `warp_scheduler` unit.
2. Write a CUDA kernel with artificial dependencies (using `__syncthreads()`) that forces warp serialization. Measure performance impact using `nvprof`.
3. Calculate the theoretical occupancy for a kernel with 128 threads per block and 32 registers per thread on an NVIDIA GPU with 64K registers and 16 SM cores.

[Cross-Platform Optimization] Exercise:

1. Convert a given CUDA kernel (using `warp-shfl` operations) to AMD HIP, replacing warp-level primitives with ROCm equivalents like `__shfl_xor`.
2. Profile both versions using `nsight` (NVIDIA) and `rocprof` (AMD). Tabulate IPC and memory bandwidth metrics.
3. Propose three architecture-specific optimizations for the HIP version to better utilize ACEs, considering the GCN's 64-wide wavefronts.

15.5 Performance Optimization Analogies

[Graph Coloring] Exercise:

1. Construct the interference graph for the following code snippet. Label nodes with variable names and edges with conflicts.

```
a = b + c
d = a * e
f = d - b
g = f + e
```

2. Assign registers to variables using graph coloring with 3 available registers (`r1`, `r2`, `r3`). Show the coloring steps.
3. List any spilled variables if the graph cannot be colored with 3 registers.

[Live Range Analysis] Exercise:

1. Compute live ranges for all variables in the following basic block:

```
x = y + z
w = x * 2
y = w - z
z = y + x
```

2. Draw the live ranges on a numbered instruction timeline.
3. Identify which variable pairs interfere based on overlapping live ranges.

[Spill Cost Heuristic] Exercise:

1. Given the following code and access frequencies:

```
t1 = a + b      (frequency: 20)
t2 = t1 * c     (frequency: 15)
a = t2 / d      (frequency: 10)
```

2. Calculate spill costs for `t1`, `t2`, and `a` using the formula: `cost = frequency * (loads + stores)`.
3. If only 2 registers are available, which variable should be spilled based on your calculations?

[VGPR/SGPR Allocation] Exercise:

1. Compute the maximum number of vector registers (`VGPRs`) per work-item for an AMD GPU with 256 KB of vector register file and 64 work-items per wavefront.
2. Given a kernel using `SGPRs` for scalar operands, calculate the total register pressure if each work-item requires 10 scalar registers and the wavefront has 32 lanes.
3. Determine the occupancy impact if an AMD GPU has a limit of 1024 `VGPRs` per wavefront and your kernel uses 128 registers per work-item in a 64-lane wavefront.

[NVIDIA Register Bank Conflict] Exercise:

1. Identify potential register bank conflicts in the following NVIDIA PTX code snippet:

```
ld.global.f32 %f0, [%rd0];
ld.global.f32 %f1, [%rd1];
add.f32 %f2, %f0, %f1;
ld.global.f32 %f3, [%rd2];
mul.f32 %f4, %f2, %f3;
```

2. Rewrite the PTX code to minimize bank conflicts while maintaining correctness.
3. Calculate the theoretical speedup if the modified code reduces bank conflicts from 4 cycles to 1 cycle per instruction.

[Register Spilling Analysis] Exercise:

1. Estimate the performance penalty of spilling 16 `VGPRs` to global memory on an AMD GPU, assuming each spill operation takes 100 cycles and occurs once per wavefront.
2. Compare the spilling overhead for `SGPRs` versus `VGPRs` if scalar spills take 20 cycles but occur twice as frequently.
3. Propose a kernel optimization to reduce register pressure by 25% without altering the algorithm's functionality.

Cache Line Alignment Exercise: Exercise:

1. Write a C function `void sum_array(int* arr, int size)` that sums all elements of an array.
2. Allocate two arrays `int a[128]` and `int b[129]` filled with random values.
3. Measure the execution time of `sum_array(a, 128)` vs `sum_array(b, 129)` using `clock_gettime()`.
4. Explain why one version might be faster, assuming a 64-byte cache line size.

Pointer Aliasing Exercise: Exercise:

1. Implement two versions of a matrix addition function: `void add_matrices_restrict(float* A, float* B, float* C)` and `void add_matrices_aliased(float* A, float* B, float* C)`.
2. The restricted version must use the `restrict` keyword.

3. Create a 1024x1024 matrix where `B = A + 1` (creating overlapping regions).
4. Compare the assembly output (`-S` flag) and execution times of both versions.

TLB Miss Analysis Exercise: Exercise:

1. Write a program that allocates a 2D array `int arr[1024][1024]`.
2. Initialize it with `arr[i][j] = i + j` using row-major and column-major traversal.
3. Measure page faults using `perf stat -e dTLB-load-misses` for both traversals.
4. Calculate the expected TLB misses given a 4KB page size and 64-entry TLB.

Memory coalescing optimization Exercise:

1. Write a CUDA kernel that performs element-wise addition of two arrays `A` and `B`, storing the result in array `C`.
2. Modify the kernel to ensure perfect memory coalescing by using shared memory for temporary storage.
3. Measure the performance difference between the naive and optimized versions using `nvprof` for arrays of size 1 million elements.

Scatter operation implementation Exercise:

1. Implement a scatter operation in CUDA where each thread writes to a random location in output array `D` based on values from index array `indices`.
2. Analyze the memory access pattern and identify potential bottlenecks.
3. Optimize the scatter operation using atomic operations and compare the results with the naive implementation.

Gather operation with strided access Exercise:

1. Create a CUDA kernel that gathers elements from array `input_data` using a stride of `k` elements between consecutive reads.
2. Profile the kernel for different stride values (1, 2, 4, 8, 16) and record the execution times.
3. Rewrite the kernel to use texture memory and compare the performance with the global memory version.

Flow Rate Calculation Exercise: Exercise:

1. A pipeline transports crude oil with a density of `850_kg/m^3`.
2. The pipe has an inner diameter of `0.5_m` and a flow velocity of `2_m/s`.
3. Calculate the volumetric flow rate in `m^3/s`.
4. Convert the result to barrels per day (`1_bbl = 0.158987_m^3`).
5. Verify your calculation using the continuity equation `Q = A * v`.

Pump Power Requirement Exercise: Exercise:

1. Water (`1000_kg/m^3`) is pumped through a `300_m` long pipeline.
2. The pipe has a friction factor `f = 0.02` and diameter `0.3_m`.
3. The required flow rate is `0.1_m^3/s` with a `50_m` elevation gain.
4. Calculate the head loss using the Darcy-Weisbach equation.
5. Determine the pump power required in kW assuming `75\%` efficiency.

Parallel Pipeline Optimization Exercise: Exercise:

1. Two parallel pipes (`D1 = 0.4_m`, `D2 = 0.6_m`) transport the same fluid.

2. Both pipes are `1000_m` long with the same friction factor `f = 0.015`.

3. The total flow rate is `1.2_m^3/s`.

4. Calculate the flow rate in each pipe using the equivalent resistance method.

5. Verify that the pressure drop is equal in both branches.

[Loop Unrolling Implementation] Exercise: Implement loop unrolling for the following C code snippet to reduce loop overhead. Assume the loop count `N` is always a multiple of 4. Modify the code to process 4 elements per iteration. Use `#pragma unroll` if available or manually unroll the loop. The original code is:

```
for (int i = 0; i < N; i++) {
    output[i] = input[i] * coefficient;
}
```

1. Write the unrolled version of the loop without using compiler pragmas

2. Measure the execution time difference between original and unrolled versions

3. Explain why unrolling might not improve performance for very small `N`

[Instruction Scheduling Analysis] Exercise: Analyze the following assembly code snippet from a RISC-V processor:

```
loop:
    ld x5, 0(x10)       # Load from memory
    addi x10, x10, 8    # Increment pointer
    mul x6, x5, x7      # Multiply operation
    addi x11, x11, -1   # Decrement counter
    bnez x11, loop      # Branch if not zero
```

1. Identify all data dependencies between instructions

2. Propose a rescheduled version with better pipeline utilization

3. Calculate the potential cycle savings per iteration

[SIMD Parallelization] Exercise: Convert the following scalar floating-point computation to use SIMD instructions:

```
void normalize(float* array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = array[i] / 255.0f;
    }
}
```

1. Write the equivalent function using Intel AVX intrinsics

2. Ensure proper alignment handling for the input array

3. Compare the performance against the scalar version

15.6 Development Ecosystems

Memory Management Comparison Exercise: Exercise:

1. Write a ROCm HIP kernel to allocate `float* device_array` on the GPU using `hipMalloc` and copy data from `float* host_array` using `hipMemcpy`.
2. Rewrite the same functionality using CUDA's `cudaMalloc` and `cudaMemcpy`.
3. Compare the API signatures of `hipMemcpy` and `cudaMemcpy` by listing their parameters and return types.

Kernel Launch Configuration Exercise: Exercise:

1. Write a HIP kernel `__global__ void add_vectors(float* A, float* B, int N)` that adds two vectors element-wise.
2. Launch the kernel with ROCm using `hipLaunchKernelGGL`, configuring 256 threads per block and enough blocks to cover `N = 1024` elements.
3. Repeat the launch configuration using CUDA's `<<>>` syntax with identical parameters.

Error Handling Implementation Exercise: Exercise:

1. Write a HIP function to check for errors using `hipGetLastError` and `hipDeviceSynchronize` after a kernel launch.
2. Implement the equivalent error checking in CUDA using `cudaGetLastError` and `cudaDeviceSynchronize`.
3. Modify both functions to print the error string using `hipGetErrorString` or `cudaGetErrorString` if an error occurs.

License compatibility analysis Exercise: Compare the GNU GPLv3 and MIT licenses for software distribution.

1. List three key differences between the GPLv3 and MIT licenses.
2. Explain how the `copyleft` clause in GPLv3 affects derivative works.
3. A developer wants to integrate `library_X` (GPLv3) into `project_Y` (MIT). Identify the legal conflict.

Build system integration Exercise: Implement a hybrid development environment using open-source and proprietary tools.

1. Write a `CMakeLists.txt` snippet that links both a proprietary `lib_vendor.so` and open-source `lib_openssl.a`.
2. Configure environment variables to isolate paths for proprietary toolchains using `.bashrc` commands.
3. Measure compilation time differences between `gcc` and a proprietary compiler using `time make`.

API protocol reverse engineering Exercise: Analyze a proprietary network protocol using open-source tools.

1. Capture USB traffic from a device using `wireshark` and export to `.pcap` format.
2. Identify repeating byte patterns in the capture using `tshark -T fields -e usb.capdata`.
3. Reimplement the protocol's checksum algorithm in Python using the `struct` module.

Matrix Multiplication Performance Comparison Exercise:

1. Write an OpenCL kernel to perform matrix multiplication for square matrices of size `N x N`.
2. Write an equivalent HIP kernel for the same matrix multiplication operation.
3. Create a host program that measures execution time for both kernels using `clGetEventProfilingInfo` and `hipEventElapsedTime`.
4. Run tests for matrix sizes 256, 512, and 1024, recording the performance results.

5. Compare the performance between OpenCL and HIP implementations on your hardware.

Platform Detection and Device Query Exercise:

1. Write an OpenCL program that lists all available platforms and devices using `clGetPlatformIDs` and `clGetDeviceIDs`.
2. Write a HIP program that lists all available HIP-capable devices using `hipGetDeviceCount` and `hipGetDeviceProperties`.
3. Modify both programs to display compute capability (OpenCL version or HIP architecture).
4. Add functionality to detect whether the current device supports double precision.
5. Output the results in a formatted table comparing OpenCL and HIP device information.

Cross-platform Reduction Kernel Exercise:

1. Implement a parallel reduction kernel in OpenCL that works on any OpenCL 1.2+ device.
2. Implement the same reduction algorithm in HIP targeting AMD and NVIDIA GPUs.
3. Both kernels should support arbitrary input sizes (not just power-of-two).
4. Add host code to verify both implementations produce identical results.
5. Benchmark both kernels with input sizes from 1K to 1M elements.

Memory Leak Detection Exercise:

1. Write a C program that allocates memory for an array of 100 integers using `malloc` but does not free it.
2. Compile the program with `gcc -g` and run it under `valgrind --leak-check=full`.
3. Identify the line number where the memory leak occurs from the `valgrind` output.
4. Modify the program to free the allocated memory and verify the fix using `valgrind`.

Performance Profiling Exercise:

1. Write a Python function `slow_sum(n)` that calculates the sum of the first `n` integers using a loop.
2. Profile the function using `cProfile.run('slow_sum(1000000)')` and note the execution time.
3. Optimize the function by replacing the loop with the formula `n*(n+1)//2`.
4. Re-profile the optimized function and compare the execution times.

Debugging Segmentation Faults Exercise:

1. Write a C program that attempts to write to a NULL pointer, causing a segmentation fault.
2. Compile the program with `gcc -g` and run it under `gdb`.
3. Use `gdb` commands `backtrace` and `print` to identify the faulty line.
4. Fix the program by initializing the pointer properly and verify the fix.

Profiling Workloads with Radeon GPU Profiler Exercise:

1. Install AMD Radeon GPU Profiler and capture a trace of a simple OpenCL kernel that performs vector addition.
2. Identify the kernel dispatch latency in the trace and compare it to the actual kernel execution time.
3. Use the profiler to determine the occupancy of the kernel and list the limiting factors reported by the tool.
4. Export the trace data to a CSV file and plot the GPU utilization over time using Python.

Nsight Compute Kernel Analysis Exercise:

1. Write a CUDA kernel that computes matrix multiplication using shared memory (tiling).
2. Profile the kernel using NVIDIA Nsight Compute and record the achieved FLOP/s.
3. Locate the `ld.shared` and `st.shared` instructions in the SASS output and count their occurrences.
4. Modify the kernel to reduce shared memory bank conflicts and re-profile to measure the improvement.

Cross-Vendor Profiler Comparison Exercise:

1. Implement a histogram calculation kernel in both OpenCL (for AMD) and CUDA (for NVIDIA).
2. Profile both implementations using their respective vendor tools (Radeon GPU Profiler and Nsight).
3. Compare the reported memory bandwidth utilization between the two platforms.
4. Create a table listing the top three performance bottlenecks identified by each profiler.

15.7 Cross-Vendor Programming and Trends

[Vulkan Instance Creation] Exercise:

1. Write a minimal Vulkan program that creates a `VkInstance` with validation layers enabled.
2. Populate the `VkApplicationInfo` struct with your application name, version 1.0, and Vulkan API version 1.2.
3. Enable the `VK_LAYER_KHRONOS_validation` layer and the `VK_EXT_debug_utils` extension.
4. Verify instance creation by checking the return value of `vkCreateInstance`.
5. Destroy the instance before program termination.

[SPIR-V Shader Compilation] Exercise:

1. Install the Vulkan SDK and locate the `glslc` compiler in your environment.
2. Write a vertex shader in GLSL that transforms vertices using a uniform buffer containing a `mat4` MVP matrix.
3. Compile the shader to SPIR-V using `glslc -O -o shader.vert.spv shader.vert`.
4. Write a fragment shader that outputs a solid red color and compile it similarly.
5. Use `spirv-dis` to inspect the generated SPIR-V bytecode for both shaders.

[Pipeline Setup] Exercise:

1. Create a `VkPipelineLayout` with one descriptor set layout binding a uniform buffer at binding point 0.
2. Define vertex input attributes for positions (location 0) and colors (location 1), both as `vec3`.
3. Configure a graphics pipeline with the shaders from the previous exercise.
4. Set the viewport and scissor to match a 800x600 window.
5. Specify a render pass with a single color attachment using `VK_FORMAT_B8G8R8A8_SRGB`.

Matrix Multiplication Optimization Exercise: Exercise: Compare the theoretical peak performance of AMD MI250X and NVIDIA A100 for mixed-precision matrix multiplication. Assume the following specifications:

- AMD MI250X: 383 TFLOPS FP16, 47.9 TFLOPS FP32, matrix cores enabled.
- NVIDIA A100: 312 TFLOPS FP16, 19.5 TFLOPS FP32, Tensor Cores enabled.

- Both use `bfloat16` for storage and FP32 for accumulation.

Calculate:

1. The effective throughput for `C = A × B` where A is `bfloat16` and B is `bfloat16`.
2. The memory bandwidth requirement for a 4096×4096 matrix multiplication.
3. The expected performance difference when using `tf32` precision on both architectures.

ROCm vs. CUDA Implementation Exercise: Exercise: Implement a simple GEMM kernel using:

- AMD's `rocBLAS` library with MI250X optimizations.
- NVIDIA's `CUBLAS` library with Tensor Core support.

Requirements:

1. Create a 1024×1024 matrix multiplication benchmark in C++.
2. Use `__half` data type for both implementations.
3. Measure execution time with `std::chrono`.
4. Compare the results against theoretical peak performance.
5. Include error checking for GPU API calls.

Deep Learning Workload Profiling Exercise: Exercise: Profile a ResNet-50 training session on both architectures:

- Use `torch.profiler` for NVIDIA GPUs with Tensor Cores.
- Use `rocprof` for AMD GPUs with matrix cores.

Tasks:

1. Record time spent in convolutional layers vs. fully connected layers.
2. Measure memory bandwidth utilization during backpropagation.
3. Identify any operations that don't utilize matrix/Tensor cores.
4. Compare the percentage of FP16 vs. FP32 operations.
5. Generate a heatmap of computational intensity across layers.

Ray-Sphere Intersection Implementation Exercise:

1. Write a function `bool ray_sphere_intersect(vec3 ray_origin, vec3 ray_dir, vec3 sphere_center, float radius)` in C++ that returns true if a ray intersects a sphere.
2. The function should handle cases where the ray origin is inside the sphere.
3. Add an output parameter `float& t` to store the intersection distance along the ray.
4. Handle edge cases where the ray is tangent to the sphere.
5. Optimize your solution by early termination when possible.

Real-Time Denoising Filter Exercise:

1. Implement a 3x3 Gaussian filter in GLSL for denoising ray-traced images.
2. The shader should sample from a texture `uniform sampler2D noisy_input`.
3. Use a sigma value of 1.5 for the Gaussian kernel weights.
4. Apply the filter only to the RGB channels, preserving the alpha channel.

5. Benchmark the performance impact on a 1080p render target.

Acceleration Structure Validation Exercise:

1. Create a BVH visualization tool that draws AABB boxes for all nodes.
2. Highlight leaf nodes containing more than 5 triangles in red.
3. Add a toggle to show/hide intermediate levels of the BVH.
4. Implement a consistency check that verifies all triangles are contained within their parent nodes.
5. Measure and display the percentage of empty volume in each AABB.

15.8 Conclusion

Matrix Operations Comparison Exercise:

1. Given matrices `A = [[1, 2], [3, 4]]` and `B = [[5, 6], [7, 8]]`, compute the element-wise product `A * B`.
2. Compute the matrix product `A @ B` using standard matrix multiplication.
3. Explain why the results from steps 1 and 2 differ, with reference to the subsection title.

Circuit Analysis Exercise:

1. For the circuit with `R1 = 100 ohms`, `R2 = 200 ohms` in series, calculate total resistance.
2. Now reconfigure `R1` and `R2` in parallel and recalculate.
3. Compare the two configurations using the subsection title's criteria.

Database Query Exercise:

1. Write a SQL query to select all records from `employees` where `salary > 50000`.
2. Write a second query using `JOIN` on `departments` and `employees` tables.
3. Explain how the result sets differ, referencing the subsection title.

Memory Alignment Optimization Exercise: Exercise:

1. Write a C function `void* aligned_malloc(size_t size, size_t alignment)` that allocates memory aligned to a specified boundary.
2. The function must return `NULL` if allocation fails.
3. Use `posix_memalign` if available, otherwise implement manual alignment using `malloc` and bitwise operations.
4. Include error handling for alignment values that are not powers of two.
5. Write a corresponding `void aligned_free(void* ptr)` function to deallocate memory.

Compiler Flag Benchmarking Exercise: Exercise:

1. Create a matrix multiplication benchmark using `float[1024][1024]` arrays.
2. Implement three versions: naive, loop-unrolled, and SIMD-optimized.
3. Measure execution time using `clock_gettime(CLOCK_MONOTONIC)`.
4. Compile with `-O0`, `-O2`, and `-O3` flags for each version.
5. Tabulate results showing speedup relative to `-O0` for each optimization level.

Vendor-Specific Intrinsic Porting Exercise: Exercise:

1. Identify equivalent SIMD intrinsics for `_mm256_load_ps` across Intel, ARM NEON, and PowerPC Altivec.
2. Write a portable function `void load_vector(float* dst, float* src)` using preprocessor macros.
3. Include architecture detection via `__AVX__`, `__ARM_NEON`, and `__ALTIVEC__`.
4. Handle misaligned memory cases with fallback to scalar operations.
5. Verify correctness by comparing output against reference implementation.

Cross-platform file path handling Exercise: Write a Python script that:

1. Accepts a user input path (either Unix-style or Windows-style) via `sys.argv`.
2. Normalizes the path to use forward slashes using `os.path.normpath`.
3. Prints whether the path is absolute or relative.
4. Creates an empty file at that path if it doesn't exist.

Memory-efficient data processing Exercise: Implement a Java program that:

1. Reads a large CSV file specified by `args[0]` using a `BufferedReader`.
2. Processes each line immediately without storing all lines in memory.
3. Counts occurrences of a target string `"ERROR"` case-insensitively.
4. Writes the final count to `System.out`.

Compiler optimization flags Exercise: Write a C program and Makefile that:

1. Contains a `compute_factorial` function with iterative implementation.
2. Builds with `-O0` and `-O3` flags in separate targets.
3. Measures execution time using `clock()` for both versions.
4. Prints the timing results with `printf` formatted as `"Optimized: %f ms\n"`.

Chapter 16

GPU Assembly Fundamentals

16.1 GPU ISA Architecture Deep Dive

[Instruction Encoding] Exercise:

1. Convert the MIPS instruction `add $t0, $t1, $t2` to its 32-bit binary machine code representation. Use the MIPS reference sheet provided.
2. Identify the opcode, rs, rt, rd, shamt, and funct fields in your binary result from part 1.
3. Calculate the hexadecimal representation of the encoded instruction.

[Immediate Value Handling] Exercise:

1. Encode the MIPS instruction `addi $s0, $s1, -42` in binary, handling the negative immediate value correctly.
2. Determine the maximum positive and negative values that can be represented in the immediate field of this instruction.
3. Explain how the assembler would handle the pseudo-instruction `li $v0, 0xABCD1234`.

[RISC-V Format Identification] Exercise:

1. Classify the following RISC-V instructions into R-type, I-type, S-type, U-type, or J-type formats: `add x5, x6, x7`, `ld x8, 16(x9)`, `jal x1, label`.
2. For the instruction `sw x10, 32(x11)`, show how the 12-bit immediate is split across the instruction fields.
3. Write the binary encoding for `beq x12, x13, offset`, assuming the offset is 0x4C.

Pipeline Hazard Identification Exercise: Identify the type of hazard (structural, data, or control) in each scenario below. For data hazards, specify RAW, WAR, or WAW.

1. A `MUL` instruction writes to `R1`, followed by an `ADD` that reads `R1`.
2. Two `LD` instructions attempt to access the same cache bank simultaneously.
3. A `BEQ` instruction alters the PC before the branch condition is evaluated.
4. An `OR` instruction writes to `R5`, followed by a `SUB` that writes to `R5`.
5. A `STORE` instruction writes to memory location `0x1000`, followed by a `LOAD` from `0x1000`.

Pipeline Stage Implementation Exercise: Implement the following MIPS instructions in a 5-stage pipeline (IF, ID, EX, MEM, WB). Show which pipeline registers (e.g., `IF/ID`) are updated at each stage.

1. `LW R1, 0(R2)`
2. `ADD R3, R1, R2`

3. `BEQ R1, R2, label`
4. `SW R4, 8(R5)`
5. `XOR R6, R7, R8`

Forwarding Unit Design Exercise: Design forwarding logic for the following sequence of instructions. Specify which operands require forwarding and from which pipeline stage.

1. `ADD R1, R2, R3`
2. `SUB R4, R1, R5` (uses `R1` from `ADD`)
3. `AND R6, R1, R4` (uses `R1` from `ADD` and `R4` from `SUB`)
4. `OR R7, R6, R1` (uses `R6` from `AND` and `R1` from `ADD`)
5. `SLT R8, R7, R6` (uses `R7` from `OR` and `R6` from `AND`)

[Vector addition optimization] Exercise: Consider a processor with separate vector and scalar execution units. The following C code performs element-wise addition of two arrays:

```
void vec_add(float* a, float* b, float* c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
```

1. Rewrite the function using ARM SVE intrinsics with `svcntw()` for vector length
2. Calculate the theoretical speedup assuming 256-bit vectors and 32-bit floats
3. Identify two constraints that prevent achieving this theoretical speedup

[Masked vector operations] Exercise: A common pattern in vector processing is conditional execution:

```
void conditional_add(float* a, float* b, float* c, int n) {
    for (int i = 0; i < n; i++) {
        if (a[i] > 0) {
            c[i] = a[i] + b[i];
        }
    }
}
```

1. Implement this using AVX-512 mask registers with `_mm512_mask_add_ps`
2. Compare the cycle count for masked vs unmasked versions on 64 elements
3. Explain how predication differs from branching in scalar units

[Vector reduction challenge] Exercise: The following code computes the sum of an array:

```
float array_sum(float* arr, int n) {
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

1. Vectorize the reduction using RISC-V V extension with `vfredusum`
2. Calculate the required vector registers for a 1024-element sum

3. Measure the speedup over scalar code for varying array sizes

Thread Affinity Configuration Exercise:

1. Write a C program using `pthread_setaffinity_np` to bind a thread to CPU core `3`.

2. Measure the execution time of a matrix multiplication task with and without thread affinity using `clock_gettime`.

3. Modify the program to alternate thread affinity between cores `0` and `2` every `100` iterations.

Real-Time Scheduling Analysis Exercise:

1. Create a Linux real-time thread using `pthread_create` with `SCHED_FIFO` policy and priority `80`.

2. Write a shell command to display the scheduling policy of all running threads using `ps -eLo pid,tid,cls,pri,cmd`.

3. Identify potential priority inversion scenarios when two threads share a mutex with priorities `75` and `85`.

Hyper-Threading Impact Exercise:

1. Write a benchmark that spawns `8` threads on a `4`-core CPU with hyper-threading.

2. Compare cache miss rates using `perf stat -e cache-misses` for workloads with `50%` and `90%` memory-bound operations.

3. Disable hyper-threading via `/sys/devices/system/cpu/smt/control` and rerun the benchmark.

Clock Domain Crossing Analysis Exercise:

1. Design a dual-flop synchronizer for a 1-bit signal crossing from a 50 MHz clock domain to a 75 MHz domain. Write the Verilog code using `always_ff` blocks.

2. Calculate the mean time between failures (MTBF) for your synchronizer given a metastability resolution time constant of 0.2 ns and a data change rate of 10 MHz.

3. Modify your design to handle a 4-bit bus crossing the same clock domains, ensuring all bits remain aligned.

Handshake Protocol Implementation Exercise:

1. Implement a four-phase handshake protocol between two clock domains (100 MHz and 33 MHz) using `req` and `ack` signals. Provide the state machine diagram and Verilog code.

2. Add a timeout mechanism to your design that resets the handshake if `ack` isn't received within 10 cycles of the faster clock.

3. Simulate your design showing a successful transfer and a timeout scenario using `$display` statements for debugging.

FIFO Synchronization Exercise:

1. Design a gray-code encoded FIFO with separate read/write clocks (80 MHz and 120 MHz). The FIFO should be 8 entries deep and 16 bits wide.

2. Write the Verilog code for the gray-code pointer generation and synchronization logic between domains.

3. Create a testbench that verifies correct operation when the FIFO transitions from empty to full and back to empty.

16.2 Memory System Architecture

Memory Address Decoding Exercise: Exercise: Design a memory controller for a system with the following requirements:

- The system has a 16-bit address bus and an 8-bit data bus.
- It must support two memory chips: a 4 KiB ROM at address range `0x0000-0x0FFF` and an 8 KiB RAM at `0x1000-0x2FFF`.
- Draw the complete address decoding logic using basic gates (AND, OR, NOT).
- Calculate the number of unused address lines for each memory chip.
- Write the Verilog code for the address decoder module.

Bus Timing Analysis Exercise: Exercise: Analyze the timing of a synchronous memory controller with the following parameters:

- Clock frequency: 100 MHz.
- Address setup time: 2 ns.
- Data valid delay: 6 ns after clock edge.
- Memory access time: 12 ns.
- Calculate the minimum clock period for reliable operation.
- Draw the timing diagram showing address, clock, and data signals.
- Modify the design to support burst transfers of 4 words per clock cycle.

DDR3 Protocol Implementation Exercise: Exercise: Implement a simplified DDR3 memory controller with these specifications:

- The controller must handle read and write operations with CAS latency 7.
- Use a finite state machine with these states: IDLE, ACTIVE, READ, WRITE, PRECHARGE.
- Write the VHDL code for the command generation logic.
- Calculate the theoretical bandwidth for a 64-bit bus at 800 MHz.
- Implement the refresh counter logic with 7.8 μs refresh interval.

Write-Invalidate Protocol Analysis Exercise:

1. A multicore system uses the MESI protocol. Core 0 reads `data[0]` from memory, marking the cache line as `Shared`. Core 1 then writes to `data[0]`. List the state transitions and bus signals for both cores.
2. Implement a Python function `mesi_transition(current_state, operation)` that returns the new state given `current_state` (M/E/S/I) and `operation` (read/write).
3. A snooping-based system has 4 cores. Core 0 holds a cache line in `Modified` state. Explain what happens when Core 2 issues a read for the same line, including bus transactions.

Cache Coherency Debugging Exercise:

1. A program produces incorrect results when run on 8 cores but works on 1 core. The shared variable is `counter`. List three possible coherency protocol violations that could cause this.
2. Given the bus trace: `READX from Core1, INVALIDATE from Core3, WRITE from Core2`, reconstruct the cache line states for all cores assuming MOESI protocol.
3. A cache line oscillates between `Invalid` and `Modified` states every 10 cycles. Identify the most likely programming error causing this thrashing.

Directory-Based Protocol Design Exercise:

1. Design a directory entry format for a 64-core system with 16-way set-associative L3 cache. Specify bits needed for state (3 states), presence vector, and LRU tracking.
2. Calculate the directory memory overhead for a 4MB cache with 64-byte lines in a system with 256 cores. Assume 2-bit state and 1-bit presence per core.
3. Write Verilog code for a directory controller that handles `READ` and `WRITE` requests, transitioning between `UNCACHED`, `SHARED`, and `EXCLUSIVE` states.

[Memory Ordering in Producer-Consumer] Exercise:

1. Implement a producer-consumer pattern in C++ using `std::atomic` for synchronization.
2. Add a release fence in the producer thread after writing data and before setting the flag.
3. Add an acquire fence in the consumer thread after checking the flag and before reading data.
4. Verify the correctness using a shared `int buffer[10]` and two threads.

[Atomic Compare-and-Swap] Exercise:

1. Write a C function `atomic_increment_if_less` that increments a `std::atomic*` only if its value is less than a threshold.
2. Use `compare_exchange_weak` with memory order `memory_order_acq_rel`.
3. Test with 4 threads concurrently incrementing a counter up to a maximum value of 1000.
4. Measure and report the number of failed CAS operations.

[Memory Barrier in Kernel] Exercise:

1. Write a Linux kernel module with two shared `unsigned long` variables: `data` and `ready`.
2. Implement a writer thread that uses `smp_wmb()` after writing `data` and before setting `ready`.
3. Implement a reader thread that uses `smp_rmb()` after checking `ready` and before reading `data`.
4. Test with `kthread_create` and verify no race conditions occur.

[Multi-level Page Tables] Exercise:

1. A system uses a 2-level page table with 8-bit virtual page numbers (VPNs) split equally between levels. The page size is 256 bytes. Calculate the total virtual address space size.
2. For the system above, draw the page table structure showing the hierarchy and label the bit ranges for each VPN level.
3. Implement a Python function `translate_virtual_address(vaddr)` that takes a 16-bit virtual address and returns the physical address using a simulated 2-level page table. Assume page tables are stored as dictionaries.

[TLB Performance Analysis] Exercise:

1. A TLB has 32 entries with 5 ns access time. Main memory access takes 80 ns. Calculate the effective memory access time for a 92% TLB hit rate.
2. Modify the TLB to use a 2-way set-associative design with LRU replacement. Show the new TLB structure and tag comparison logic.
3. Write a C code snippet using `struct` to represent a TLB entry containing `tag`, `physical_frame`, and `valid_bit` fields.

[Page Table Conversion] Exercise:

1. Convert the following linear page table entries to physical addresses (page size = 4KB): `PTE[0] = 0x8003`, `PTE[1] = 0x0001`, virtual addresses `0x00001234` and `0x00011000`.
2. Design a hardware circuit using AND/OR gates to extract the offset from a virtual address when page size is 4KB.
3. A process has the following page table: `{0:5, 1:3, 2:7, 3:1}`. Draw the mapping of virtual to physical memory for this process.

[Run-Length Encoding Implementation] Exercise: Implement a run-length encoding (RLE) compression function in Python for binary data.

- Write a function `rle_compress(data: bytes)-> bytes` that encodes consecutive repeated bytes as `[count , value]` pairs.
- Handle edge cases: empty input, single-byte sequences, and maximum run length (255 bytes).
- Include a decompression function `rle_decompress(compressed: bytes)-> bytes`.
- Test with the binary pattern `b'\x01\x01\x01\x00\xFF\xFF'` and verify round-trip correctness.

[LZW Dictionary Management] Exercise: Analyze and implement LZW dictionary handling for ASCII text compression.

- Initialize the dictionary with all 256 single-byte values (0-255).
- Implement dictionary growth up to 4096 entries (12-bit codes) using `dict[bytes] -> int`.
- Write pseudocode for the compression phase that handles dictionary overflow by freezing (stop adding new entries).
- Calculate the compression ratio for the string `"ABABABAB"` versus raw ASCII.

[Huffman Tree Construction] Exercise: Build a Huffman coding tree from character frequencies and compute codewords.

- Given the frequency table `{'A': 15, 'B': 7, 'C': 6, 'D': 6, 'E': 5}`, draw the Huffman tree step-by-step.
- Derive the canonical code lengths for each symbol.
- Implement a Python function `symbol_to_code(tree: Node)-> dict` that generates the bitstring code for each symbol.
- Compute the average code length and compression ratio versus fixed 8-bit encoding.

16.3 Execution Model Implementation

[Warp Divergence Analysis] Exercise:

1. Given the following CUDA kernel code snippet, identify all potential warp divergence points:

```
__global__ void kernel(int* data, int threshold) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (data[idx] > threshold) {
        data[idx] *= 2;
    } else {
        data[idx] /= 2;
    }
    __syncthreads();
}
```

2. Calculate the theoretical performance penalty for a warp when 16 threads take the `if` path and 16 take the `else` path.

3. Modify the kernel to reduce warp divergence using predicated execution techniques.

[Occupancy Calculation] Exercise:

1. Compute the maximum theoretical occupancy for a GPU with:
   - 64K 32-bit registers per SM
   - 48 warps per SM maximum
   - 32 threads per warp
   - Shared memory: 96KB per SM
2. A kernel uses 128 registers per thread and 16KB of shared memory per block. Calculate the actual occupancy for this kernel when using 256-thread blocks.
3. Determine the optimal block size to maximize occupancy for this kernel.

[Wavefront Scheduling] Exercise:

1. Compare and contrast the Greedy-Then-Oldest (GTO) and Round-Robin wavefront scheduling policies.
2. Implement a simplified GTO scheduler in Python that processes a list of wavefronts with given ready times:

```
wavefronts = [
    {'id': 0, 'ready': 0, 'instructions': 10},
    {'id': 1, 'ready': 2, 'instructions': 8},
    {'id': 2, 'ready': 0, 'instructions': 5}
]
```

3. Calculate the total cycles needed to complete all wavefronts using both scheduling policies.

[Pipeline Stall Analysis] Exercise:

1. A 5-stage RISC pipeline (IF, ID, EX, MEM, WB) encounters a RAW hazard when instruction `ADD R1, R2, R3` is followed by `SUB R4, R1, R5`. Calculate the total cycles lost if no forwarding is used.
2. Modify the following VHDL code to implement forwarding from the EX/MEM pipeline register to the ALU inputs:

```
process(clk)
begin
    if rising_edge(clk) then
        EX_MEM_ALUResult <= ALU_Result;
        EX_MEM_RegisterRd <= ID_EX_RegisterRd;
    end if;
end process;
```

3. A processor has a 1-cycle branch penalty. For a loop with 100 iterations and 90% prediction accuracy, compute the total branch penalty cycles.

[Dispatch Slot Constraints] Exercise:

1. Given a 4-wide superscalar processor with these functional units: 2 ALUs, 1 FPU, 1 LSU. Can the following instructions dispatch in one cycle? `ADD R1,R2,R3`, `FMUL F0,F1,F2`, `LD R4,0(R5)`, `XOR R6,R7,R8`.
2. Write a SystemVerilog constraint for a random test generator ensuring no more than 2 memory operations are dispatched per cycle.
3. Calculate the minimum dispatch width needed to sustain 4 IPC (instructions per cycle) with 20% structural hazards.

[Issue Queue Implementation] Exercise:

1. Design a CAM (Content-Addressable Memory) tag structure for an 8-entry issue queue tracking `R1` to `R8` readiness. Show the bits when `R2` and `R5` are ready.

2. Implement in C a circular issue queue with head/tail pointers and this API:

```
bool is_full(struct queue* q);
void enqueue(struct queue* q, inst_t inst);
```

3. An issue queue has 32 entries and 4 read ports. What is the total SRAM bit cells needed if each entry is 128 bits?

[Static vs. Dynamic Prediction] Exercise:

1. Explain the difference between static and dynamic branch prediction techniques.

2. Given the following assembly code snippet, identify all branches and classify them as likely-taken or likely-not-taken for static prediction:

```
loop:
    cmp r1, r2
    bgt exit
    add r3, r3, #1
    sub r2, r2, #1
    b loop
exit:
```

3. Modify the code to use a branch hint (e.g., `likely`/`unlikely` macros) for a processor supporting static prediction hints.

[Branch Target Buffer Analysis] Exercise:

1. Describe the purpose of a Branch Target Buffer (BTB) in modern processors.

2. A BTB has 64 entries with 4-way associativity. Calculate the number of index bits required for addressing.

3. Given the following branch instruction addresses (hexadecimal), determine which would collide in the same BTB set: `0x400A10`, `0x400A14`, `0x400C10`, `0x400E14`.

[Speculative Execution Risks] Exercise:

1. Explain how speculative execution can lead to security vulnerabilities like Spectre.

2. Identify which of these operations should *not* be performed speculatively: (a) cache access, (b) TLB lookup, (c) system call invocation, (d) register renaming.

3. Write a C code snippet that could potentially leak data through a Spectre-style attack using `array1` and `array2`.

[Vector conditional selection] Exercise: Given two vectors `A` and `B` of 2022-03-16
□ 0 □

[Spinlock Implementation] Exercise: Implement a basic spinlock in C using the `atomic_flag` type from .

1. Declare a global `atomic_flag` named `lock_flag` and initialize it to `ATOMIC_FLAG_INIT`.

2. Write a function `void acquire_lock()` that busy-waits using `atomic_flag_test_and_set`.

3. Write a function `void release_lock()` that clears the flag using `atomic_flag_clear`.

4. Demonstrate usage by protecting a critical section that increments a shared counter.

[Semaphore Deadlock Analysis] Exercise: Analyze the following pseudocode for potential deadlocks involving two semaphores:

```
semaphore A = 1;
semaphore B = 1;

thread1() {
    wait(A);
    wait(B);
    // Critical section
    signal(B);
    signal(A);
}

thread2() {
    wait(B);
    wait(A);
    // Critical section
    signal(A);
    signal(B);
}
```

1. Identify the deadlock scenario by listing the specific sequence of operations that causes it.
2. Modify the code to prevent deadlocks while maintaining mutual exclusion.
3. Prove your solution avoids deadlocks using the resource allocation graph method.

[Barrier Synchronization] Exercise: Implement a reusable barrier for N threads using only atomic operations and condition variables.

1. Declare shared variables: `count` (atomic integer), `barrier_generation` (atomic integer), and a condition variable `barrier_cv`.
2. Write a function `void barrier_wait()` that blocks until all N threads have called it.
3. Use `count` to track arrivals and `barrier_generation` to distinguish between consecutive barrier uses.
4. Test your implementation with 4 threads that print messages before and after the barrier.

Chapter 17

Assembly Language Specifics

17.1 Instruction Set Deep Dive

[Fixed-length vs variable-length encoding] Exercise:

1. Given a 16-bit fixed-length instruction format with 4-bit opcode, 3-bit register fields, and 5-bit immediate value, draw the bit layout.
2. Convert the assembly instruction `ADD R1, R2, 0x1F` to binary using your format from step 1.
3. Explain why variable-length encoding might reduce program size compared to fixed-length for an ISA with frequent `MOV` instructions between registers.

[RISC-V immediate encoding] Exercise:

1. Decode the 32-bit RISC-V instruction `0xFE010113` (ADDI x2, x2, -32) by extracting its 12-bit immediate value.
2. Show how the immediate value -32 is sign-extended to 32 bits in this instruction.
3. Write the binary encoding for `ADDI x5, x0, 42` using the RISC-V format.

[x86 ModR/M byte analysis] Exercise:

1. Given the x86 instruction byte sequence `0x8B 0x4D 0xF8`, identify the ModR/M byte and its fields (mod, reg, r/m).
2. Determine the effective address calculation for this instruction (assuming 32-bit mode).
3. Encode a new instruction `MOV EAX, [EBX+ECX*4]` by constructing its ModR/M byte.

Immediate value encoding Exercise: Given the following 32-bit RISC-V instructions, determine the immediate values they encode and their decimal equivalents:

1. `addi x5, x0, 0x1F4`
2. `lw x6, -32(x7)`
3. `jal x1, 0x00001234`

ARM immediate rotation Exercise: For each ARM instruction below, determine if the immediate value is valid and compute its actual value:

1. `mov r0, #0x56000000`
2. `cmp r1, #0xFF0`
3. `add r2, r3, #0x104`

MIPS immediate expansion Exercise: Convert these pseudo-instructions to actual MIPS instructions with proper immediate handling:

1. `li $t0, 0x12345ABCD`
2. `move $t1, $t2`
3. `bgt $t3, $t4, label`

Register flag analysis Exercise:

1. Given a processor with 8 predicate registers `p0` through `p7`, write an assembly sequence that sets `p3` to 1 if the zero flag (ZF) is set and the carry flag (CF) is cleared.
2. Modify the sequence to also set `p5` if the overflow flag (OF) is set, regardless of other flags.
3. Write a C code snippet that uses inline assembly to check `p3` and branches to `label_x` if it is set.

Condition code propagation Exercise:

1. Implement an ARMv8 assembly function that takes two 32-bit integers (passed in `w0` and `w1`) and sets `p2` if their sum would set the negative flag (NF).
2. Extend the function to clear `p4` if the sum produces an unsigned overflow (carry flag set).
3. Write a test case where the input values are `0x80000000` and `0x80000000`, and document the expected predicate register states.

Predicate register debugging Exercise:

1. A program has unexpected behavior when `p1` is set during a loop. Write a debugger command sequence to break when `p1` transitions from 0 to 1.
2. The following assembly snippet incorrectly updates `p6`. Identify the bug:

```
cmp x2, x3
cset p6, gt  // Should set p6 if x2 > x3
and p6, p6, #0x1  // Bug is here
```

3. Rewrite the snippet to properly maintain the predicate register state.

[Bessel Function Evaluation] Exercise: Compute the value of the modified Bessel function of the first kind for order 0 at x=2.5 using its series expansion.

1. Write a C function `double bessel_i0(double x)` that implements the series expansion up to 10 terms.
2. Use the recurrence relation $I_0(x) = \sum_{k=0}^{\infty} \frac{(x^2/4)^k}{(k!)^2}$.
3. Verify your result against the reference value 3.289839 in the range ± 0.0001.
4. Print both your computed value and the absolute error.

[Error Function Implementation] Exercise: Implement the complementary error function erfc(x) using Chebyshev approximations.

1. Create a function `double my_erfc(double x)` using the approximation from Numerical Recipes.
2. Use the coefficients for $0 \leq x \ \infty$ from the book (or standard library documentation).
3. Test your implementation against `erfc` from for x=1.5.
4. Report the maximum relative error in the range [0, 5] using 1000 test points.

[Gamma Function Optimization] Exercise: Optimize a Lanczos approximation implementation of the gamma function.

1. Profile the given `double gamma_lanczos(double x)` function with g=5 and n=6.
2. Replace all `pow` calls with Horner's scheme for polynomial evaluation.

3. Vectorize the coefficient array using SIMD intrinsics (`__m256d` for AVX).
4. Measure the speedup on an x86 processor with AVX support.

Vector element selection Exercise: Given a vector `v = [3, 7, 2, 9, 5, 1]` and a mask `m = [True, False, True, False, True, False]`:

1. Write Python code using NumPy to extract elements from `v` where `m` is `True`
2. Calculate the sum of the selected elements
3. Create a new mask that selects elements greater than 4 from `v`
4. Apply both masks sequentially (first `m`, then the new mask) to `v`

Image thresholding Exercise: A grayscale image is represented as a 2D NumPy array `img` with values 0-255:

1. Create a binary mask for pixels with intensity above 128
2. Apply the mask to set all pixels below 128 to zero
3. Count how many pixels remain above the threshold
4. Write code to display both original and masked images side-by-side using `matplotlib`

Sparse vector operations Exercise: Given two sparse vectors represented as dictionaries `a = {0:1.2, 2:3.4, 5:7.8}` and `b = {1:5.6, 2:1.0, 5:7.8}`:

1. Create boolean masks indicating which indices exist in each vector
2. Compute the element-wise product using vector masks
3. Find the indices where both vectors have non-zero values
4. Implement a masked dot product operation between `a` and `b`

17.2 Register Architecture

Register File Access Timing Exercise:

1. A register file has 32 registers, each 64 bits wide. It has 2 read ports and 1 write port. Calculate the total number of bits required for the complete register file implementation.
2. Given a clock cycle time of 500 ps, and register file read/write delays of 150 ps, calculate the maximum frequency at which this register file can operate without pipeline stalls.
3. Implement a Verilog module for a 3-port register file (2 read, 1 write) with synchronous writes and asynchronous reads. Use `reg [63:0] rf [0:31]` for storage.

Register Bypassing Hazard Exercise:

1. Draw a pipeline diagram showing a RAW hazard when instruction `ADD x1, x2, x3` is followed by `SUB x4 , x1, x5` without bypassing.
2. Modify the following MIPS code to avoid stalls using register renaming:

```
LW   x1, 0(x2)
ADD  x3, x1, x4
SUB  x1, x5, x6
```

3. Calculate the minimum number of bypass paths needed for a 5-stage pipeline with the register file from the first exercise.

Multi-Banked Register File Exercise:

1. Design a 4-bank register file organization for a SIMD processor with 128 registers. Show the mapping of register `v0` to `v127` across banks.

2. Calculate the bank conflict probability when accessing 4 random registers from a 32-register file organized in 4 banks (use modulo-4 mapping).

3. Write ARM assembly code that demonstrates bank conflicts when accessing `s0`, `s4`, `s8` in sequence, then modify it to avoid conflicts.

[Detecting bank conflicts] Exercise: Consider the following CUDA kernel that accesses shared memory:

```
__global__ void bankConflictKernel(float* out, float* in) {
    __shared__ float smem[32];
    int tid = threadIdx.x;
    smem[tid] = in[tid];
    __syncthreads();
    out[tid] = smem[(tid * 2) % 32];
}
```

item Identify which threads will experience shared memory bank conflicts item Calculate the bank conflict ratio (number of conflicting accesses per bank) item Modify the kernel to eliminate all bank conflicts while maintaining the same functionality

[Optimizing for bank conflicts] Exercise: A matrix transpose kernel exhibits 4-way bank conflicts when accessing shared memory:

```
__global__ void transpose(float* out, float* in, int width) {
    __shared__ float tile[32][32];
    int x = blockIdx.x * 32 + threadIdx.x;
    int y = blockIdx.y * 32 + threadIdx.y;
    tile[threadIdx.y][threadIdx.x] = in[y * width + x];
    __syncthreads();
    out[x * width + y] = tile[threadIdx.x][threadIdx.y];
}
```

item Explain why the bank conflicts occur during both read and write operations item Propose two different padding strategies to reduce bank conflicts item Implement the more efficient padding solution and verify the conflict reduction

[Bank conflict analysis] Exercise: Given a shared memory array `s_data[64]` in a warp with 32 threads:

item List all access patterns of form `s_data[(tid + offset)% 64]` that cause no bank conflicts item Calculate the maximum possible bank conflicts for any access pattern using this array item Design an experiment to measure actual bank conflict cycles using CUDA profiler metrics

[Graph coloring] Exercise:

1. Implement a basic interference graph for the following variables used in a basic block:

   ```
   a = 1
   b = a + 2
   c = b * a
   d = c - b
   e = d / a
   ```

2. Draw the interference graph showing which variables cannot share registers

3. Assign registers using graph coloring with a palette of 3 physical registers `r1`, `r2`, `r3`

4. Identify if spilling would be needed if only 2 registers were available

[Linear scan] Exercise:

1. Given the following live ranges for variables:

```
v1: [1, 15]
v2: [3, 8]
v3: [5, 12]
v4: [7, 10]
v5: [9, 14]
```

2. Apply linear scan register allocation with 2 available registers
3. Show the active set at each step and when spills occur
4. Modify the live ranges so no spilling would be needed with 3 registers

[Spill cost analysis] Exercise:

1. Calculate spill costs for variables in this loop (assume each instruction executes 10 times):

```
x = y + z
a = x * 2
b = a + x
z = b / y
```

2. Implement a spill metric that considers both use count and loop depth
3. Determine which variable should be spilled first if only 3 registers are available
4. Modify the code to reduce spill costs by recomputing values instead of storing them

Spill-to-fill ratio calculation Exercise: A fuel depot has a storage tank with a capacity of 5000 liters. During operations:

- The tank receives 1200 liters/hour for 3 hours (fill phase)
- Then dispenses 800 liters/hour for 4 hours (spill phase)
- The cycle repeats continuously

Calculate:

1. The total spill-to-fill ratio for one complete cycle
2. The percentage of time the system spends in fill mode
3. The minimum tank capacity required to prevent overflow if fill rate increases by 20%

Pipeline spill optimization Exercise: A crude oil pipeline uses the following Python function to calculate optimal batch sizes:

```
def calculate_batch(pipe_diameter, flow_rate):
    max_batch = 3.14 * (pipe_diameter/2)**2 * flow_rate * 3600
    return min(max_batch, 50000)  # 50,000 barrel limit
```

Given:

- Pipe diameter: 0.8 meters
- Flow rate: 2.5 m/s
- Crude oil density: 870 kg/m³

Modify the function to:

1. Convert the output to barrels (1 barrel = 0.158987 m³)
2. Add density validation rejecting densities below 800 kg/m³

3. Implement a 10% safety margin on the 50,000 barrel limit

Tank farm scheduling Exercise: A tank farm has 3 tanks with these characteristics:

- Tank A: 10,000 gal capacity, 500 gal/min transfer rate
- Tank B: 15,000 gal capacity, 700 gal/min transfer rate
- Tank C: 8,000 gal capacity, 400 gal/min transfer rate

Create a transfer schedule where:

1. Tank A receives from a pipeline for exactly 15 minutes
2. Then Tank B spills to Tank C until Tank B reaches 40% capacity
3. Finally, Tank C spills to Tank A until Tank C reaches minimum 1000 gal
4. Calculate total operation time assuming instantaneous valve switching

[SIMD register layout] Exercise: Assume a SIMD architecture with 128-bit vector registers that can be partitioned to operate on:

- Sixteen 8-bit integers (`int8_t`)
- Eight 16-bit integers (`int16_t`)
- Four 32-bit integers (`int32_t`)
- Two 64-bit integers (`int64_t`)

Write a C function using intrinsics that:

- Takes two `int32_t` arrays `A` and `B` of length 4
- Loads them into vector registers using `_mm_load_si128`
- Performs element-wise multiplication using `_mm_mullo_epi32`
- Stores the result in array `C`
- Returns the sum of all elements in `C` using horizontal addition

[Vector mask generation] Exercise: Given a vector register containing eight `uint16_t` values:

- Write x86-64 assembly to create a mask where elements > 1000 are set to 0xFFFF
- All other elements should be set to 0x0000
- Use `PCMPGTW` for comparison
- Demonstrate with initial values: [1500, 200, 3000, 40, 5000, 600, 7000, 800]
- Show the final 16-bit mask pattern as a hexadecimal value

[Register banking conflict] Exercise: A GPU has 32 vector registers partitioned into 4 banks:

- Bank 0: Registers `v0-v7`
- Bank 1: Registers `v8-v15`
- Bank 2: Registers `v16-v23`
- Bank 3: Registers `v24-v31`

Identify which of these warp instructions would cause bank conflicts:

- `LDG R0, [v0, v8, v16, v24]`
- `LDG R1, [v1, v9, v17, v25]`
- `LDG R2, [v2, v10, v18, v26]`
- `LDG R3, [v0, v1, v2, v3]`
- Explain your answer for each case in one sentence

17.3 Memory Access Patterns

Memory access patterns and alignment Exercise:

1. Given a CPU with 64-byte cache lines, calculate the worst-case number of cache lines accessed when reading a 256-byte array with:
    - 4-byte aligned starting address
    - 16-byte aligned starting address
    - 64-byte aligned starting address
2. Write a C function `void* aligned_malloc(size_t size, size_t alignment)` that allocates memory aligned to the specified boundary.
3. Measure the performance difference between aligned and unaligned accesses using `__builtin_prefetch` on x86.

Cache line false sharing Exercise:

1. Identify the false sharing problem in this code:

```
struct ThreadData {
    int counter;
    bool flag;
};
```

2. Rewrite the structure to avoid false sharing when accessed by 4 threads.
3. Implement a padding mechanism using C++11's `alignas` specifier.

Vectorized load/store alignment Exercise:

1. Write x86-64 assembly code that loads 32-byte AVX2 vectors with proper alignment checks.
2. Modify this code to handle unaligned accesses using `vmovdqu` instead of `vmovdqa`.
3. Benchmark both versions on an Intel processor using `rdtsc`.

Stride length calculation Exercise: Given a robotic leg with a hip joint at `(x_hip, y_hip)` and foot position at `(x_foot, y_foot)`:

1. Derive the formula for stride length `L` as a function of joint angles `theta_1` and `theta_2`
2. Implement the calculation in Python using `numpy` with the signature `def stride_length(x_hip, y_hip, theta_1, theta_2):`
3. Calculate the stride length for `x_hip=0`, `y_hip=1.2`, `theta_1=0.4`, `theta_2=1.1` radians

Gait efficiency analysis Exercise: For a quadruped robot with the following stride pattern data:

```
stride_patterns = [
    [0.2, 0.3, 0.2, 0.3],  # Pattern A
    [0.4, 0.1, 0.4, 0.1],  # Pattern B
    [0.25, 0.25, 0.25, 0.25]  # Pattern C
]
```

1. Calculate the duty factor for each leg in all three patterns
2. Determine which pattern has the highest stability margin
3. Write a function to compute the total energy expenditure using `energy = sum(0.5 * k * x^2 for x in pattern)`

Stride timing optimization Exercise: Given a legged robot with maximum leg velocity `v_max = 0.8 m/s` and desired stride length `L_desired = 0.6 m`:

1. Calculate the minimum stride period `T_min` that avoids velocity saturation
2. Implement a PID controller in Python that adjusts stride period to maintain `L_desired`
3. Simulate the controller response for a sudden change from `L_desired=0.6` to `0.4` meters

Memory access patterns Exercise:

1. Given a GPU with 32 memory banks and 4-byte words, identify which of these access patterns will cause bank conflicts:

```
thread 0: offset 0
thread 1: offset 8
thread 2: offset 16
thread 3: offset 24
```

2. Rewrite the pattern from (1) to avoid bank conflicts while maintaining the same total memory footprint.
3. Calculate the bank index for thread ID 7 when accessing `shared_mem[thread_id * 5]` with 32 banks.

Shared memory padding Exercise:

1. A kernel uses `__shared__ float data[128][32]`. Determine if this causes bank conflicts when accessed as `data[threadIdx.x][threadIdx.y]`.
2. Add padding to the declaration in (1) to eliminate potential bank conflicts for warp-sized accesses.
3. Write the memory access pattern formula for the padded array from (2) with thread `(x,y)`.

Matrix transpose optimization Exercise:

1. Identify bank conflicts in this naive matrix transpose:

```
__global__ void transpose(float *out, float *in, int width) {
    __shared__ float tile[32][32];
    int x = blockIdx.x * 32 + threadIdx.x;
    int y = blockIdx.y * 32 + threadIdx.y;
    tile[threadIdx.y][threadIdx.x] = in[y*width + x];
    __syncthreads();
    out[x*width + y] = tile[threadIdx.x][threadIdx.y];
}
```

2. Modify the transpose kernel to use a padded shared memory array that avoids bank conflicts.
3. Calculate the optimal padding size for a 48x48 tile transpose on a GPU with 32 banks.

MPI Scatter Implementation Exercise: Implement an MPI program where the root process (rank 0) scatters an array of integers to all other processes.

1. Initialize an array of size N (where N is divisible by the number of processes) with values 1 to N on the root process
2. Use `MPI_Scatter` to distribute equal-sized chunks to all processes
3. Each process should print its received subarray
4. Include proper MPI initialization and finalization
5. Handle the case where N is not divisible by the number of processes by returning an error

Gather Timing Analysis Exercise: Measure and compare the performance of `MPI_Gather` versus manual gathering using point-to-point communication.

1. Create a program where each process generates a random array of 1000 doubles
2. Time the operation using `MPI_Gather` to collect all data on the root

3. Implement manual gathering using `MPI_Send` and `MPI_Recv`
4. Compare execution times for 4, 8, and 16 processes
5. Plot the results showing the time difference between the two approaches

Hybrid Scatter-Gather Application Exercise: Implement a parallel matrix-vector multiplication using scatter/gather operations.

1. Initialize a matrix A (NxN) and vector x (Nx1) on the root process
2. Use `MPI_Scatter` to distribute rows of A to all processes
3. Broadcast vector x to all processes
4. Each process computes its portion of the result vector y = A*x
5. Use `MPI_Gather` to collect the results on the root
6. Verify the result against sequential multiplication
7. Handle edge cases where N is not evenly divisible by the number of processes

Memory barrier implementation Exercise:

1. Write a C function `void atomic_store(int* ptr, int val)` that performs an atomic store with release semantics using GCC built-ins.
2. Implement a corresponding `int atomic_load(int* ptr)` function with acquire semantics.
3. Demonstrate their use by creating two threads where one stores a value and the other loads it, ensuring proper synchronization.

CAS loop optimization Exercise:

1. Implement a thread-safe counter using `__atomic_compare_exchange_n` in a loop.
2. Measure the performance impact of adding `__builtin_expect` to predict the CAS success case.
3. Modify the counter to use exponential backoff when CAS fails, and compare its throughput under contention.

Lock-free queue validation Exercise:

1. Identify the memory ordering requirements for a Michael-Scott queue's `enqueue` operation.
2. Write a stress test that verifies queue correctness under concurrent insertions and removals.
3. Use `ThreadSanitizer` to detect any data races in your implementation.

Chapter 18

AMD GPU Assembly Architecture

18.1 GCN/RDNA ISA Technical Details

RISC-V Immediate Encoding Exercise:

1. Given the RISC-V instruction `addi x5, x6, -42`, determine the 12-bit immediate value in binary.
2. Encode the immediate value from step 1 into the instruction format for I-type instructions.
3. Calculate the hexadecimal representation of the complete 32-bit instruction word.

ARM Thumb-2 Instruction Packing Exercise:

1. Identify which 16-bit halfword contains the opcode in a 32-bit Thumb-2 instruction.
2. Pack the following Thumb-2 instructions into their correct 32-bit format: `MOVS R0, #0x1F` and `LDR R1, [R2, #4]`.
3. Determine if these two instructions can be executed in parallel in a Cortex-M4 processor.

x86 ModR/M Byte Construction Exercise:

1. Construct the ModR/M byte for the x86 instruction `MOV [EBX+ESI*4+0x100], EAX`.
2. Calculate the SIB byte value required for this addressing mode.
3. Determine the total instruction length including displacement bytes.

[Single-cycle ALU latency calculation] Exercise:

1. A scalar ALU has the following propagation delays: 2ns for add/subtract, 3ns for logical operations, and 5ns for shifts.
2. Calculate the minimum clock period required if all operations must complete in one cycle.
3. The ALU now adds a multiplier with 8ns delay. Recalculate the minimum clock period.
4. Explain why this implementation might still use the original clock period despite the multiplier's longer delay.

[Vector ALU throughput analysis] Exercise:

1. A 4-lane vector ALU processes `float32` elements with 1 cycle per operation.
2. Compute the peak throughput in FLOPS at 2GHz clock frequency.
3. The same ALU processes `int16` elements with 8 lanes. Compute the integer OPS rate.
4. A workload has 30% vector utilization due to data dependencies. Calculate the effective throughput.

[ALU control signal decoding] Exercise:

1. Given this ALU control truth table:

```
opcode[1:0] | Operation
00          | ADD
01          | XOR
10          | SRL
11          | SLT
```

2. Write the Verilog code for the control signal decoder.
3. The ALU adds a `MUL` operation using opcode `100`. Modify your decoder.
4. Implement a testbench that verifies all operations including `MUL`.

Data Partitioning Schema Exercise: Design a schema for a local data share system that stores sensor readings from IoT devices.

1. Create a table named `sensor\_readings` with columns: `device\_id` (UUID), `timestamp` (datetime), `value` (float), and `sensor\_type` (varchar).
2. Add a constraint to ensure `value` cannot be negative for `sensor\_type` `"temperature"` or `"pressure"`.
3. Write a SQL query to create a materialized view named `hourly\_averages` that aggregates readings by device and hour.

Access Control Implementation Exercise: Implement role-based access control for the local data share system.

1. Define three PostgreSQL roles: `data\_ingester` (INSERT only), `analyst` (SELECT only), and `admin` (full privileges).
2. Write GRANT statements to give `analyst` read access to `hourly\_averages` but not raw `sensor\_readings`.
3. Create a row-level security policy that restricts `data\_ingester` to only modify rows where `device\_id` matches their session variable `current\_user\_devices`.

Data Synchronization Script Exercise: Write a Python script to synchronize data between nodes in the local data share.

1. Use `psycopg2` to connect to a source and destination PostgreSQL database.
2. Implement a function `get\_pending\_records()` that queries for records with `sync\_status = 'pending'`.
3. Write the synchronization logic using batch processing with 100-record chunks and error logging to `sync\_errors.log`.

18.1.1 Item 4: Wave32/Wave64 execution models

[Wave32/Wave64 execution models]

[Wavefront occupancy calculation] Exercise:

1. Compute the maximum number of wavefronts that can concurrently execute on a GPU with 40 compute units (CUs) if each CU supports 10 wavefronts.
2. A kernel launches with a grid size of 1024 workgroups and a workgroup size of 256 threads. Assuming `wave32` mode, determine the total number of wavefronts generated.
3. If each wavefront requires 32 vector registers and the GPU has 65536 registers per CU, calculate the register-limited wavefront occupancy per CU.

[Wave64 divergence analysis] Exercise:

1. Identify the SIMD execution width for `wave64` mode and explain how it affects branch divergence penalties.

2. Given a `wave64` kernel with a branch condition where 40 threads take the `true` path and 24 take the `false` path, compute the total cycles required assuming 2 cycles per divergent branch.

3. Rewrite the following `wave32` kernel snippet to minimize divergence in `wave64` mode:

```
if (threadIdx.x % 16 < 8) {
    result = a[threadIdx.x] * 2;
} else {
    result = a[threadIdx.x] + 5;
}
```

[Wave32/Wave64 memory coalescing] Exercise:

1. Compare the ideal memory access patterns for `wave32` and `wave64` when loading a `float4` array from global memory.

2. A `wave64` kernel accesses 64 consecutive `int` values starting at address `0x1000`. Determine if this access is fully coalesced for a 128-byte cache line.

3. Modify the following `wave32` memory access to optimize for `wave64`:

```
int idx = threadIdx.x * 4 + (laneID % 4);
int val = data[idx];
```

Task queue management Exercise: Design a hardware task scheduler for a multi-core processor with the following requirements:

- Implement a 16-entry circular task queue using `task_queue_t` struct with fields `task_id`, `priority`, and `core_affinity`.
- Write Verilog code for a round-robin arbiter that selects tasks from the queue.
- Add logic to handle priority inversion when a high-priority task waits for a low-priority task.
- Include a starvation prevention mechanism that boosts priority of aged tasks.
- Verify your design can handle simultaneous enqueue and dequeue operations.

Interrupt latency analysis Exercise: Analyze and optimize interrupt handling in a hardware scheduler:

- Measure worst-case interrupt latency using a logic analyzer on `irq_ack` and `task_switch` signals.
- Implement a shadow register bank to reduce context save/restore time.
- Modify the ISR to use `ldmia` and `stmdb` ARM instructions for register operations.
- Compare the cycle counts for nested vs non-nested interrupts.
- Plot latency distributions for 1000 interrupt events with and without optimizations.

Cache-aware scheduling Exercise: Develop a cache-optimized hardware scheduler:

- Extend the scheduler to track cache lines using `cache_tag_ram`.
- Implement a `migrate_task` function that considers cache locality.
- Design a prefetch trigger based on task scheduling patterns.
- Add performance counters for cache hits/misses per scheduled task.
- Evaluate the impact on IPC (Instructions Per Cycle) for a matrix multiplication benchmark.

18.2 AMD Memory System

[Cache Hierarchy Latency Analysis] Exercise: A system has the following cache specifications:

- L1 cache: 64 KiB, 4-way set-associative, 2-cycle latency
- L2 cache: 512 KiB, 8-way set-associative, 10-cycle latency
- Main memory: 100-cycle latency

Assume hit rates of 95% for L1 and 60% for L2. Calculate:

- The average memory access time (AMAT) for this hierarchy
- The percentage of total accesses that reach main memory
- The speedup if L1 hit rate improves to 97% (keeping L2 hit rate constant)

[Cache Line Replacement Policy] Exercise: Implement the Least Recently Used (LRU) replacement policy for a 4-way set-associative cache in C. Assume:

- Cache line structure: `struct cache_line { int tag; int valid; int timestamp; };`
- Input: `int set_index, int tag` for each access
- Output: Index (0-3) of the line to replace

```
int lru_replacement(struct cache_line set[4], int set_index, int tag) {
    // Your implementation here
}
```

|[Cache Coherency Protocol] Exercise:| A multicore system uses the MESI protocol. For each scenario, identify the state transitions:

- Core A reads a cache line in Shared state from Core B
- Core C writes to a line in Exclusive state
- Core D requests a line in Modified state from Core E
- Core F evicts a line in Invalid state

For each case, specify:

- Initial states of all involved cores
- Final states after the operation
- Bus transactions required (if any)

Memory Timing Analysis Exercise: Exercise:

item Calculate the minimum clock period for a memory controller with the following specs: `t_RCD = 15 ns`, `t_CAS = 12 ns`, `t_RP = 10 ns`, and `t_BURST = 4 clock cycles`. Assume the burst length is 8 words. item Determine if the controller can operate at 100 MHz given these timing constraints. item Modify the `t_CAS` value in the following Verilog code snippet to meet the 100 MHz requirement:

```
parameter t_CAS = 12; // Current value in ns
```

Register Map Implementation Exercise: Exercise:

item Design a register map for a memory controller with 16-bit data width that supports: `refresh_interval` (bits 15-12), `burst_mode` (bit 11), `error_correction` (bit 10), and `bank_address` (bits 9-0). item Write a C function to set the `burst_mode` to 1 and `refresh_interval` to 5 without modifying other bits. item Implement the register write operation in SystemVerilog using the following interface:

```
module mem_ctrl_reg (
    input logic [15:0] reg_data_in,
    input logic reg_write_en
);
```

Command Encoding Exercise: Exercise:

item The memory controller uses 3-bit command codes: `000` for NOP, `001` for READ, `010` for WRITE, and `100` for REFRESH. Encode a sequence of READ-WRITE-REFRESH-NOP. item Calculate the total cycles required for this sequence given each command takes 2 cycles except REFRESH which takes 8 cycles. item Write a Python function that takes a command string like "READ-WRITE-REFRESH" and returns the encoded byte sequence and total cycles.

MSI Protocol State Transitions Exercise: Given the MSI (Modified, Shared, Invalid) cache coherency protocol, complete the following:

1. List all possible states a cache line can be in under the MSI protocol.
2. For each state, describe the allowed transitions and the triggering events (e.g., read miss, write hit).
3. Write a state transition diagram in `graphviz` format showing all states and transitions.
4. Identify which transitions require bus transactions (e.g., BusRd, BusRdX).

False Sharing Detection Exercise: Analyze the following code snippet for potential false sharing:

```
struct Data {
    int thread1_counter;
    int thread2_flag;
    char padding[64];
    int thread2_counter;
    int thread1_flag;
};
```

1. Calculate the byte offset between `thread1_counter` and `thread2_counter`.
2. Assuming a 64-byte cache line size, determine if false sharing could occur.
3. Modify the struct to eliminate all false sharing risks while maintaining the same functionality.
4. Explain how the MESI protocol would handle coherency for this struct.

Cache Coherency Implementation Exercise: Implement a simplified directory-based cache coherency protocol:

```
class DirectoryEntry:
    def __init__(self):
        self.state = "UNCACHED"  # UNCACHED/SHARED/EXCLUSIVE
        self.caches = set()      # IDs of caches with copies
```

1. Add methods `handle_read_request` and `handle_write_request` to update the directory state.
2. Implement transition logic for when a cache requests read or write access.
3. Add invalidation logic when a write request is received and other caches hold copies.
4. Simulate a sequence where Cache A reads, Cache B reads, then Cache A writes to the same line.

Page table walker state machine Exercise:

1. Design a finite state machine for a 2-level page table walker that handles x86-64 4KB pages.
2. Implement the state transitions in Verilog for the states: `IDLE`, `L1_LOOKUP`, `L2_LOOKUP`, `PAGE_FAULT`.

3. Add a timeout mechanism that transitions to `PAGE_FAULT` if any level takes more than 10 cycles.

TLB invalidation synchronization Exercise:

1. Write a C function `invalidate_tlb_entry(uint64_t vaddr)` that safely invalidates a TLB entry while other cores may be accessing it.
2. Add memory barriers to prevent reordering of the invalidation operation.
3. Handle the case where the virtual address spans multiple pages due to huge pages.

Hardware page walk cache Exercise:

1. Design a 4-way associative cache for page table entries with LRU replacement.
2. Calculate the required tag bits for a 48-bit virtual address system with 64-byte cache lines.
3. Implement the cache lookup logic in SystemVerilog, including the hit/miss detection.

[Cache line alignment] Exercise: A system has a 64-byte cache line size. Given the following C struct:

```
typedef struct {
    int32_t id;
    float   temperature;
    char    label[16];
    bool    active;
} sensor_t;
```

1. Calculate the total size of `sensor_t` including padding.
2. Determine how many cache lines are needed to store an array of 10 `sensor_t` elements.
3. Rewrite the struct using `#pragma pack(1)` and recalculate the cache line usage.

[TLB miss analysis] Exercise: A processor has a 4KB page size and 64-entry fully associative TLB. Consider the following memory access pattern:

```
for (int i = 0; i < 1024; i += 64) {
    sum += array[i] * 2;
}
```

1. Calculate the number of TLB misses for this loop if the array starts at address `0x4000`.
2. Modify the access pattern to reduce TLB misses while maintaining the same stride.
3. Calculate the new TLB miss count for your modified version.

[NUMA memory access] Exercise: A NUMA system has two nodes with the following latency characteristics:

- Local memory access: 100 ns
- Remote memory access: 300 ns

1. Calculate the effective latency when 70% of accesses are local.
2. Write a C code snippet using `numa.h` to allocate memory on node 1.
3. Modify the code to migrate a `float` array from node 0 to node 1.

18.3 AMD Performance Optimization

[VGPR Occupancy Tradeoff] Exercise:

1. Compute the maximum number of VGPRs per work-item that can be used while maintaining 100% occupancy on an AMD GPU with 64 compute units and 256 KB of VGPR storage per compute unit.
2. Given a kernel using `__private` memory arrays, explain how reducing VGPR usage could allow for more work-groups to be scheduled concurrently.
3. Write a CUDA or HIP kernel declaration that explicitly limits VGPR usage to 32 registers per work-item using the `__attribute__((amdgpu_num_vgpr))` annotation.

[SGPR Spill Analysis] Exercise:

1. Identify three common kernel operations that typically increase SGPR pressure in AMD GPU architectures.
2. A kernel spills 20 SGPRs to memory. Calculate the potential performance impact assuming each spill requires 10 additional memory instructions and the kernel executes 1 million work-items.
3. Modify the following OpenCL kernel to reduce SGPR usage by replacing the `switch` statement with an alternative control flow:

```
__kernel void sgpr_test(__global int* out) {
    int tid = get_global_id(0);
    switch(tid % 4) {
        case 0: out[tid] = tid * 2; break;
        case 1: out[tid] = tid / 2; break;
        default: out[tid] = tid + 1;
    }
}
```

[Register Allocation Optimization] Exercise:

1. Analyze the following ROCm assembly snippet and count the total VGPR and SGPR usage:

```
v_mov_b32 v0, s0
v_add_u32 v1, v0, v0
s_mov_b32 s1, 0x42
v_cmp_gt_i32 vcc, v1, s1
```

2. Propose two code transformations that could reduce register pressure in a kernel performing successive `float4` vector operations.
3. Implement a matrix multiplication kernel using tile sizes that keep VGPR usage below 64 registers per work-item, assuming each multiply-add operation requires 4 VGPRs.

Instruction scheduling for pipelining Exercise:

1. Given the following MIPS assembly code with data hazards, identify all RAW hazards by listing the instruction pairs involved:

```
lw $t0, 0($s0)
add $t1, $t0, $s1
sub $t2, $t1, $s2
sw $t2, 4($s0)
```

2. Rewrite the code to eliminate hazards using instruction scheduling while preserving correctness.
3. Calculate the total cycles saved in a 5-stage pipeline compared to the original code.

VLIW bundle construction Exercise:

1. Given these three independent operations that can execute in parallel:

```
a = x + y
b = z * w
c = a - b
```

2. Construct a valid VLIW instruction bundle for a machine with 2 ALUs and 1 multiplier.
3. Show the cycle-by-cycle execution timeline for your bundle.
4. Explain why `c = a - b` cannot be in the same bundle as the first two operations.

ARM Thumb-2 encoding Exercise:

1. Convert this ARM instruction to Thumb-2 using the most compact encoding:

```
ADD r4, r5, #255
```

2. Show the binary encoding of your Thumb-2 instruction.
3. Calculate the code size reduction compared to the original ARM encoding.
4. Identify one limitation of Thumb-2 encoding for this instruction.

Direct Memory Access Bypass Exercise:

1. Design a C function using `mmap` and `O_DIRECT` to read a file while bypassing the CPU cache.
2. Measure the execution time difference between cache-bypassing and regular file I/O for a 1GB file using `clock_gettime`.
3. Identify two scenarios where this approach would degrade performance instead of improving it.

Non-Temporal Store Exercise:

1. Implement an x86 assembly routine using `MOVNTPS` to write 128-bit values without polluting the cache.
2. Compare the cache miss rates (using `perf stat`) between non-temporal and regular stores for a 4KB memory block.
3. Modify the routine to handle unaligned memory addresses while maintaining cache bypass.

ARM PMU Configuration Exercise:

1. Write a Linux kernel module that programs ARM Performance Monitoring Unit registers to count L2 cache misses.
2. Capture the PMU counters during a matrix multiplication workload with and without `__builtin_prefetch` hints.
3. Calculate the percentage reduction in L2 misses achieved by prefetching for a 1024x1024 float matrix.

Memory reordering analysis Exercise: Given the following code snippet running on an x86 system with relaxed memory ordering:

```
int x = 0, y = 0;

void thread1() {
    x = 1;
    int r1 = y;
}

void thread2() {
    y = 1;
    int r2 = x;
}
```

1. Identify all possible final values for `r1` and `r2` without memory barriers.
2. Modify the code using `std::atomic` with appropriate memory orders to prevent all but the sequentially consistent outcome.
3. Explain why the x86 architecture might still show reordering despite its strong memory model.

Barrier placement optimization Exercise: Consider this ARMv8 kernel code implementing a shared counter:

```
atomic_t counter = ATOMIC_INIT(0);

void increment() {
    atomic_inc(&counter);
    smp_mb();
}

int read() {
    int val = atomic_read(&counter);
    smp_mb();
    return val;
}
```

1. Analyze whether the memory barriers are optimally placed for correctness and performance.
2. Rewrite both functions to use the minimal necessary barriers while maintaining correctness.
3. Replace the explicit barriers with appropriate `atomic_*` variants that imply memory ordering.

Compiler barrier implementation Exercise: A developer writes this code to prevent compiler reordering:

```
#define COMPILER_BARRIER() asm volatile("" ::: "memory")

void unsafe_write(int* dst, int src) {
    *dst = src;
    COMPILER_BARRIER();
}
```

1. Identify all cases where this barrier fails to ensure memory visibility across threads.
2. Modify the macro to also include a hardware memory barrier for ARM architectures.
3. Implement an equivalent barrier using C11 `atomic_signal_fence` and compare its effects.

Circular shift implementation Exercise: Implement a circular shift function for a 1D array of wave samples in Python. The function should take three parameters: the input array `wave_samples`, shift amount `shift`, and direction `is_left_shift`.

1. Create a function `circular_shift(wave_samples, shift, is_left_shift)`.
2. Handle positive and negative shift values correctly.
3. Ensure the function works for shifts larger than the array length.
4. Return a new array rather than modifying the input.
5. Include edge case handling for empty arrays.

```
# Example usage:
wave = [0.1, 0.5, 0.8, 0.3, -0.2]
shifted = circular_shift(wave, 2, True)  # [0.8, 0.3, -0.2, 0.1, 0.5]
```

Wave phase inversion Exercise: Create a C function that inverts the phase of a stereo wave signal.

1. Write a function `void invert_phase(float* left_channel, float* right_channel, int length).`
2. Multiply each sample in both channels by -1.0.
3. Handle the case where either channel pointer is NULL.
4. Preserve the original values if length is zero or negative.
5. Use pointer arithmetic for array traversal.

```
// Example structure:
typedef struct {
    float* left;
    float* right;
    int samples;
} StereoWave;
```

Permutation validation Exercise: Verify the correctness of a wave permutation function in MATLAB.

1. Load a test wave file using `audioread` into variable `original`.
2. Apply a known permutation `permuted = original(end:-1:1).`
3. Write a validation function `is_valid_permutation(original, permuted).`
4. Check that the permutation preserves all sample values exactly.
5. Return logical true only if the permutation is perfect.
6. Handle different array orientations (row/column vectors).

```
% Test case:
orig = sin(linspace(0,2*pi,1000));
perm = orig(length(orig):-1:1);
assert(is_valid_permutation(orig, perm));
\chapter{NVIDIA GPU Assembly Architecture}
\section{PTX/SASS Technical Implementation}
\textbf{[PTX instruction encoding analysis] Exercise:}
\begin{enumerate}
    \item Given the PTX instruction \lstinline|add.s32 %r1, %r2, %r3;|, identify the operation
        , destination register, and source registers.
    \item Convert the following PTX instruction to binary encoding using the format: opcode (6
         bits), destination (8 bits), source1 (8 bits), source2 (8 bits). Assume \lstinline|
        add.s32| has opcode \lstinline|0x18| and registers are numbered sequentially.
    \item List three differences between PTX register addressing modes and x86 register
        addressing modes.
\end{enumerate}

\textbf{[PTX instruction modification] Exercise:}
\begin{enumerate}
    \item Rewrite the PTX instruction \lstinline|ld.global.f32 %f1, [%r4];| to use 64-bit
        addressing with register \lstinline|%r5| as an offset.
    \item Identify and fix the error in this PTX instruction: \lstinline|mul.f64 %d1, %r2, %r3
        ;|
    \item Write a PTX instruction sequence to perform: \lstinline|%r1 = (%r2 + %r3) * 4| using
         only one arithmetic instruction.
\end{enumerate}

\textbf{[PTX instruction optimization] Exercise:}
\begin{enumerate}
```

```
    \item Replace this instruction sequence with a single PTX instruction: \lstinline|mov.s32
        %r1, %r2; add.s32 %r1, %r1, 1;|
    \item Convert the following PTX code to use predicate registers instead of branches:
\begin{lstlisting}
    @p bra L1;
    add.s32 %r1, %r2, %r3;
L1:
```

Analyze the PTX instruction `mad.lo.s32 %r1, %r2, %r3, %r4;` and calculate how many clock cycles it would take on an architecture with 32-bit multipliers having 4-cycle latency.

Mixin Efficiency Exercise: Exercise:

1. Create a SASS mixin named `border-radius` that accepts one parameter for radius value
2. The mixin should output vendor-prefixed versions (`-webkit-`, `-moz-`) for maximum compatibility
3. Use the mixin to apply a 5px radius to a `.button` class and 10px to `.card`
4. Convert the mixin to use default parameter of 3px when no value is passed

Selector Nesting Exercise: Exercise:

1. Given HTML with `<nav class="main-nav"><ul><li><a>` structure, write optimized SASS nesting
2. Avoid selector nesting deeper than 3 levels
3. Use `&` parent selector for hover states on anchor tags
4. Extract repeated color values into a `$nav-colors` map variable

Partial Import Exercise: Exercise:

1. Create a SASS project structure with these partials: `_variables`, `_mixins`, `_buttons`
2. Set up a main `styles.scss` that imports all partials in correct order
3. Implement `@use` instead of `@import` with proper namespace references
4. Add a forward rule in `_utils.scss` that bundles `_variables` and `_mixins`

[Branchless Conditional Assignment] Exercise: Implement a branchless version of the following C function using predication:

```
int conditional_abs(int x) {
    if (x < 0) return -x;
    else return x;
}
```

1. Declare a mask variable using `int mask = x >> 31`
2. Compute the absolute value using `(x + mask)^ mask`
3. Write the complete function and verify it with test cases `x = -5`, `x = 0`, and `x = 42`

[SIMD Predication Mask] Exercise: Create an AVX2 implementation of element-wise conditional selection:

```
void select_if(float* dst, float* a, float* b, int* cond, size_t n);
```

1. Load 8 elements from `a`, `b`, and `cond` into `__m256` registers
2. Convert the condition mask using `_mm256_castsi256_ps(_mm256_cmpgt_epi32(...))`
3. Use `_mm256_blendv_ps` to select elements based on the mask
4. Store the result in `dst` with proper alignment handling

[Predicated Instruction Scheduling] Exercise: Analyze the following ARM assembly code with predicated execution:

```
cmp     r0, #0
movgt   r1, #1
movle   r1, #0
add     r2, r2, r1
```

1. Identify all predicated instructions in the sequence
2. Calculate the CPI assuming 50% taken probability for the branch
3. Rewrite the code using conditional select `sel r1, r0, #1, #0`
4. Compare the cycle counts for both versions

Repository Branch Merge Exercise: Exercise:

1. Clone the repository at `https://github.com/example/repo.git` to your local machine.
2. Create a new branch named `feature_x` from the `main` branch.
3. Make three commits to `feature_x`, each modifying a separate file.
4. Fetch the latest changes from the remote `main` branch.
5. Rebase `feature_x` onto the updated `main` branch.
6. Resolve any merge conflicts that arise during the rebase.
7. Push the rebased `feature_x` branch to the remote repository.

Conflict Resolution Simulation Exercise: Exercise:

1. Initialize a new Git repository with `git init`.
2. Create a file `data.txt` with the content `A=1` and commit it.
3. Create two branches: `branch1` and `branch2`.
4. In `branch1`, modify `data.txt` to `A=1,B=2` and commit.
5. In `branch2`, modify `data.txt` to `A=1,C=3` and commit.
6. Attempt to merge `branch2` into `branch1`.
7. Manually resolve the merge conflict by combining both changes to `A=1,B=2,C=3`.
8. Create a merge commit with the resolved file.

Remote Tracking Branch Exercise: Exercise:

1. Fork the repository `https://github.com/example/upstream.git` to your GitHub account.
2. Clone your fork locally and add the upstream repository as a remote.
3. Create a tracking branch for `upstream/main` in your local repository.
4. Fetch all branches from the upstream remote.
5. Create a local branch `patch_y` that tracks `upstream/dev`.
6. Make a commit on `patch_y` and push it to your fork.
7. Create a pull request from your fork's `patch_y` to `upstream/dev`.

[Basic warp shuffle implementation] Exercise: Implement a CUDA kernel that uses warp shuffle operations to compute the sum of 32 floating-point values across a warp. Follow these steps:

- Declare a shared memory array of 32 floats in the kernel.
- Each thread should store its thread ID (converted to float) in the shared array.
- Use `__shfl_down_sync` with a mask of `0xffffffff` to perform the reduction.
- Only thread 0 should store the final result in global memory.
- Write the complete kernel code including all necessary synchronization.

[Warp shuffle for matrix transpose] Exercise: Create a CUDA kernel that transposes a 32x32 matrix using warp shuffle operations. Follow these steps:

- Assume the input matrix is stored in row-major order in global memory.
- Each thread should load one element from global memory.
- Use `__shfl_sync` to exchange data between threads within the warp.
- Store the transposed elements back to global memory in column-major order.
- Handle all memory operations with proper coalescing and alignment.
- Include boundary checks in case the matrix dimensions aren't perfect multiples of 32.

[Error checking in warp operations] Exercise: Write a CUDA function that validates proper usage of warp shuffle operations. Complete these tasks:

- Create a test case where threads in a warp are diverged (some threads execute different code paths).
- Use `__activemask` to detect active threads before shuffle operations.
- Implement proper error handling when the active mask doesn't match the shuffle mask.
- Return an error code if any thread detects improper shuffle usage.
- Include a test kernel that demonstrates both valid and invalid cases.

18.4 NVIDIA Memory Architecture

Memory Bank Conflict Analysis Exercise:

1. A shared memory system has 8 memory banks with interleaving at the word level (4 bytes per word). The address mapping uses bits [3:5] for bank selection. Calculate the bank index for addresses `0x1A74` and `0x2BC8`.
2. For a 4-bank system with 32-bit words and byte addressing, design the address partitioning scheme showing which bits select the bank and which select the word within the bank.
3. Explain how false sharing would occur in this system if two threads access different elements of `int array[8]` where each thread updates `array[0]` and `array[4]` respectively.

Bank Parallelism Calculation Exercise:

1. Given a 16-bank memory system with 64-byte cache lines, compute the maximum achievable bandwidth when reading a 256-byte contiguous block versus a 256-byte strided block with stride 512 bytes.
2. A GPU has 32 memory banks with 4-byte granularity. Determine if the following access pattern will cause bank conflicts: `thread0: addr0, thread1: addr32, thread2: addr64, thread3: addr96`.
3. Modify the address sequence `0, 16, 32, 48, 64, 80` to avoid bank conflicts in a 16-bank system with 16-byte words by applying an offset.

Memory Bank Implementation Exercise:

1. Write Verilog code for a 4-way bank selector that routes 32-bit addresses to one of four banks using bits [4:5] for bank selection.
2. Design a C++ struct with padding to ensure elements `a` and `b` reside in different memory banks in a 4-bank system with 64-byte cache lines.
3. Implement a CUDA kernel that performs a parallel reduction across 128 elements while minimizing shared memory bank conflicts.

[Cache Line Replacement] Exercise: Implement a least-recently-used (LRU) replacement policy for a 4-way set-associative L1 cache with 64-byte lines.

1. Declare a C struct `cache_set_t` containing 4 tags, 4 valid bits, and 4 LRU counters (8-bit unsigned integers).
2. Write a function `update_lru(cache_set_t* set, int way)` that updates the LRU counters when accessing way `way`.
3. Implement `find_lru_victim(cache_set_t* set)` that returns the way index (0-3) with the highest LRU counter value.
4. Handle the special case where invalid lines (valid bit=0) should be preferred over LRU replacement.

[Cache Coherence Protocol] Exercise: Implement the MESI protocol transitions for a single cache line in a multicore system.

1. Define an enum `mesi_state_t` with states Modified, Exclusive, Shared, and Invalid.
2. Create a struct `cache_line_t` with state field, tag, and 64-byte data array.
3. Write a function `handle_read(cache_line_t* line)` that implements state transitions for local read requests.
4. Implement `handle_bus_read(cache_line_t* line)` for bus read transactions from other cores.
5. Add error checking that forces transition to Invalid state upon any protocol violation.

[Cache Performance Analysis] Exercise: Measure and analyze cache performance using hardware performance counters.

1. Write a C program that allocates a 4MB array and performs sequential reads with stride=64.
2. Use `perf_event_open()` to measure L1-dcache-load-misses during the access pattern.
3. Repeat measurements for stride=128, 256, 512 while keeping total access count constant.
4. Calculate the miss rate for each stride and plot the results using gnuplot.
5. Identify the stride value where L1 cache capacity is exceeded based on your CPU's specifications.

[Coalesced access patterns] Exercise:

1. Identify which of the following CUDA kernel memory access patterns would achieve perfect coalescing for 32-bit words on compute capability 7.0:

```
// Pattern A
__global__ void kernelA(float* out, float* in) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    out[tid] = in[tid * 2];
}

// Pattern B
__global__ void kernelB(float* out, float* in) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    out[tid] = in[tid + blockDim.x * threadIdx.y];
}
```

2. Modify Pattern A to achieve coalesced access while maintaining the same stride-2 semantic.
3. Calculate the memory transaction efficiency for Pattern B when `blockDim.x`=32 and `blockDim.y`=4.

[Padding for alignment] Exercise:

1. Given a structure `struct {float a; int b; char c;}`, determine if padding is needed between members for optimal coalesced access.
2. Write a CUDA kernel that processes an array of the above structures with coalesced access, assuming 128 threads per block.
3. Measure the performance difference between aligned and unaligned access using CUDA events for a 1MB array of these structures.

[Bank conflict analysis] Exercise:

1. Analyze the shared memory bank conflicts in the following access pattern for 32 banks:

```
__shared__ float data[256];
float value = data[threadIdx.x * 4 + threadIdx.y];
```

2. Rewrite the access pattern to eliminate bank conflicts while maintaining the same logical addressing scheme.
3. Implement both versions and compare their execution times using `nvprof` metrics for a 1024-element array.

Memory Ordering in C++ Atomics Exercise: Exercise:

1. Declare an atomic integer `shared_counter` initialized to 0.
2. Write a function `increment_counter` that increments `shared_counter` using `memory_order_relaxed`.
3. Write a second function `load_counter` that loads `shared_counter` using `memory_order_acquire`.
4. Explain why these memory orders might be insufficient for synchronizing two threads.

SC for Delay Slots Exercise: Exercise:

1. Given a MIPS pipeline with delay slots, draw the execution timeline for two instructions: `LW R1, 0(R2)` followed by `ADD R3, R1, R4`.
2. Mark all pipeline stages where the `ADD` depends on `LW`.
3. Annotate where a memory barrier would be needed if this architecture didn't guarantee SC.
4. Calculate the total cycles lost without SC guarantees.

TSO Store Buffer Simulation Exercise: Exercise:

1. Simulate a TSO architecture with two cores and store buffers using this code:

```
// Core 1
mov [x], 1  // x initially 0
mov r1, [y] // y initially 0

// Core 2
mov [y], 1
mov r2, [x]
```

2. List all possible final values of `r1` and `r2` under TSO.
3. Identify which outcomes would be impossible under sequential consistency.
4. Modify the code to enforce SC using memory fences.

[Compare-and-swap implementation] Exercise: Implement a thread-safe counter using the compare-and-swap (CAS) atomic operation.

1. Declare an integer counter variable `shared_counter` and initialize it to 0
2. Write a function `increment_counter()` that uses CAS to atomically increment `shared_counter`
3. The function should return the new counter value after successful increment
4. Handle the case where CAS fails by implementing a retry loop
5. Test your implementation with 4 concurrent threads each incrementing 1000 times

[Atomic flag synchronization] Exercise: Implement a simple mutex using atomic flags for thread synchronization.

1. Declare an atomic boolean flag `lock_flag` initialized to false
2. Implement `acquire_lock()` that sets `lock_flag` to true only if it was false
3. Implement `release_lock()` that resets `lock_flag` to false
4. Add proper memory ordering constraints using `memory_order_acquire` and `memory_order_release`
5. Demonstrate usage by protecting a critical section that increments a shared counter

[Lock-free stack push] Exercise: Implement the push operation for a lock-free stack using atomic operations.

1. Define a node structure with `data` and `next` pointer members
2. Declare an atomic stack head pointer `stack_top` initialized to nullptr
3. Implement `push(int value)` that creates a new node and atomically updates `stack_top`
4. Use CAS to handle concurrent pushes from multiple threads
5. Ensure proper memory ordering using `memory_order_release` for the atomic update
6. Test with 3 threads pushing 500 values each and verify all values are preserved

18.5 NVIDIA Performance Engineering

[Pipeline Hazard Analysis] Exercise:

1. Given the following MIPS assembly code, identify all register dependency chains longer than 2 instructions:

```
loop:   lw   $t0, 0($a0)
        addi $t1, $t0, 5
        sw   $t1, 0($a1)
        addi $a0, $a0, 4
        addi $a1, $a1, 4
        sub  $t2, $t1, $a0
        bnez $t2, loop
```

2. For each dependency chain found, calculate the minimum number of pipeline stalls required if the processor has no forwarding unit.
3. Rewrite the code to reduce stalls by reordering instructions while preserving correctness.

[Out-of-Order Execution] Exercise:

1. A processor has 4 execution units: 2 ALUs, 1 load/store unit, and 1 branch unit. Given the following instruction sequence, build a dependency graph:

```
ldr r0, [r1]
add r2, r0, r3
mul r4, r2, r5
str r4, [r6]
cmp r2, #10
bgt label
```

2. Schedule these instructions for earliest possible execution, respecting dependencies and resource constraints.
3. Calculate the total cycles required assuming: load/store takes 3 cycles, ALU ops take 1 cycle, branch takes 2 cycles.

[Register Renaming] Exercise:

1. Trace the following x86-64 code through a register renaming table with 8 physical registers (p0-p7):

```
mov rax, [rdi]
add rbx, rax
mov rcx, rbx
shl rcx, 2
mov [rsi], rcx
```

2. Show the mapping of architectural registers (rax, rbx, rcx) to physical registers after each instruction.
3. Identify any false dependencies that would exist without renaming.

[Pipeline stall analysis] Exercise: Consider a 5-stage RISC pipeline (IF, ID, EX, MEM, WB) with the following characteristics:

- Branches are resolved in the ID stage (no branch prediction)
- Memory operations have 3-cycle latency
- ALU operations have 1-cycle latency
- The processor has no forwarding or hazard detection

Given the following assembly code:

```
loop:  lw   t0, 0(a0)
       addi t1, t0, 5
       sw   t1, 0(a1)
       addi a0, a0, 4
       addi a1, a1, 4
       bne  a0, a2, loop
```

- Calculate the total number of cycles needed to execute 10 loop iterations
- Identify all pipeline stalls in the first iteration
- Propose two hardware modifications to reduce stalls

[Software prefetching] Exercise: A matrix multiplication kernel processes 64x64 floating-point matrices using this inner loop:

```
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 64; j++) {
        for (int k = 0; k < 64; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

- Identify which array accesses cause cache misses due to striding
- Rewrite the loop using `__builtin_prefetch` to hide memory latency
- Calculate the minimum prefetch distance needed for a 100-cycle memory latency

[Out-of-order execution] Exercise: An out-of-order processor has these characteristics:

- 4-wide issue with 128-entry reorder buffer
- 64 physical registers
- 4-cycle minimum branch misprediction penalty

Given the following code sequence:

```
ldr   x0, [x1]
add   x2, x0, x3
mul   x4, x2, x5
str   x4, [x6]
cmp   x7, #0
b.eq  target
```

- Identify all true data dependencies in the code
- Determine the maximum possible IPC for this sequence
- Calculate how many cycles the sequence would take with perfect branch prediction

Coalescing Address Analysis Exercise:

1. Given a memory access pattern with the following addresses (in hex): `0x1000`, `0x1004`, `0x1008`, `0x1010`, determine which accesses can be coalesced into a single 16-byte transaction.
2. Calculate the total bandwidth savings (in bytes) if coalescing is applied to the pattern from task 1, assuming each uncoalesced access is 4 bytes.
3. Implement a Python function `can_coalesce(addr1, addr2, burst_size)` that returns `True` if two addresses can be coalesced within the given burst size (in bytes).

DRAM Burst Transaction Exercise:

1. A DRAM controller receives the following sequence of requests: `0x2000_0000`, `0x2000_0020`, `0x2000_0010`, `0x2000_0008`. The burst length is 32 bytes. Identify which requests belong to the same burst.
2. Compute the effective bandwidth utilization (percentage) for the sequence in task 1, assuming each request is 8 bytes and burst transfers take 5 cycles.
3. Modify the following Verilog code to implement a simple coalescing buffer that combines 4-byte writes into 16-byte bursts:

```
module coalescing_buffer (
    input [31:0] wr_addr,
    input [31:0] wr_data,
    input wr_en
);
```

Cache Line Optimization Exercise:

1. A GPU warp issues memory requests to `0xA040`, `0xA044`, `0xA048`, and `0xA04C`. The cache line size is 32 bytes. Determine if all accesses hit the same cache line.
2. Design a C++ function `coalesce_requests(vector& addresses)` that groups addresses into cache-line-aligned clusters.
3. Calculate the memory throughput improvement (in GB/s) when coalescing 32 random 4-byte accesses into 4 cache-line transactions, assuming a 64-byte cache line and 100 ns access latency.

[Warp divergence analysis] Exercise: A GPU kernel processes an array of 1024 elements using 32-thread warps. The kernel contains the following branching condition:

```
if (threadIdx.x % 16 < 8) {
    // Path A
} else {
    // Path B
}
```

1. Calculate the number of warps that will experience full divergence.
2. Determine the percentage of threads that will follow Path A in each divergent warp.
3. Modify the condition to reduce warp divergence while maintaining the same logical behavior.

[Occupancy calculation] Exercise: Consider a GPU with the following specifications:

- 64 KB shared memory per SM
- 2048 maximum threads per SM
- 32 registers per thread
- 64 KB register file per SM

1. Calculate the maximum theoretical occupancy for a kernel using `__launch_bounds__(256, 4)`.
2. Determine the limiting factor (registers, shared memory, or threads) for this kernel.
3. Propose two modifications to increase occupancy without changing the kernel's functionality.

[Warp scheduling efficiency] Exercise: A CUDA kernel exhibits the following characteristics:

- 50% of warps stall for memory operations
- 30% of warps stall for synchronization
- 20% of warps are ready to execute

1. Calculate the theoretical maximum IPC (instructions per cycle) if the GPU has 4 warp schedulers.
2. Identify which type of stall (memory or synchronization) has greater impact on performance.
3. Write a `__global__` kernel that uses shared memory to reduce memory stalls.

[Matrix multiplication precision analysis] Exercise:

1. Given two FP16 matrices `A[8,8]` and `B[8,8]` being multiplied using tensor cores, calculate the theoretical number of floating-point operations required.
2. Implement a CUDA kernel using `wmma::mma_sync()` to perform this multiplication, storing results in FP32 matrix `C[8,8]`.
3. Compare the numerical accuracy against a naive FP32 CPU implementation when multiplying matrices with values in range [0.1, 1.0].

[Tensor core memory layout] Exercise:

1. Explain why matrix `B` must be stored in column-major format for optimal tensor core performance.
2. Write CUDA code to transpose a row-major FP16 matrix `B_rowmajor[16,16]` into column-major format `B_colmajor[16,16]`.
3. Measure the performance difference between using pre-transposed versus on-the-fly transposed matrices in a tensor core operation.

[Mixed-precision tensor operations] Exercise:

1. Design a tensor core operation that multiplies FP16 matrix `A[16,8]` by INT8 matrix `B[8,16]` with FP32 accumulation.
2. Implement proper scaling factors for the INT8 matrix to prevent overflow during multiplication.
3. Validate your implementation by comparing against a reference CPU implementation using double precision.

Chapter 19

Cross-Vendor Techniques

19.1 Comparative Analysis

Compute Unit vs SM Analysis Exercise: Exercise:

1. Compare the architectural differences between AMD's Compute Unit (CU) and NVIDIA's Streaming Multiprocessor (SM).
2. List three key hardware components found in both CUs and SMs.
3. Explain how `wavefronts` (AMD) differ from `warps` (NVIDIA) in terms of scheduling.
4. Write a CUDA kernel that would require 32 threads per warp, then describe how you'd modify it for AMD's 64-thread wavefronts.

Memory Hierarchy Benchmarking Exercise: Exercise:

1. Design an experiment to measure the bandwidth differences between NVIDIA's `L2_cache` and AMD's `Infinity_Cache`.
2. Write a `OpenCL` kernel that accesses memory with different stride patterns to test cache performance.
3. Compare the results when running on an NVIDIA GPU with unified L2 cache versus an AMD GPU with Infinity Cache.
4. Identify which architecture benefits more from `coalesced_memory_accesses` based on your results.

Instruction Set Comparison Exercise: Exercise:

1. Contrast AMD's `GCN_ISA` with NVIDIA's `PTX` in terms of instruction-level parallelism.
2. Write two matrix multiplication implementations: one using `VOP3` instructions (AMD) and one using `FMA` instructions (NVIDIA).
3. Measure the IPC (Instructions Per Cycle) difference between the two implementations on respective hardware.
4. Explain how each architecture's SIMD width affects your results.

[Instruction Encoding] Exercise:

1. Given a 32-bit RISC-V instruction `0x00C58533`, decode it into its assembly mnemonic and register operands.
2. Convert the MIPS instruction `add $t0, $s1, $s2` to its 32-bit binary encoding. Assume opcode=0, funct=32.
3. Identify the addressing mode used in the x86 instruction `mov eax, [ebx+4*esi+16]`.

[Pipeline Hazards] Exercise:

1. For the ARM sequence `ADD R1,R2,R3` followed by `SUB R4,R1,R5`, identify the hazard type and propose a solution.
2. Calculate the stall cycles needed for the MIPS pipeline when `lw $t0, 0($s0)` is followed by `add $t1, $t0, $t2`.
3. Rewrite the x86 code `mov eax, [ecx]` and `add ebx, eax` to avoid data hazards using register renaming.

[SIMD Implementation] Exercise:

1. Write AVX2 intrinsics to multiply eight single-precision floats in `ymm0` by those in `ymm1`.
2. Convert the scalar ARM NEON operation `vadd.f32 s0, s1, s2` to a vectorized version processing four floats.
3. Identify the minimum vector width needed to compute 16 parallel `int16_t` multiplications on RISC-V.

[Thread vs. Process Overhead] Exercise:

1. Measure the time to create 1000 threads using `pthread_create()` in C, recording the total elapsed time with `clock_gettime()`.
2. Repeat the measurement for 1000 processes using `fork()`.
3. Calculate the per-thread and per-process creation overhead in microseconds.
4. Explain which method scales better for fine-grained parallelism based on your measurements.

[GPU Memory Latency] Exercise:

1. Write a CUDA kernel that measures global memory latency by accessing a 1MB array with `stride` values from 1 to 1024.
2. Plot the access time versus stride using `nvprof` metrics.
3. Identify the stride value where latency increases significantly due to cache line boundaries.
4. Modify the kernel to use shared memory and compare the latency profile.

[Message Passing Bottlenecks] Exercise:

1. Implement a ring communication pattern in MPI where each rank sends a 1KB message to its right neighbor.
2. Measure the total communication time for 1000 iterations with `MPI_Wtime()`.
3. Repeat the experiment with non-blocking `MPI_Isend`/`MPI_Irecv` and overlapping computation.
4. Calculate the bandwidth improvement percentage from overlapping.

19.2 Portable Assembly Code

Buffer Copy Performance Analysis Exercise: Exercise:

1. Write an OpenCL host program that creates two buffers: `input_buffer` (read-only) and `output_buffer` (write-only), each 16MB in size.
2. Profile the copy operation from `input_buffer` to `output_buffer` using `clEnqueueCopyBuffer` with and without the `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` flag.
3. Compare the execution times for both cases and explain the observed performance difference in a comment.

Vulkan Pipeline Creation Exercise: Exercise:

1. Create a Vulkan compute pipeline using SPIR-V bytecode from the following GLSL compute shader:

```
#version 450
layout(local_size_x = 32) in;
layout(binding = 0) buffer Data { float values[]; };
void main() {
    uint idx = gl_GlobalInvocationID.x;
    values[idx] = values[idx] * 2.0f;
}
```

2. Include all necessary Vulkan structures: `VkPipelineLayout`, `VkShaderModule`, and `VkComputePipelineCreateInfo`.
3. Verify pipeline creation by checking the return value of `vkCreateComputePipelines`.

SPIR-V Cross-Compilation Exercise: Exercise:

1. Compile the following OpenCL kernel to SPIR-V using the OpenCL to SPIR-V compiler:

```
__kernel void vec_add(__global float* a,
                      __global float* b,
                      __global float* c) {
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

2. Inspect the generated SPIR-V using `spirv-dis` and identify the OpLoad and OpStore instructions corresponding to the memory operations.
3. Modify the kernel to use `__constant` instead of `__global` for input buffer `a` and regenerate the SPIR-V.

[Memory Coalescing Adaptation] Exercise:

1. Rewrite the following AMD GPU memory access pattern for NVIDIA GPUs, ensuring proper coalescing. The original code uses `__attribute__((amd_flat_work_group_size(64, 256)))`.
2. Replace AMD's `__lds` with the equivalent NVIDIA CUDA shared memory declaration.
3. Benchmark both versions using `nvprof` and compare the achieved memory bandwidth.

[Wavefront to Warp Conversion] Exercise:

1. Convert this AMD wavefront (64 threads) reduction kernel to NVIDIA warp (32 threads) operations:

```
__kernel void reduce(__global float* input, __global float* output) {
    __local float lds[64];
    int lid = get_local_id(0);
    lds[lid] = input[get_global_id(0)];
    for (int i = 32; i > 0; i >>= 1) {
        if (lid < i) lds[lid] += lds[lid + i];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if (lid == 0) output[get_group_id(0)] = lds[0];
}
```

2. Handle the case where the original workgroup size (256) isn't a multiple of NVIDIA's warp size.
3. Verify correctness by comparing outputs from both implementations.

[ROCm to CUDA Portability] Exercise:

1. Port this ROCm HIP atomic operation to CUDA:

```
__device__ float atomicAdd_float(float* address, float val) {
    union { int i; float f; } old, assumed;
    old.f = *address;
    do {
        assumed.f = old.f;
        old.i = atomicCAS((int*)address, assumed.i, __float_as_int(assumed.f + val));
    } while (assumed.i != old.i);
    return old.f;
}
```

2. Replace AMD-specific `__float_as_int` with CUDA equivalents.
3. Test both versions using a histogram kernel with floating-point bin updates.

Memory Alignment Optimization Exercise:

1. Write a C function `void* aligned_malloc(size_t size, size_t alignment)` that allocates memory with the specified alignment.
2. Implement a corresponding `void aligned_free(void* ptr)` function to safely deallocate the memory.
3. Test your implementation by allocating three 256-byte buffers with alignments of 16, 32, and 64 bytes respectively.
4. Verify the alignment of each pointer using bitwise operations and print the results.

SIMD Vectorization Exercise:

1. Create an x86-64 assembly function using AVX2 instructions to compute the dot product of two arrays of 32-bit floats.
2. The function signature should be `float dot_product_avx2(const float* a, const float* b, size_t count)`.
3. Handle array lengths that aren't multiples of 8 by using a scalar cleanup loop.
4. Benchmark your implementation against a naive C version for arrays of size 1000 elements.

GPU Memory Coalescing Exercise:

1. Write a CUDA kernel to transpose a 32x32 matrix stored in global memory.
2. Implement two versions: one with coalesced reads and scattered writes, and another with scattered reads and coalesced writes.
3. Use `cudaEvent_t` to measure execution time of both versions.
4. Analyze the performance difference using the NVIDIA Visual Profiler.

19.3 Cross-Vendor Debugging and Profiling

[RenderDoc Frame Capture] Exercise:

1. Launch RenderDoc and attach it to a running instance of `my_game.exe` on Windows.
2. Trigger a frame capture during a scene with heavy GPU usage (e.g., particle effects).
3. Locate the draw call with the highest pixel overdraw in the Pipeline State view.
4. Export the captured frame as a `.rdc` file and share it via a cloud storage link.

[GDB Memory Breakpoint] Exercise:

1. Compile `debug_program.cpp` with `-g -O0` flags and start GDB.

2. Set a hardware watchpoint on the global variable `g_player_health`.
3. Reproduce a crash and identify the function modifying `g_player_health` incorrectly.
4. Use `info registers` to dump the CPU state at the crash point.

[Cross-Platform Shader Debugging] Exercise:

1. Capture a Vulkan frame in RenderDoc on Linux with `vk_layer_settings.txt` enabling validation.
2. In the Mesh Viewer, locate a corrupted vertex attribute in the `POSITION` buffer.
3. Modify the vertex shader in RenderDoc's Shader Editor to add `debugPrintfEXT` output.
4. Compare the disassembly between Windows and Linux drivers for the same shader stage.

CPU-bound process identification Exercise:

1. Write a Python script using `psutil` to list the top 5 processes by CPU usage percentage.
2. Modify the script to flag any process consuming over `80%` CPU for more than `30` seconds.
3. Add functionality to log the flagged processes to a file named `cpu_bottlenecks.log` with timestamps.

Disk I/O latency analysis Exercise:

1. Use the `iostat` command with appropriate flags to measure disk read/write latency on a Linux system.
2. Create a Bash script that runs `iostat` every `5` seconds and saves output to `disk_latency.log`.
3. Parse the log file to identify periods where average wait time exceeds `10` ms, reporting the start/end timestamps.

Database query optimization Exercise:

1. Execute a slow SQL query on a PostgreSQL database with `EXPLAIN ANALYZE`.
2. Identify the most expensive operation from the execution plan (e.g., sequential scan).
3. Rewrite the query using appropriate indexes and compare the execution times.

[GPU Memory Bandwidth Analysis] Exercise:

1. Calculate the theoretical memory bandwidth of an NVIDIA RTX 3090 GPU given:
    - Memory clock: 1219 MHz (effective 9752 MHz with GDDR6X)
    - Memory bus width: 384-bit
    - Transfer rate per pin: 19.5 Gbps
2. Compare this with an AMD RX 6900 XT (512-bit bus, 16 Gbps per pin) and explain which GPU has higher theoretical bandwidth.
3. Write a CUDA kernel using `cudaMallocManaged` that performs a memory bandwidth test by copying 1GB of data between device and host.

[Kernel Occupancy Optimization] Exercise:

1. Given a CUDA kernel with the following characteristics:
    - Threads per block: 256
    - Registers per thread: 64
    - Shared memory per block: 16KB

    Calculate the maximum occupancy on an NVIDIA A100 GPU (compute capability 8.0).

2. Modify the kernel to achieve at least 75% occupancy by adjusting either thread count or resource usage.
3. Profile both versions using

   ```
   nvprof --metrics achieved_occupancy
   ```

   and compare results.

[Cross-Platform Performance Validation] Exercise:

1. Implement a matrix multiplication kernel in both CUDA and HIP that uses shared memory tiling.
2. Measure performance on NVIDIA (e.g., V100) and AMD (e.g., MI100) GPUs using:

   ```
   hipEventRecord(start);
   // Kernel launch
   hipEventRecord(stop);
   ```

3. Create a performance parity metric comparing:

   - GFLOPs achieved
   - Memory bandwidth utilization percentage
   - Kernel duration in milliseconds

Chapter 20

Low-Level Optimization Strategies

20.1 Memory System Optimization

[Cache coherence protocol] Exercise: Consider a multicore system using the MESI cache coherence protocol. A cache line is initially in the `Modified` state in Core 0's cache.

1. List all possible state transitions for this cache line if Core 1 issues a read request.
2. Write the pseudocode for the snooping logic that Core 0 would execute upon detecting this bus transaction.
3. Identify which bus signals would be asserted during this transaction.

[False sharing mitigation] Exercise: A performance-critical structure is declared as:

```
struct {
    int thread1_counter;
    char padding[64];
    int thread2_counter;
} counters;
```

1. Calculate the minimum required size of `padding` to prevent false sharing on a system with 64-byte cache lines.
2. Rewrite the structure using C11 `alignas` instead of explicit padding.
3. Explain why this technique improves performance when accessed by two threads.

[Cache line prefetching] Exercise: Given the following memory access pattern in a 64-byte cache line system:

```
for (int i = 0; i < 1024; i += 8) {
    sum += data[i];
}
```

1. Calculate how many cache lines are touched for array sizes of 8KB and 16KB.
2. Rewrite the loop using GCC's `__builtin_prefetch` to fetch two cache lines ahead.
3. Determine the optimal prefetch distance if the latency to fetch a cache line is 200 cycles.

[TLB Replacement Policy Analysis] Exercise:

1. Compare FIFO and LRU TLB replacement policies by simulating 5 page accesses: `A, B, C, D, A` with a 3-entry TLB.
2. Count TLB misses for each policy and identify which access causes each miss.
3. Modify the sequence to `A, B, C, D, B, A` and repeat the analysis.

4. Explain why one policy outperforms the other in this specific case.

[TLB Preloading Implementation] Exercise:

1. Write x86 assembly code to preload the TLB with entries for virtual addresses `0x4000` to `0x5000` in 4KB increments.
2. Use the `prefetch` instruction and measure execution time with and without preloading.
3. Modify the code to handle a 2MB huge page at `0x200000` and compare TLB performance.
4. List three scenarios where hardware preloading would be more effective than software preloading.

[TLB Shootdown Benchmarking] Exercise:

1. Create a Linux kernel module that triggers TLB shootdowns using `flush_tlb_range()`.
2. Measure shootdown latency for ranges of 1, 10, and 100 pages using `ktime_get_ns()`.
3. Plot the results and identify any non-linear scaling.
4. Propose two optimizations to reduce shootdown overhead in multi-core systems.

Queue arbitration policy analysis Exercise:

1. A memory controller uses a round-robin arbitration policy between 4 request queues. Each queue has the following pending requests: `Q0=3`, `Q1=5`, `Q2=1`, `Q3=4`. Calculate the order of service for the next 12 requests.
2. Implement the arbitration logic in Verilog using a 2-bit counter and priority encoder.

```
module round_robin_arbiter (
  input clk,
  input [3:0] req,
  output reg [1:0] grant
);
// Your code here
endmodule
```

3. Analyze the worst-case latency for a request in `Q1` when all queues are constantly full.

Request queue sizing Exercise:

1. A DDR4 memory controller must buffer 64-byte cache lines. Calculate the minimum queue depth required to hide 100ns DRAM latency when receiving requests at 20 GB/s bandwidth.
2. Derive the relationship between queue depth, request rate, and service rate using Little's Law. Show the formula with all variables defined.
3. Modify the following SystemVerilog struct to include a timestamp field for tracking queue entry age:

```
typedef struct packed {
  logic [63:0] address;
  logic [7:0]  burst_length;
  logic        read_not_write;
} mem_request_t;
```

Priority queue implementation Exercise:

1. Design a hardware-friendly priority queue that sorts requests by `address[15:12]` (to exploit row buffer locality). Draw the block diagram of your design.
2. Write Python code to simulate this priority queue with 8 entries, showing the sorted order after inserting these addresses (hex): `0x1234`, `0x5678`, `0x9ABC`, `0xDEF0`.
3. Calculate the gate count for your design assuming each comparator is 4 gates and each multiplexer is 2 gates per bit (64-bit data path).

Memory reordering analysis Exercise: Given the following code snippet running on an x86 system, analyze potential memory reordering issues:

```
int x = 0, y = 0;

void thread1() {
    x = 1;
    asm volatile("" ::: "memory");  // Compiler barrier
    int r1 = y;
}

void thread2() {
    y = 1;
    asm volatile("" ::: "memory");  // Compiler barrier
    int r2 = x;
}
```

- Identify all possible final values of `r1` and `r2`
- Explain whether the compiler barrier prevents all reordering
- Modify the code to prevent all unwanted reorderings using appropriate memory barriers

Barrier placement optimization Exercise: Consider this lock-free queue implementation:

```
struct queue {
    int *buffer;
    atomic_int head;
    atomic_int tail;
};

void enqueue(struct queue *q, int item) {
    int tail = q->tail.load(memory_order_relaxed);
    q->buffer[tail] = item;
    q->tail.store(tail + 1, memory_order_release);
}

int dequeue(struct queue *q) {
    int head = q->head.load(memory_order_relaxed);
    int item = q->buffer[head];
    q->head.store(head + 1, memory_order_acquire);
    return item;
}
```

- Identify which memory barriers could be safely removed without breaking correctness
- Explain the data dependency relationships between operations
- Rewrite the functions using the minimal necessary barriers

ARM memory model exercise Exercise: Examine this ARMv8 assembly code with weak memory ordering:

```
// Thread 1        // Thread 2
STR X1, [X0]       LDR X2, [X0]
DMB ISH            DMB ISH
STR X3, [X4]       LDR X5, [X4]
```

- List all possible final values of `X2` and `X5`
- Determine if the `DMB` barriers are necessary for correctness

- Rewrite the code using the minimal number of barriers while maintaining sequential consistency

Compare-and-swap implementation Exercise: Implement a thread-safe stack using compare-and-swap (CAS) operations in C++. Your solution must:

1. Define a `node` struct containing `int data` and `node* next`
2. Implement `push()` using `std::atomic_compare_exchange_weak`
3. Implement `pop()` with proper CAS retry logic for empty stack handling
4. Include memory ordering constraints with `std::memory_order_release` for writes
5. Test with 4 concurrent threads pushing and popping 1000 elements each

Atomic flag synchronization Exercise: Create a ticket lock system using `std::atomic_flag` in C++ that:

1. Implements a `TicketLock` class with `std::atomic_flag` for synchronization
2. Uses `test_and_set()` with `std::memory_order_acquire`
3. Provides `lock()` and `unlock()` methods
4. Handles 8 threads incrementing a shared counter to 50000
5. Measures average acquisition time using `std::chrono`

Memory ordering practical Exercise: Analyze and fix a broken double-checked locking implementation:

1. Identify the memory ordering bug in given `Singleton` class
2. Rewrite using `std::atomic` with `std::memory_order_acquire`/release
3. Prove thread safety using `std::thread` stress tests
4. Measure performance impact versus naive mutex approach
5. Document the happens-before relationships in your solution

20.2 Instruction Scheduling

File compilation dependencies Exercise: Given the following C project structure:

```
src/
  main.c
  utils.c
  utils.h
  parser.c
  parser.h
```

1. List all direct dependencies of `main.c` assuming it includes both `utils.h` and `parser.h`
2. Write the minimal `gcc` command to compile `main.c` into an executable considering all dependencies
3. Identify which files would need recompilation if `utils.h` is modified

Makefile dependency graph Exercise: Given this Makefile snippet:

```
program: main.o utils.o parser.o
    gcc -o program main.o utils.o parser.o

main.o: main.c utils.h parser.h
utils.o: utils.c utils.h
parser.o: parser.c parser.h parser_internal.h
```

1. Draw the dependency graph as a directed acyclic graph (DAG)
2. List all leaf nodes in the dependency graph
3. Determine which targets would rebuild if `parser_internal.h` changes

Package dependency resolution Exercise: Given these Python package requirements:

```
pkgA==1.2.0  # requires pkgC>=2.0
pkgB==3.1.4  # requires pkgC<2.5
pkgC==2.3.0
```

1. Verify if the current dependency constraints are satisfiable
2. Identify which package would need version adjustment if `pkgB` changes to require `pkgC>=3.0`
3. Write a pip install command that explicitly enforces these version constraints

Memory allocation conflict Exercise:

1. Write a C function `allocate\_buffer` that dynamically allocates a 256-byte buffer using `malloc`.
2. Modify the function to check if the allocation succeeded and return `NULL` on failure.
3. Add thread-safety by wrapping the allocation in a mutex lock/unlock pair using `pthread\_mutex\_t`.
4. Create a test case where two threads simultaneously call `allocate\_buffer` and verify no memory corruption occurs.

File access contention Exercise:

1. Implement a Python script that writes timestamps to `/var/log/app\_log.txt` every second.
2. Modify the script to use `fcntl.flock` for exclusive file locking.
3. Create a second script that attempts to read the log file while the first script is running.
4. Verify the reader script waits for the lock using `timeout=10` in the flock call.

Database transaction isolation Exercise:

1. Set up a PostgreSQL table `inventory` with columns `item\_id` and `stock`.
2. Start two concurrent transactions that both attempt to decrement stock for `item\_id=100`.
3. Use `BEGIN ISOLATION LEVEL SERIALIZABLE` and handle serialization failures.
4. Implement retry logic with exponential backoff when conflicts occur.

Load-store reordering analysis Exercise:

1. Given the following assembly code snippet, identify all pairs of instructions that could be reordered without changing the program's behavior:

   ```
   ldr x0, [x1]
   add x2, x3, x4
   str x5, [x6]
   mul x7, x8, x9
   ```

2. For each reorderable pair you identified, explain why the reordering is safe using memory dependency rules.
3. Modify the code by adding one instruction that would prevent all possible reorderings between the load and store operations.

Memory barrier implementation Exercise:

1. Write an ARM assembly sequence that performs two stores to different memory locations `data1` and `data2`, with a memory barrier ensuring `data1` is always written before `data2`.
2. Convert your assembly to equivalent C code using the `__atomic_thread_fence()` intrinsic.
3. Explain what would happen if the barrier was removed in terms of observable processor behavior.

Out-of-order pipeline simulation Exercise:

1. Design a 5-stage pipeline (fetch, decode, execute, memory, writeback) that can reorder independent arithmetic instructions but must maintain load-store ordering.
2. Simulate the execution of this instruction sequence through your pipeline:

```
ldr r0, [r1]
add r2, r3, r4
mul r5, r6, r7
str r8, [r9]
```

3. Show the cycle-by-cycle pipeline diagram and highlight where reordering occurs.

[Manual loop unrolling] Exercise: Given the following C function that computes the sum of an array:

```
int sum_array(int* arr, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}
```

1. Manually unroll the loop by a factor of 4, assuming `size` is always a multiple of 4.
2. Modify the unrolled version to handle cases where `size` is not a multiple of 4.
3. Compare the instruction count between the original and unrolled versions for `size = 100`.

[Compiler directives for unrolling] Exercise:

1. Write a C function that multiplies two `N x N` matrices using nested loops.
2. Add `#pragma unroll` directives to unroll the innermost loop by a factor of 8.
3. Compile both versions with `-O3` and compare the assembly output using `objdump`.
4. Measure the runtime difference for `N = 256` using `clock_gettime()`.

[Performance analysis of unrolling] Exercise:

```
float dot_product(float* a, float* b, int n) {
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

1. Create unrolled versions with factors 2, 4, and 8.
2. Benchmark each version for `n = 1024` using `perf stat`.
3. Plot cycles per iteration versus unroll factor.
4. Identify the point of diminishing returns in unrolling.

[Loop unrolling analysis] Exercise: Given the following C code snippet:

```
for (int i = 0; i < 100; i++) {
    a[i] = b[i] * c[i];
    d[i] = a[i] + e[i];
}
```

1. Manually unroll this loop by a factor of 4, showing the transformed code.
2. Identify all data dependencies in the original and unrolled versions.
3. Calculate the theoretical speedup from unrolling, assuming a 5-stage pipeline.

[Pipeline hazard resolution] Exercise: Consider this assembly code for a RISC processor:

```
LOOP:  LD   R1, 0(R2)     ; Load from memory
       ADD  R3, R1, R4    ; Add operation
       SD   R3, 0(R5)     ; Store to memory
       ADDI R2, R2, 4     ; Increment pointer
       ADDI R5, R5, 4     ; Increment pointer
       BNE  R2, R6, LOOP  ; Branch if not equal
```

1. Identify all pipeline hazards in this loop.
2. Propose three software solutions to mitigate these hazards.
3. Rewrite the code with your chosen hazard resolution technique.

[Software pipelining implementation] Exercise: Given the pseudocode for a FIR filter:

```
for (i = 0; i < N; i++) {
    sum = 0;
    for (j = 0; j < M; j++) {
        sum += x[i+j] * h[j];
    }
    y[i] = sum;
}
```

1. Apply software pipelining to this nested loop, showing the transformed code.
2. Calculate the minimum initiation interval for your pipelined version.
3. Implement your solution in C using `#pragma` directives for a specific compiler.

20.3 Register Optimization

[Instruction scheduling] Exercise:

1. Given the following x86 assembly code, identify the instruction with the highest register pressure:

   ```
   mov eax, [mem1]
   add ebx, eax
   mov ecx, [mem2]
   imul edx, ecx, 5
   add eax, edx
   ```

2. Rewrite the code to reduce register pressure by reordering instructions while maintaining correctness.
3. Calculate the maximum number of live registers before and after your optimization.

[Loop unrolling] Exercise:

1. Analyze the register pressure for this unrolled loop (2 iterations) on ARM:

```
ldr r0, [r1], #4
add r2, r2, r0
ldr r0, [r1], #4
add r2, r2, r0
```

2. Unroll the loop to 4 iterations and calculate the new register pressure.
3. Modify the unrolled version to use different registers, keeping the same functionality but reducing pressure.

[Spill code generation] Exercise:

1. Given this MIPS code with high register pressure:

```
lw $t0, 0($a0)
add $t1, $t0, $a1
lw $t2, 4($a0)
mul $t3, $t1, $t2
sw $t3, 8($a0)
```

2. Identify which register should be spilled to memory first.
3. Rewrite the code with spill/reload instructions for your chosen register.
4. Calculate the total memory accesses added by your spill code.

[Identifying Live Ranges] Exercise: Given the following code snippet, identify all live ranges for the variable `temp_var`. For each live range, specify the start and end instructions where the variable is live. Use the format: "Live range N: from instruction X to instruction Y."

```
1: temp_var = load(a)
2: b = temp_var + 5
3: store(b, temp_var)
4: c = temp_var * 2
5: temp_var = load(d)
6: e = temp_var - 3
7: store(e, temp_var)
```

- List all live ranges for `temp_var` in the above code.

[Splitting Live Ranges] Exercise: Modify the following code to split the live range of `result` into two separate live ranges. Insert a new variable `result2` to achieve this. Ensure the program semantics remain unchanged.

```
1: result = x + y
2: z = result * 2
3: print(result)
4: result = a - b
5: w = result / 4
6: print(result)
```

- Rewrite the code with split live ranges for `result`.
- Verify that the output of the program remains identical.

[Register Allocation Impact] Exercise: The following assembly code uses `%r1` for two overlapping live ranges of `val`. Rewrite the code to split the live ranges and use `%r2` for the second live range.

```
load %r1, [mem_x]
add %r1, %r1, 10
store %r1, [mem_y]
load %r1, [mem_z]
sub %r1, %r1, 5
store %r1, [mem_w]
```

- Identify the two live ranges of `val` in the original code.
- Rewrite the assembly code with split live ranges using `%r2`.
- Ensure no data dependencies are violated.

[Live-range analysis] Exercise:

1. Given the following code snippet, identify all live ranges for variable `tmp_1`:

```
mov tmp_1, r1
add r2, tmp_1, r3
mul r4, tmp_1, r2
str r4, [sp, #8]
mov tmp_1, r5
add r6, tmp_1, r7
```

2. For each live range found, specify the start and end instruction numbers (1-6).
3. Determine if `tmp_1` could be coalesced with `r1` without conflicts.

[Interference graph construction] Exercise:

1. Given these register live ranges:
   - `v1`: instructions 1-4
   - `v2`: instructions 3-6
   - `v3`: instructions 1-3
   - `v4`: instructions 5-7
2. Draw the interference graph showing all edges.
3. Identify which pairs of variables could potentially be coalesced.
4. Calculate the graph coloring degree for `v2`.

|[Coalescing implementation] Exercise:

1. Given this assembly code with virtual registers:

```
ldr %v1, [sp]
add %v2, %v1, #4
mov %v3, %v2
str %v3, [sp, #8]
```

2. Apply conservative coalescing to merge `%v2` and `%v3`.
3. Rewrite the code after coalescing.
4. List any potential spill costs that would prevent aggressive coalescing.

[Register Allocation Spill Analysis] Exercise: Given the following code snippet and register pressure analysis:

```
int foo(int a, int b, int c, int d) {
    int x = a * b;
    int y = c + d;
    int z = x * y;
    int w = z / a;
    return w + y;
}
```

1. Identify which variables would be spilled first if only 4 registers are available.
2. Rewrite the function with explicit stack spills/restores for the identified variables.
3. Calculate the total spill cost in memory accesses for your rewritten version.

[Spill Code Placement] Exercise: Consider this basic block with live ranges:

```
1: t1 = load a
2: t2 = load b
3: t3 = t1 + t2
4: t4 = t3 * t1
5: store t4, c
6: t5 = t4 - t2
7: store t5, d
```

1. Determine the optimal spill point for `t3` if it must be spilled.
2. Insert spill/store and reload/load instructions for `t3` at your chosen location.
3. Verify your solution by showing the updated live ranges after spilling.

[Spill Cost Heuristics] Exercise: Given the following control flow graph with execution frequencies:

```
      [A] (50%)
     /   \
[B] (30%) [C] (20%)
     \   /
      [D] (100%)
```

1. Calculate the spill cost for variable `v1` used in blocks A, B, and D.
2. Compare this with the spill cost for `v2` used in blocks C and D.
3. Decide which variable should be spilled first based on your calculations.

[Tomasulo's Algorithm Analysis] Exercise:

1. Given a Tomasulo-based processor with 4 reservation stations for the FP adder, draw the reservation station status table after executing the following instructions:

```
LD F6, 32(R2)
LD F2, 44(R3)
MULT F0, F2, F4
SUB F8, F6, F2
DIV F10, F0, F6
```

2. Identify all instances where register renaming occurs in the above instruction sequence.
3. Calculate how many physical registers would be needed to support this execution without WAW hazards.

[ROB Implementation] Exercise:

1. Design a 16-entry Reorder Buffer (ROB) with the following fields: `valid_bit`, `instruction_type`, `destination`, `value`, and `ready_bit`. Specify the bit width for each field.
2. Write a Verilog code snippet for the ROB's commit logic that checks `ready_bit` and `valid_bit` before committing.
3. Calculate the total storage overhead (in bits) for your ROB design, including all control bits.

[Physical Register File Design] Exercise:

1. A processor has 32 architectural registers and uses 64 physical registers. Draw the register alias table (RAT) structure and specify its width in bits.
2. Compute the number of read/write ports needed for the physical register file when issuing 4 instructions per cycle, where each instruction has up to 2 source operands and 1 destination.
3. Propose a circuit design for handling RAT updates during branch mispredictions, showing the critical path.

Chapter 21

Practical Applications

21.1 Scientific Computing

FFT radix-2 implementation Exercise: Implement a radix-2 FFT algorithm in Python using the Cooley-Tukey approach.

1. Write a function `fft_radix2(x)` that computes the FFT of a 1D input array `x` with length $N = 2^n$.
2. Pre-compute twiddle factors using `np.exp(-2j * np.pi * np.arange(N//2)/ N)`.
3. Use recursion to split the problem into even and odd-indexed subproblems.
4. Validate your implementation against `np.fft.fft` for a 256-point random complex input.

FFT memory optimization Exercise: Optimize an in-place FFT implementation to reduce memory usage.

1. Modify the radix-2 algorithm to operate in-place using a single array `X`.
2. Implement the bit-reversal permutation step without auxiliary memory.
3. Measure peak memory usage using `memory_profiler` for $N = 2^{20}$ points.
4. Compare with a non-in-place version and report the memory reduction percentage.

Real-valued FFT optimization Exercise: Implement the real-valued FFT optimization for a 1D signal.

1. Write a function `rfft(x)` that exploits conjugate symmetry for real inputs.
2. Pack two real signals `x` and `y` into a single complex array using `z = x + 1j*y`.
3. Compute `Z = fft(z)` and extract `X` and `Y` using the symmetry relations.
4. Verify your results against `np.fft.rfft` for a 1024-point random real input.

[5-point stencil implementation] Exercise: Implement a 5-point stencil operation for heat diffusion simulation. The stencil computes each output cell as the average of itself and its 4 direct neighbors (N,S,E,W).

1. Create a 100x100 input matrix `A` with random values between 0 and 100
2. Implement the stencil operation without using boundary checks
3. Handle boundaries by assuming `A[i,j] = 0` for out-of-bound indices
4. Store results in matrix `B` of same size
5. Measure execution time for 100 iterations using `time.time()`

[Stencil optimization] Exercise: Optimize a 7-point 3D stencil computation for a 50x50x50 grid.

1. Initialize a 3D array `data` with values from `np.random.rand`

2. Implement the naive stencil: each point averages itself and 6 face-adjacent neighbors
3. Create a version using NumPy array slicing for optimization
4. Compare execution times for both versions
5. Calculate the L2 norm difference between both results

[Boundary condition handling] Exercise: Implement different boundary conditions for a 9-point 2D stencil.

1. Create a 200x200 grid with initial temperature distribution
2. Implement Dirichlet boundary (constant value 20 at edges)
3. Implement Neumann boundary (zero derivative at edges)
4. Implement periodic boundary conditions
5. Visualize results after 50 iterations using `matplotlib`

[COO to CSR conversion] Exercise: Given a sparse matrix in COO (Coordinate List) format with three arrays:

```
row_ind = [0, 1, 1, 2, 2, 2]
col_ind = [0, 1, 2, 0, 1, 2]
values = [3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
```

1. Convert this matrix to CSR (Compressed Sparse Row) format manually
2. Write Python code using `scipy.sparse` to verify your conversion
3. Calculate the memory savings (in bytes) compared to dense storage for a 1000×1000 matrix with 1% nonzero elements

[Sparse matrix-vector multiplication] Exercise: Implement sparse matrix-vector multiplication for CSR format:

1. Write a function `spmv_csr` that takes CSR arrays (`data`, `indices`, `indptr`) and a vector `x`
2. Test your function with the matrix from the previous exercise and vector `x = [1, 2, 3]`
3. Compare the FLOP count with dense multiplication for an n×n matrix with k nonzeros
4. Optimize your function to skip zero elements in `x` (if known)

[Diagonal storage optimization] Exercise: A tridiagonal matrix has nonzeros only on the main diagonal and first sub/super-diagonals:

1. Design a storage scheme using three 1D arrays instead of CSR
2. Implement matrix-vector multiplication for your scheme in Python
3. Calculate the memory footprint for an n×n matrix in your scheme versus CSR
4. Extend your scheme to handle banded matrices with bandwidth `b`

Linear Congruential Generator Implementation Exercise:

1. Implement a linear congruential generator (LCG) in Python using the recurrence relation: `X_{n+1} = (a * X_n + c)% m`.
2. Use parameters `a = 1664525`, `c = 1013904223`, and `m = 2**32` with seed `X_0 = 12345`.
3. Generate 10 pseudorandom numbers and print them with 8 decimal places.
4. Calculate the period length by detecting when the sequence repeats the seed value.

Exponential Distribution Sampling Exercise:

1. Using the inverse transform method, write a Python function that samples from an exponential distribution with rate parameter `lambda = 1.5`.
2. Your function should accept a uniform random number `u` from `U(0,1)` as input.
3. Generate 1000 samples using your function with `numpy.random.uniform()` as the source of `u`.
4. Plot a histogram of the results with 20 bins and compare it to the theoretical PDF.

Mersenne Twister Verification Exercise:

1. Using Python's `random` module (which uses Mersenne Twister), generate 10000 random integers between 0 and 99.
2. Compute the sample mean and variance, comparing them to the theoretical values for a discrete uniform distribution.
3. Perform a chi-squared goodness-of-fit test with 10 equal-probability bins (0-9, 10-19, ..., 90-99).
4. At significance level `alpha = 0.05`, determine if the sample passes the test for uniformity.

21.2 Real-Time Graphics

[Ray-Sphere Intersection] Exercise: Implement a ray-sphere intersection test in x86-64 assembly. Assume the ray origin is at `ray_origin`, direction at `ray_dir`, and sphere center at `sphere_center` with radius `sphere_radius`.

- Compute the vector from ray origin to sphere center: `L = sphere_center - ray_origin`.
- Calculate the projection of `L` onto `ray_dir`: `t_ca = dot(L, ray_dir)`.
- If `t_ca` is negative, return no intersection.
- Compute squared distance from sphere center to ray: `d^2 = dot(L,L)- t_ca*t_ca`.
- If `d^2 > sphere_radius*sphere_radius`, return no intersection.
- Otherwise, compute intersection distance `t = t_ca - sqrt(sphere_radius*sphere_radius - d^2)`.
- Return intersection point `ray_origin + t*ray_dir`.

[SIMD Ray Batching] Exercise: Optimize ray tracing using AVX-512 instructions to process 8 rays simultaneously. Assume rays are stored in SoA format:

```
struct RayBatch {
    float origin_x[8];
    float origin_y[8];
    float origin_z[8];
    float dir_x[8];
    float dir_y[8];
    float dir_z[8];
};
```

- Load 8 ray origins into `zmm0-zmm2` (x,y,z components).
- Load 8 ray directions into `zmm3-zmm5`.
- Compute dot products of directions with themselves using `vdpbf32ps`.
- Calculate reciprocal square roots for normalization using `vrsqrt14ps`.
- Normalize all 8 directions in parallel.
- Store results back to memory.

[Bounding Volume Hierarchy] Exercise: Implement a BVH node intersection test in assembly. Assume node data:

```
struct BVHNode {
    float min[3];  // bbox min
    float max[3];  // bbox max
    uint32_t children[2];  // child indices
};
```

- Load bounding box min/max into `xmm0-xmm3`.
- Compute ray inverse direction (1/dx,1/dy,1/dz) using `divps`.
- Calculate t-values for all 6 planes using `minps`/`maxps`.
- Determine if `t_min <= t_max` for intersection.
- If intersected, push child node pointers onto stack (simulated via `push`).
- Otherwise, skip subtree traversal.

[Shader Loop Unrolling] Exercise:

1. Write a Vulkan GLSL compute shader that calculates the sum of an array of 32 floating-point values using a loop.
2. Modify the shader to manually unroll the loop by a factor of 4, ensuring the logic remains equivalent.
3. Benchmark both versions using `vkCmdDispatch` and `VK_QUERY_TYPE_TIMESTAMP`, reporting the performance difference.

[Memory Access Coalescing] Exercise:

1. Create a Vulkan fragment shader that samples from a 1024x1024 `VK_FORMAT_R32G32B32A32_SFLOAT` image with random UV coordinates.
2. Rewrite the shader to use `texelFetch` with integer coordinates aligned to 4x4 pixel blocks.
3. Compare the GPU cycle counts using `VK_EXT_pipeline_robustness` instrumentation.

[Push Constant Optimization] Exercise:

1. Implement a Vulkan pipeline with a vertex shader that reads 12 uniform variables via `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`.
2. Convert the uniforms to use `VK_PIPELINE_LAYOUT_CREATE_PUSH_CONSTANT_BIT` with a 128-byte limit.
3. Measure the reduction in `vkCmdBindDescriptorSets` calls per frame using `VK_EXT_debug_utils` markers.

Nearest-neighbor filtering implementation Exercise:

1. Implement a nearest-neighbor texture sampler in GLSL that samples from a 2D texture.
2. The function signature should be `vec4 nearestNeighbor(sampler2D tex, vec2 uv, vec2 textureSize)`.
3. Handle texture coordinates that are outside the [0,1] range by clamping them.
4. Test your implementation by creating a checkerboard pattern texture and sampling it at various coordinates.

Mipmap level calculation Exercise:

1. Write a function to compute the appropriate mipmap level for anisotropic filtering.
2. The function signature should be `float computeMipLevel(vec2 ddx, vec2 ddy, vec2 textureSize)`.
3. Use the formula: max level from derivatives in x and y directions.

4. Verify your function by comparing with built-in `textureQueryLod` results.

Bilinear filtering optimization Exercise:

1. Optimize a bilinear texture sampler by reducing the number of texture fetches.
2. Start with a reference implementation using 4 `texture2D` calls.
3. Modify it to use `textureGather` for improved performance.
4. Compare the visual quality and performance of both implementations.

21.3 Machine Learning

[1D Convolution Implementation] Exercise: Implement a 1D discrete convolution function in Python that:

- Takes input arrays `x` and `h` as arguments
- Returns the full convolution result (non-circular)
- Uses zero-padding for edge handling
- Does not use any built-in convolution functions
- Includes a docstring explaining input/output dimensions

```
# Your implementation here
def convolve_1d(x, h):
    pass
```

[Image Filter Application] Exercise: Apply a 3x3 Sobel edge detection filter to a grayscale image:

- Load an image using `cv2.imread` in grayscale mode
- Manually define the Sobel X and Y kernels as numpy arrays
- Compute both directional gradients using your convolution function
- Combine gradients to create the final edge map
- Display original and processed images using `matplotlib`

[Convolution Complexity Analysis] Exercise: Analyze the computational complexity of 2D convolution:

- Derive the time complexity for an MxN image and KxK kernel
- Compare direct convolution vs FFT-based methods
- Calculate the theoretical speedup when K=7 and M=N=1024
- Implement both methods and measure actual execution times
- Plot the results using `matplotlib.pyplot`

Forward pass implementation Exercise: Implement the forward pass of batch normalization in Python for a 2D input tensor. Assume the input is `x` with shape `(batch_size, features)`.

1. Compute the mean `mu` and variance `sigma_squared` along the batch dimension (axis=0).
2. Normalize the input using `(x - mu)/ sqrt(sigma_squared + epsilon)` where `epsilon=1e-5`.
3. Scale and shift using learned parameters `gamma` and `beta` (both of shape `(features,)`).
4. Return the normalized output and a tuple `(mu, sigma_squared)` for the backward pass.

```
def batchnorm_forward(x, gamma, beta, eps=1e-5):
    # Your implementation here
```

Gradient derivation Exercise: Derive the gradients for batch normalization parameters during backpropagation. Given upstream gradient `dout`, normalized input `x_hat`, and cached statistics `(mu, sigma_squared)`:

1. Write the expression for `dbeta` (gradient of beta).
2. Write the expression for `dgamma` (gradient of gamma).
3. Derive the expression for `dx` (gradient w.r.t. input) using the chain rule.
4. Show how to compute `dx` using intermediate terms `dx_hat` and `dsigma_squared`.

ConvNet integration Exercise: Modify a CNN layer to include batch normalization:

1. Insert batch norm after a `nn.Conv2d` layer but before ReLU in PyTorch.
2. Initialize `gamma` as ones and `beta` as zeros for a layer with `C` output channels.
3. Ensure the batch norm layer tracks running statistics during training but uses them during inference.
4. Write the forward pass logic for both training and evaluation modes.

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=3)
        self.bn = # Your initialization here
        self.relu = nn.ReLU()

    def forward(self, x):
        # Your implementation here
```

[Finite difference gradient verification] Exercise:

1. Implement a Python function `f(x)` that computes $f(x) = x_1^2 + 3x_2^3 - x_1x_2$ where $x = [x_1, x_2]^T$.
2. Write a function `finite_difference_gradient(f, x, h=1e-5)` that computes the gradient using central differences.
3. Test your implementation at $x = [2, -1]^T$ with $h = 0.001$ and compare against the analytical gradient $\nabla f = [2x_1 - x_2, 9x_2^2 - x_1]^T$.
4. Report the absolute error for each component and suggest reasons for any discrepancies.

[Sparse Jacobian computation] Exercise:

1. Consider a function $F : \mathbb{R}^4 \to \mathbb{R}^3$ with the following sparse Jacobian pattern:

```
[ * * 0 0 ]
[ 0 * * 0 ]
[ * 0 0 * ]
```

2. Design a coloring scheme that allows computing the full Jacobian with only 2 function evaluations.
3. Implement your scheme in Python using `numpy`, assuming you have access to `F(x)`.
4. Verify your implementation for $F(x) = [x_1 + x_2, x_2x_3, x_1 + x_4]^T$ at $x = [1, 2, 3, 4]^T$.

[Automatic differentiation implementation] Exercise:

1. Create a Python class `DualNumber` with attributes `value` and `derivative`.
2. Implement `__add__`, `__mul__`, and `__pow__` methods to handle automatic differentiation rules.
3. Use your class to compute the gradient of $f(x, y) = (x + 2y)^3$ at $(1, 2)$.
4. Compare your result with both symbolic differentiation and finite differences.

Chapter 22

Performance Analysis Techniques

22.1 Performance Counters

Performance counter analysis Exercise:

1. Using `perf stat`, measure the `instructions` and `cpu-cycles` counters for a matrix multiplication kernel.
2. Calculate the CPI (cycles per instruction) from your measurements.
3. Identify which of these hardware events would help diagnose cache misses: `L1-dcache-load-misses`, `branch-misses`, or `stalled-cycles-frontend`.
4. Modify the matrix multiplication to use blocking and remeasure the counters.

Branch prediction investigation Exercise:

1. Write a C function with a loop containing an unpredictable `if` condition using `rand()`.
2. Measure `branch-misses` using `perf stat -e branch-misses`.
3. Rewrite the condition to be predictable (e.g., `i % 2 == 0`) and compare measurements.
4. Explain how the branch predictor might behave differently in both cases.

Cache hierarchy profiling Exercise:

1. Profile a memory-intensive workload using `perf stat -e LLC-load-misses,LLC-store-misses`.
2. Calculate the miss rate for last-level cache accesses.
3. Run the same workload with different array sizes (smaller/larger than L3 cache).
4. Correlate the performance counters with observed execution times.

Poisson process simulation Exercise:

1. Simulate a Poisson process with rate `lambda = 0.5` events per second for 60 seconds.
2. Plot the event times as points on a timeline using `matplotlib`.
3. Calculate and print the empirical mean inter-arrival time.
4. Compare with the theoretical expectation `1/lambda`.

Event-driven system analysis Exercise:

1. Given a CSV file `event_log.csv` with columns `timestamp` and `event_type`:
2. Read the data using `pandas.read_csv()`.
3. Plot the cumulative number of events over time for each `event_type`.

4. Calculate the 90th percentile of inter-arrival times for `event_type = "alert"`.

Real-time event buffer Exercise:

1. Implement a circular buffer class `EventBuffer` with capacity 1000.
2. The buffer must store tuples of `(timestamp, event_data)`.
3. Write methods `add_event()` and `get_oldest()`.
4. Handle buffer overflow by overwriting oldest events.
5. Test with synthetic timestamps from `time.time()`.

Data hazard identification Exercise:

1. Given the following MIPS assembly code, identify all data hazards (RAW, WAR, WAW) between instructions:

```
add $t0, $t1, $t2
sub $t3, $t0, $t4
lw $t0, 0($t5)
and $t6, $t0, $t7
```

2. For each hazard found, specify the instruction pair and hazard type.
3. Calculate how many stall cycles would be needed if no forwarding is used.

Forwarding unit design Exercise:

1. Design a forwarding unit for a 5-stage pipeline that handles `EX/MEM` and `MEM/WB` register forwarding.
2. List all possible input signals needed (e.g., `EX/MEM.RegWrite`, register numbers).
3. Write the logic equations for forwarding control signals to the ALU inputs.
4. Sketch the circuit diagram for one forwarding multiplexer.

Branch penalty calculation Exercise:

1. A pipeline has 5 stages with branch resolution in EX stage. Calculate the branch penalty cycles.
2. Modify the pipeline to move branch resolution to ID stage, showing the new penalty.
3. Given a 20% branch frequency and 60% prediction accuracy, compute the overall CPI increase.
4. Suggest one improvement to reduce branch penalties and quantify its benefit.

Three-state cache analysis Exercise:

1. A direct-mapped cache has 64 lines with 32 bytes per line. Main memory is byte-addressable with 20-bit addresses. Compute the number of bits used for tag, index, and offset fields.
2. For the cache in question 1, show the binary address decomposition for address `0xABCDE`.
3. Classify the cache miss type (compulsory, capacity, conflict) for this access pattern: `for (int i=0; i<128; i++){ access A[i*64]; }` when array `A` is aligned to a cache line.

Cache replacement policy Exercise:

1. Implement a basic LRU cache simulator in Python that tracks hits/misses for a 4-way set-associative cache with 8 sets. Use the following skeleton:

```
class LRUCache:
    def __init__(self, ways, sets):
        # Initialize data structures here
        pass

    def access(self, address):
        # Update LRU state and return hit/miss
        pass
```

2. Run your simulator with this sequence: `[0, 4, 8, 12, 0, 4, 8, 16, 20, 24]`. Report the final hit rate.
3. Modify your simulator to implement FIFO replacement. Compare hit rates for both policies on the same sequence.

Prefetching impact analysis Exercise:

1. Given a stride prefetcher with lookahead=2, predict the prefetch addresses for this sequence: `[0x1000, 0x1008, 0x1010, 0x1018]`.
2. Calculate the potential speedup from prefetching for a loop accessing `array[i]` with 64-byte cache lines, assuming:
    - 100 cycle memory latency
    - 1 cycle L1 cache access
    - 1 memory request per 4 cycles
3. Identify one scenario where stride prefetching would degrade performance and explain why in one sentence.

Memory access patterns Exercise:

1. Calculate the effective bandwidth (GB/s) for a system with:
    - 64-bit DDR4 memory running at 3200 MHz
    - 4 memory channels
    - CAS latency of 16 cycles
    - Burst length of 8 transfers
    - Assume 1:1 command-to-data ratio
2. Rewrite the following C code to improve spatial locality:

```
for (int j = 0; j < 1024; j++) {
    for (int i = 0; i < 1024; i++) {
        A[i][j] = B[j][i] * C[i][j];
    }
}
```

3. Explain how a stride of `N+1` in a `N x N` matrix affects cache performance.

DRAM timing Exercise:

1. Compute the total latency (ns) for a memory read operation with:
    - tCAS = 15 cycles
    - tRCD = 17 cycles
    - tRP = 15 cycles

- Memory clock period = 0.625 ns

2. Identify which DRAM timing parameter (tCAS, tRCD, tRP) becomes critical for:
   - Random access patterns
   - Sequential access patterns
   - Bank conflicts
3. Design a memory access pattern that minimizes tRCD overhead.

Bandwidth optimization Exercise:

1. Calculate the peak theoretical bandwidth for:
   - GDDR6 memory with 16 Gbps/pin
   - 384-bit memory interface
   - 8 memory chips
2. Modify this CUDA kernel to improve memory coalescing:

```
__global__ void copy(float* out, float* in) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    out[tid * 4] = in[tid * 4];
}
```

3. Compare the bandwidth impact of using `float4` versus `float` in GPU memory transfers.

22.2 Optimization Methodology

Memory Leak Detection Exercise:

1. Analyze the following C code snippet using a static analysis tool like `cppcheck` or `clang-tidy`:

```
void load_data() {
    char* buffer = malloc(1024);
    if (condition) {
        return; // Potential leak
    }
    free(buffer);
}
```

2. Identify the memory leak scenario and modify the code to prevent it.
3. Write a configuration file for the static analyzer to flag similar leaks in a project.

Buffer Overflow Analysis Exercise:

1. Use `gcc -fstack-protector` and static analysis to evaluate this vulnerable code:

```
void copy_string(char* input) {
    char local_buf[16];
    strcpy(local_buf, input);
}
```

2. List three static analysis warnings that would be generated for this code.
3. Rewrite the function using `strncpy` with proper bounds checking.

Type Safety Verification Exercise:

1. Run `PVS-Studio` on this C++ code and document the type safety issues:

```
void process(int* data) {
    float* ptr = (float*)data; // Unsafe cast
    *ptr = 3.14f;
}
```

2. Replace the unsafe cast with a C++ `static_cast` or safer alternative.
3. Configure the analyzer to treat all C-style casts as errors in your build system.

[Tracepoint instrumentation] Exercise: Implement a Python decorator `@trace_execution` that logs function entry and exit. The decorator must:

- Print the function name when called, using `f.__name__`
- Print all positional arguments (values only)
- Print the return value when the function exits
- Handle both successful returns and exceptions
- Use `logging.debug()` instead of `print()`

[Dynamic code analysis] Exercise: Write a bash script that monitors a running Python process and:

- Takes the PID as a command-line argument
- Uses `strace` to capture system calls
- Filters output to only show file operations (`open`, `read`, `write`)
- Logs results to `/tmp/python_file_ops.log`
- Runs until manually interrupted with `Ctrl+C`

[Execution profiling] Exercise: Create a C program that measures the overhead of:

- Function calls with different numbers of arguments (0 to 6)
- Empty vs non-empty function bodies
- Using `clock_gettime(CLOCK_MONOTONIC)` for timing
- Outputting results in CSV format: `arg_count,empty_body,nsec`
- Compiling with both `-O0` and `-O3` flags

Throughput analysis in a manufacturing line Exercise:

1. A factory produces widgets through three stations: cutting (20 widgets/hour), assembly (15 widgets/hour), and packaging (25 widgets/hour). Calculate the bottleneck station.
2. The assembly station's throughput improves to 22 widgets/hour after maintenance. Recalculate the bottleneck.
3. Propose two methods to balance the line after the improvement in (2), showing calculations for each method.

Database query optimization Exercise:

1. A query on `customer_orders` table with 1M records takes 4.2 seconds. The `EXPLAIN` output shows a full table scan. Identify the likely bottleneck.
2. Rewrite the query `SELECT * FROM customer_orders WHERE order_date > '2023-01-01'` to avoid the bottleneck, assuming an index exists on `order_date`.

3. The same query now takes 0.8 seconds but still slows during peak hours. Suggest two server-side improvements beyond query optimization.

Network bandwidth calculation Exercise:

1. A video streaming server has 1 Gbps uplink and serves 2000 concurrent clients at 480p (1.5 Mbps/stream). Calculate the bottleneck and percentage utilization.
2. The provider upgrades to 2.5 Gbps uplink. How many 1080p (5 Mbps/stream) clients can now be served without bottleneck?
3. During peak hours, packet loss occurs at 80

CPU Load Calculation Exercise:

1. A server has 4 CPU cores with the following utilization over 5 minutes: 65%, 72%, 68%, and 71%. Calculate the average CPU load.
2. Using `top` output showing `us=45`, `sy=15`, `ni=2`, and `id=38`, determine the total CPU usage percentage.
3. Write a Bash command using `mpstat` to sample CPU utilization every 2 seconds for 10 iterations and save to `cpu\_log.txt`.

Memory Allocation Exercise:

1. A process has resident memory of 512 MB and shared memory of 128 MB. Calculate its unique memory footprint.
2. Given `MemTotal=16384MB` and `MemAvailable=4096MB` from `/proc/meminfo`, compute used memory percentage.
3. Write a Python snippet using `psutil` to print current swap usage in GB with 2 decimal places.

Disk I/O Analysis Exercise:

1. A disk shows `r\_await=4.2ms` and `w\_await=7.8ms` in `/proc/diskstats`. Convert these to seconds in scientific notation.
2. Using `iostat -dx 1` output with `util=78\%` and `svctm=5.4ms`, calculate the average I/O queue length.
3. Write a Linux command to list all files larger than 100MB in `/var/log` sorted by size, using `find` and `sort`.

Pipeline stall analysis Exercise: A 5-stage RISC processor (IF, ID, EX, MEM, WB) has the following characteristics:

- Clock frequency: 2 GHz
- Memory access latency: 4 cycles for cache miss
- Branch misprediction penalty: 3 cycles
- Data hazard stall: 2 cycles (when not forwarded)

Analyze the following code snippet:

```
LOOP:  lw   $t0, 0($s0)
       addi $t1, $t0, 1
       sw   $t1, 0($s0)
       addi $s0, $s0, 4
       bne  $s0, $s1, LOOP
```

- Identify all pipeline stalls per iteration
- Calculate total cycles per iteration
- Propose two forwarding optimizations

Network throughput calculation Exercise: A TCP/IP network link has:

- Bandwidth: 1 Gbps
- Round-trip time: 50 ms
- Maximum segment size: 1460 bytes
- Receiver window size: 64 KB
- Calculate the theoretical maximum throughput
- Determine if the link is receiver-window-limited
- Compute the optimal window size for full bandwidth utilization
- Suggest two methods to improve throughput if RTT increases to 200 ms

Disk I/O scheduling Exercise: A disk has the following characteristics:

- Rotation speed: 7200 RPM
- Average seek time: 5 ms
- Transfer rate: 200 MB/s
- Sector size: 512 bytes

Given the I/O request queue (cylinder numbers):

```
53, 110, 32, 45, 12, 98, 147, 25
```

Current head position is at cylinder 50 moving toward higher numbers.

- Calculate total seek time for FCFS scheduling
- Compute total access time for SCAN (elevator) algorithm
- Determine the optimal scheduling for minimal latency
- Propose a method to reduce rotational latency

Chapter 23

Emerging Trends in GPU Assembly

23.1 Next-Generation Architectures

[Wave32 vs. Wave64] Exercise:

1. Compare the trade-offs between Wave32 and Wave64 execution modes in RDNA3.
2. Write a CUDA or HIP kernel that demonstrates a scenario where Wave32 outperforms Wave64 due to better occupancy.
3. Modify the kernel to use Wave64 and analyze the performance difference using `rocprof` or `nsight`.

[Matrix Multiply on Hopper Tensor Cores] Exercise:

1. Implement a matrix multiplication kernel using Hopper's `wgmma` (Warp Group Matrix Multiply Accumulate) instruction.
2. Benchmark the kernel against a CUDA Core-based implementation for matrices of size `1024x1024`.
3. Identify the minimum matrix size where Tensor Cores provide a speedup over CUDA Cores.

[RDNA3 Dual-Issue Scalar ALU] Exercise:

1. Write an AMD GCN assembly snippet that utilizes RDNA3's dual-issue scalar ALU to compute `a = (x + y)* (x - y)` in one cycle.
2. Measure the IPC improvement over a single-issue implementation using `RGP (Radeon GPU Profiler)`.
3. Explain how the dual-issue capability affects occupancy in compute-bound kernels.

Ray tracing performance analysis Exercise:

1. Implement a simple ray tracer in CUDA using unified memory.
2. Measure the execution time for rendering a scene with 1000 spheres.
3. Compare the performance with and without using unified memory.
4. Analyze the impact of unified memory on ray tracing performance.

Memory access patterns Exercise:

1. Write a CUDA kernel that traces rays through a voxel grid.
2. Use `cudaMemPrefetchAsync` to optimize memory access.
3. Profile the kernel with NVIDIA Nsight to identify memory bottlenecks.
4. Modify the voxel grid traversal to improve cache locality.

Hybrid rendering pipeline Exercise:

1. Create a rendering system that combines rasterization and ray tracing.
2. Use unified memory for shared geometry data between pipelines.
3. Implement a denoiser for the ray traced components.
4. Measure the frame rate impact of unified memory management.

Matrix Multiplication Optimization Exercise: Exercise:

1. Write a CUDA kernel to perform matrix multiplication using tensor cores on an NVIDIA GPU.
2. The kernel should use `wmma::fragment` and `wmma::fill_fragment` for matrix storage.
3. Benchmark the kernel against a naive CUDA implementation for two 1024x1024 FP16 matrices.
4. Explain why the tensor core version performs better or worse in your specific hardware setup.

AI Chip Power Profiling Exercise: Exercise:

1. Profile the power consumption of a TPUv4 chip during inference using Google's Cloud TPU tools.
2. Measure the energy per inference for ResNet-50 at batch sizes 32, 64, and 128.
3. Create a plot showing the relationship between batch size and energy efficiency (inferences/Joule).
4. Identify the batch size that maximizes throughput while staying within 150W power budget.

Quantization Aware Training Exercise: Exercise:

1. Implement a quantization-aware training loop for a 3-layer CNN using TensorFlow's `tfmot` toolkit.
2. Train the model on CIFAR-10 with and without quantization-aware training.
3. Compare the accuracy drop when deploying to an 8-bit integer accelerator.
4. Measure the latency improvement on a Coral Edge TPU using both model versions.

23.2 Future of Low-Level Programming

Code generation for matrix operations Exercise:

1. Use an AI code generation tool to produce Python code that creates a 3x3 identity matrix using `numpy`.
2. Modify the generated code to accept an integer parameter `n` that creates an nxn identity matrix.
3. Profile the execution time of both functions for `n=1000` using `timeit` and compare the results.

Profiling AI-generated sorting algorithms Exercise:

1. Generate Python code for bubble sort using an AI assistant.
2. Generate Python code for merge sort using the same AI assistant.
3. Create a random list of 10,000 integers and profile both sorting functions using `cProfile`.
4. Identify which lines of code consume the most time in each implementation.

Optimizing generated neural network code Exercise:

1. Use an AI tool to generate a simple neural network with one hidden layer using `tensorflow`.
2. Profile the memory usage during training using `memory_profiler`.
3. Modify the batch size parameter and measure its impact on peak memory consumption.
4. Identify the three most memory-intensive operations in the original code.

Memory Allocation Profiler Exercise: Implement a memory profiler in C that tracks dynamic allocations and deallocations.

1. Override `malloc` and `free` using `LD_PRELOAD` or function pointers.
2. Log all allocations to a file with timestamps, sizes, and call sites (use `__FILE__` and `__LINE__`).
3. Detect memory leaks by comparing allocations and deallocations at program exit.
4. Print a summary report showing total bytes allocated, peak usage, and leaked blocks.

Embedded Register Manipulation Exercise: Write a bare-metal C program for an ARM Cortex-M microcontroller to configure GPIO pins.

1. Declare a volatile struct mapping GPIO registers (e.g., `GPIOA_MODER`, `GPIOA_ODR`).
2. Set pin 5 of `GPIOA` as output and pin 3 as input with pull-up resistor.
3. Toggle pin 5 at 1Hz while monitoring pin 3's state.
4. Use bitmask operations (no libraries) to modify registers atomically.

Network Packet Parsing Exercise: Parse an Ethernet frame in C++ without external libraries.

1. Define a packed struct for Ethernet headers (MAC addresses, EtherType).
2. Read a raw packet from a `.pcap` file or network socket.
3. Validate the frame check sequence (FCS) using a CRC-32 algorithm.
4. Extract and print source/destination MACs and payload protocol type.

Timeline of Version Control Systems Exercise:

1. Research and list three version control systems (VCS) developed before Git, including their release years.
2. Explain one key limitation of centralized VCS like Subversion compared to distributed systems like Git.
3. Write a `git log` command to show the last 5 commits with author names and timestamps.

Debugging Tool Comparison Exercise:

1. Compare `gdb` and `valgrind` by listing one primary use case for each tool.
2. Write a C code snippet with a memory leak that `valgrind` would detect.
3. Explain how setting a breakpoint in `gdb` differs from using print statements for debugging.

Build System Migration Exercise:

1. Convert this Makefile rule to a CMake `CMakeLists.txt` equivalent:

```
myapp: main.c utils.c
    gcc -o myapp main.c utils.c
```

2. List two advantages of using CMake over plain Makefiles.
3. Identify one potential issue when migrating a project from `autotools` to CMake.

Chapter 24

Advanced Development Tools

24.1 Assembly Development Tools

Disassembling a simple function Exercise:

1. Given the following x86 assembly snippet, identify the purpose of the function and its return value:

```
mov eax, [esp+4]
add eax, 5
ret
```

2. Rewrite this function in C syntax.
3. Calculate the result when this function is called with argument `input_value=10`.

Identifying buffer overflow vulnerabilities Exercise:

1. Analyze the following C code for potential buffer overflow:

```
void copy_string(char* src) {
    char dest[16];
    strcpy(dest, src);
}
```

2. Determine the minimum length of input that would cause a buffer overflow.
3. Suggest two mitigation techniques for this vulnerability.

Extracting strings from a binary Exercise:

1. List the Unix command to extract all ASCII strings from a binary file named `target_program`.
2. Given the following output snippet, identify which strings might be suspicious:

```
/bin/sh
secret_key_123
/home/user/docs
```

3. Explain how string extraction can help in reverse engineering malware.

Reverse Engineering a Simple Function Exercise: Given the following x86 assembly code snippet:

```
mov eax, [ebp+8]
add eax, [ebp+12]
mov [ebp-4], eax
mov eax, [ebp-4]
```

1. Identify all memory access operations and their types (read/write).

2. Reconstruct the equivalent C function prototype.
3. Determine how many local variables exist in the stack frame.
4. Write the corresponding C code for this assembly snippet.

ARM Disassembly Challenge Exercise: The following ARMv7 assembly implements a loop:

```
mov r3, #0
loop:
ldr r2, [r1, r3, lsl #2]
add r2, r2, #5
str r2, [r0, r3, lsl #2]
add r3, r3, #1
cmp r3, #10
blt loop
```

1. Identify the loop counter register and its increment value.
2. Determine the data type being processed based on the addressing mode.
3. Calculate the total number of memory accesses during execution.
4. Convert this assembly to equivalent C code with a `for` loop.

Stack Frame Analysis Exercise: Given this x86-64 function prologue and epilogue:

```
push rbp
mov rbp, rsp
sub rsp, 32
...
leave
ret
```

1. Calculate the total stack space allocated for local variables.
2. Identify which registers are preserved across function calls.
3. Determine if this function follows the System V AMD64 ABI calling convention.
4. Sketch the stack frame layout showing all relevant components.

Code generation for matrix operations Exercise:

1. Write a Python script using `numpy` to generate C code for matrix multiplication of two 3x3 matrices.
2. The generated code must include proper memory allocation and deallocation for the result matrix.
3. Add a function to print the resulting matrix in row-major format.
4. Include error handling for invalid matrix dimensions in the Python generator.

Automated SQL query builder Exercise:

1. Create a Java program that generates SQL `SELECT` queries based on user input.
2. The generator must support filtering via `WHERE` clauses with `AND`/`OR` conditions.
3. Implement column selection through a configuration file in `JSON` format.
4. Add support for `ORDER BY` and `LIMIT` clauses.

Embedded system register config generator Exercise:

1. Develop a MATLAB script to generate C header files for microcontroller register configurations.

2. The generator must create `#define` macros for register addresses and bit masks.
3. Include bit-field union structures for registers with multiple configuration options.
4. Generate documentation comments in Doxygen format for each register.

Queueing Theory Basics Exercise: Consider a single-server queueing system where:

- Arrivals follow a Poisson process with rate $\lambda = 4$ customers per hour
- Service times are exponentially distributed with mean $\mu^{-1} = 10$ minutes
- The system has infinite queue capacity

Calculate:

1. The server utilization ρ
2. The average number of customers in the system L
3. The average waiting time in the queue W_q
4. The probability that there are exactly 3 customers in the system

Amdahl's Law Application Exercise: A parallel program has the following characteristics:

- Serial fraction $s = 0.15$
- Parallelizable fraction $p = 0.85$
- Current execution time on 4 processors is 18 seconds

Determine:

1. The theoretical speedup when using 8 processors
2. The maximum possible speedup (as processors $\to \infty$)
3. The number of processors needed to achieve a speedup of 5
4. The execution time on 16 processors

Performance Measurement Coding Exercise: Given the following Python function `process_data` in a file `perf_test.py`:

```
import time
import numpy as np

def process_data(size):
    data = np.random.rand(size)
    result = np.zeros(size)
    for i in range(size):
        result[i] = data[i] * 2 + 5
    return result
```

Modify the code to:

1. Add timing measurements using `time.perf_counter()`
2. Print the execution time for processing 1 million elements
3. Vectorize the computation to eliminate the `for` loop
4. Compare the execution times before and after vectorization

Memory Leak Detection Exercise: Exercise:

1. Write a C program that allocates memory using `malloc()` in a loop but never frees it.

2. Compile the program with `gcc -g` and run it under `valgrind --leak-check=full`.
3. Identify the line numbers where memory leaks occur from the Valgrind output.
4. Modify the program to properly free all allocated memory using `free()`.
5. Verify the fix by re-running Valgrind and confirming no leaks are reported.

Stack Overflow Debugging Exercise: Exercise:

1. Create a Python function with a recursive call that lacks a base case, causing a stack overflow.
2. Run the script and observe the `RecursionError` traceback.
3. Rewrite the function using iteration instead of recursion.
4. Add a maximum recursion depth check using `sys.getrecursionlimit()` as a safeguard.
5. Test the modified function with inputs that previously caused stack overflow.

Race Condition Reproduction Exercise: Exercise:

1. Write a Java program with two threads that increment a shared `int counter` variable 1000 times each.
2. Run the program 10 times without synchronization and record the final counter values.
3. Observe inconsistent results and identify the race condition.
4. Fix the issue using `synchronized` blocks or `AtomicInteger`.
5. Verify the fix by running the program 10 times and confirming consistent results (2000).

24.2 Profiling Implementation

Reservoir Sampling Implementation Exercise: Implement a reservoir sampling algorithm in Python to select `k` random items from a stream of unknown length. Follow these steps:

1. Create a function `reservoir_sample(stream, k)` that takes an iterable `stream` and integer `k`.
2. Initialize a reservoir list with the first `k` elements.
3. For each subsequent element at index `i` (starting from `k+1`), generate a random integer `j` between 0 and `i-1`.
4. If `j < k`, replace the `j`-th element in the reservoir with the current element.
5. Return the reservoir list after processing the entire stream.

Stratified Sampling Design Exercise: Design a stratified sampling procedure for a dataset of sensor readings with three distinct operating modes. Complete these tasks:

1. Load the dataset `sensor_data.csv` with columns `timestamp`, `value`, and `mode`.
2. Calculate the proportion of readings in each mode (A, B, C).
3. Determine the sample size per stratum for a total sample of 1000 readings.
4. Implement proportional allocation across strata.
5. Verify the sampled modes match the original distribution within ±2%.

Antithetic Variates Verification Exercise: Verify the variance reduction properties of antithetic variates for Monte Carlo integration. Perform these steps:

1. Define the function `f(x)= exp(-x^2)` to integrate over [0,1].
2. Generate 1000 uniform random numbers `u_i` in [0,1].

3. Create antithetic pairs `(u_i, 1-u_i)`.
4. Compute two estimates: standard Monte Carlo and antithetic variates.
5. Compare the variances of both estimates using `numpy.var`.
6. Confirm the antithetic estimate has lower variance.

Buffer trace analysis Exercise:

1. Capture a memory trace using `perf record -e mem-loads,mem-stores -a sleep 1`.
2. Convert the binary trace to ASCII using `perf script > trace.txt`.
3. Count how many times the variable `buffer_size` appears in `trace.txt` using `grep`.
4. Plot the memory access pattern over time using Python's `matplotlib` with timestamps on the x-axis.

System call visualization Exercise:

1. Run `strace -c -o trace.log ls /tmp` to collect system call statistics.
2. Extract the top 5 most frequent system calls from `trace.log`.
3. Create a bar chart showing call frequencies using `gnuplot`.
4. Identify which system call has the highest average execution time.

Network packet capture Exercise:

1. Capture 100 HTTP packets using `tcpdump -i eth0 -w http.pcap port 80`.
2. Open `http.pcap` in Wireshark and export the packet lengths to CSV.
3. Calculate the mean and standard deviation of packet sizes using Python.
4. Generate a histogram of packet inter-arrival times with 20 bins.

CPU Cache Miss Analysis Exercise:

1. Compile and run the provided C program `cache_miss.c` with optimization level `-O0`.
2. Use `perf stat` to measure the cache-miss rate for the program's main loop.
3. Modify the array access pattern in `cache_miss.c` to improve spatial locality.
4. Re-run the program with `perf stat` and compare the cache-miss rates before and after optimization.

Disk I/O Bottleneck Identification Exercise:

1. Set up a test database using `sqlite3` with the provided `sample_data.sql` script.
2. Execute the query `SELECT * FROM large_table WHERE non_indexed_column > 1000`.
3. Use `iotop` to measure disk I/O during query execution.
4. Create an appropriate index and repeat the measurement.
5. Calculate the percentage reduction in disk I/O operations.

Network Throughput Optimization Exercise:

1. Capture network traffic using `tcpdump` during a file transfer with `scp`.
2. Analyze the packet capture in `wireshark` to identify TCP retransmissions.
3. Adjust the `/etc/sysctl.conf` TCP window size parameters.
4. Repeat the file transfer and capture, comparing throughput using `iperf3` measurements.