Lecture 23:
Vectorized Execution & Course Retrospective

Logistics

- In-class presentations on **Nov 21**

Recap

- Columnar Storage
- Compression
- Compressed Columnar Storage

Lecture Overview

- Vectorized Execution
- Course Retrospective

Vectorized Execution

Limitations of Tuple-at-a-time Processing

- Each tuple incurs the cost of:
  - Function calls between operators.
  - Deserializing, interpreting, and processing the tuples.
  - Doesn't leverage the CPU's ability to process batches of data efficiently.
  - Results in frequent pipeline stalls and cache misses.

Vector-at-a-time Processing

- Process a vector of tuples at a time to reduce overhead of function calls etc.
- Use SIMD (Single Instruction, Multiple Data) instructions
- A single instruction operates on multiple data points simultaneously.

18	40	25	15	
				> 20
0	1	1	1	

SIMD in Query Execution

- **SIMD Use Cases**:
  - Filtering (e.g., select rows within a range).
  - Aggregations (e.g., sum, average).
  - Compression (e.g., decoding bit-packed data).
- **Key SIMD Operations**
  - **Vector Loads**: Load multiple data points.
  - **Masks**: Apply conditions to filter data.
  - **Horizontal Reduction**: Sum vector elements.

Filter timestamps using SIMD

- Load data in **batches** using vld1q_s32.
- Perform **vectorized comparisons** using vcgeq_s32 and vcleq_s32.
- Combine results with a **bitwise AND** using vandq_u32.

```
int32x4_t ts_vec = vld1q_s32(&data.timestamps[i]);  // Load timestamps
uint32x4_t mask = vandq_u32(vcgeq_s32(ts_vec, 1),
                            vcleq_s32(ts_vec, end)); // Mask for range
```

Filter timestamps using SIMD

vld1q_s32	18	40	25	15	
vcgeq_s32					> 20
	0	1	1	1	
vcleq_s32					< 30
	1	0	1	1	
vandq_u32					> 20 AND < 30
	0	0	1	0	

Scalar vs SIMD Execution

- **Scalar Execution**:
  - Processes **one data element per cycle**.
  - Repeated instruction fetch, decode, and execute for each element.
- **SIMD Execution**:
  - Processes **multiple data elements per cycle** by leveraging wide registers.
  - Executes the same operation on an entire vector (batch) with a single instruction.

Benefits of SIMD Execution

- **Instruction-Level Efficiency**
  - Fewer instructions due to vectorized operations.
  - Scalar processing requires **N instructions** for **N data points**.
  - SIMD requires **N / W instructions**, where **W** is the width of the SIMD vector.
- **Cache and Memory Efficiency**
  - Contiguous memory access aligns with cache lines.
  - SIMD operates on **contiguous memory** (columnar layouts align well with SIMD).
  - Cache lines are fully utilized, reducing memory latency.

Benefits of SIMD Execution

- **Minimized Control Overhead**:
  - SIMD minimizes branching by applying the same operation to all elements in a vector.
  - With scalar execution, pipeline stalls if the branch prediction is incorrect.
  - With SIMD execution, masks handle conditional operations, avoiding pipeline stalls.
- **Hardware Support**:
  - Modern CPUs have dedicated SIMD execution units optimized for throughput.

History of SIMD

- **1960s-1970s: Supercomputers and Scientific Computing**
  - SIMD first appeared in systems like **ILLIAC IV** for scientific workloads.
  - Allowed simultaneous operations on multiple data points (e.g., matrix rows).
- **1980s-1990s: Multimedia Processing**
  - MMX (Intel, 1996): Integer operations for multimedia.
  - Example: Increase the brightness of an image's pixels by a constant value.

Pixels	100	120	140	160	
					+ 20
	120	140	160	180	

Modern SIMD in General-Purpose Computing

- **2000s**: Integration in General-Purpose CPUs
  - SIMD became a standard in consumer CPUs.
  - **SSE/AVX (Intel)**: Wide registers for floats and integers.
  - **NEON (ARM)**: Optimized for embedded and mobile devices.
- **2010s-Present**: Acceleration for Analytical Workloads
  - SIMD now powers modern databases and big data systems.
  - Vectorized query execution (e.g., Apache Arrow).

Sensor Data Analysis

- Designed with **SoA (Structure of Arrays)** layout for SIMD.
- Ensures contiguous memory access for efficient vectorized operations.

```
struct SensorData {
    int* timestamps;    // Contiguous array of timestamps
    float* temperatures; // Contiguous array of temperatures

    SensorData(size_t count) {
        timestamps = static_cast<int*>(aligned_alloc(16, sizeof(int) * count));
        temperatures = static_cast<float*>(aligned_alloc(16, sizeof(float) * count));
    }
};
```

Sensor Data Generation

- Create synthetic data with timestamps and temperatures.

```
void generateData(SensorData& data, int count) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::normal_distribution<float> temp_dist(25.0, 5.0);
    std::uniform_int_distribution<int> time_dist(1, 5);

    int timestamp = START_TIMESTAMP;
    for (int i = 0; i < count; ++i) {
        data.timestamps[i] = timestamp;
        data.temperatures[i] = temp_dist(gen);
        timestamp += time_dist(gen);  // Increment timestamp
    }
}
```

SIMD Query

- Task: Calculate the average temperature within a timestamp range

Timestamps	1	2	5	7
Temperature	25.5	26.0	27.2	28.3
> 2	0	1	1	0
< 6	1	1	1	0
> 2 AND < 6	0	1	1	0
Masked Temps	0	26.0	27.2	0
Sum	53.2			

SIMD Query: Loading Data

- Load 4 consecutive timestamps and temperatures into SIMD registers.

```
int32x4_t ts_vec = vld1q_s32(&data.timestamps[i]);  // Load 4 timestamps
float32x4_t temp_vec = vld1q_f32(&data.temperatures[i]);  // Load 4 temperatures
```

SIMD Query: Filtering Timestamps

- Filter timestamps within the query range.
- **Compare for Lower Bound**: vcgeq_s32: Compares timestamps with startTimestamp.
- **Compare for Upper Bound**: vcleq_s32: Compares timestamps with endTimestamp.
- **Combine Results**: vandq_u32: Combines the two masks with a bitwise AND.

```
uint32x4_t in_range_mask = vandq_u32(vcgeq_s32(ts_vec, vdupq_n_s32(startTimestamp)),
                                     vcleq_s32(ts_vec, vdupq_n_s32(endTimestamp)));
```

SIMD Query: Mask Application

- Apply the mask to the temperatures and sum up the valid values.
- vmulq_f32: Multiplies the mask with the temperature vector.
- Keeps valid temperatures, zeros out invalid ones.
- vaddq_f32: Adds the valid temperatures to the running sum.

```
float32x4_t masked_temps = vmulq_f32(temp_vec, vcvtq_f32_u32(in_range_mask));
sum_vec = vaddq_f32(sum_vec, masked_temps);
```

SIMD Query: Final Steps

- **Horizontal Reduction**: Sum up all elements in the SIMD vector.
- **Handle Remaining Scalar Elements**: Process leftover elements not divisible by the SIMD width.

```
float total_sum = vaddvq_f32(sum_vec);
for (int i = count - (count % 4); i < count; ++i) {
    if (data.timestamps[i] >= startTimestamp && data.timestamps[i] <= endTimestamp) {
        total_sum += data.temperatures[i];
    }
}
```

SIMD Instruction Naming

- SIMD instruction names are structured to reflect:
  - **Operation type**: Add, multiply, compare, etc.
  - **Data type**: Integer, floating-point, or specialized formats.
  - **Vector width**: Number of data elements processed.

vaddq_f32	Addition Operation	32-bit floating-point numbers	Quad-word (4 floats)
vcgeq_s32	Compare greater than or equal to	Operates on signed 32-bit integers.	64-bit vector (2 integers)

Evolution of SIMD Widths Over Time

- Wider SIMD registers enable higher parallelism.
- **64-bit SIMD**:
  - Technologies: **MMX (Intel, 1996)**.
  - Operated on **64-bit registers** (e.g., 4 integers).
- **128-bit SIMD**:
  - Technologies: **SSE (Intel)**, **NEON (ARM)**.
  - Supported **4x32-bit floats** or **2x64-bit doubles**.

Evolution of SIMD Widths Over Time

- **256-bit SIMD**:
  - Technologies: **AVX (Intel, 2010)**.
  - Doubled vector width for 8x32-bit floats or 4x64-bit doubles.
- **512-bit SIMD**:
  - Technologies: **AVX-512 (Intel, 2017)**.
  - Supported **16x32-bit floats** or **8x64-bit doubles** in a single operation.

Advanced SIMD: Hash Join Algorithm

- Hash computation and comparison in parallel using SIMD masks.
- Hash table lookups may involve non-contiguous memory accesses

```
// Load probe keys
int32x4_t probe_keys = vld1q_s32(&probe_table.keys[i]);
// Compute hash
int32x4_t hash_vec = vmodq_s32(vmulq_s32(probe_keys, hash_multiplier), hash_table_size);
// Gather hash table values
int32x4_t hash_table_vals = vld1q_s32(&hash_table[hash_vec]);
// Compare keys
uint32x4_t match_mask = vceqq_s32(probe_keys, hash_table_vals);
```

Course Retrospective

Takeaways from the Course

- Let's take a step back and reflect on what you've accomplished.
- Systems programming is challenging.
  - Delving into internals teaches attention to detail.
  - It forces understanding of how things work under the hood.
- Foundational systems knowledge beyond databases.
  - Threading, memory management, and I/O.
  - You now have tools to approach any system-level problem.
  - Reflect on how much you've learned and grown as a programmer.

Big Ideas from the Course

- Database Systems Are Awesome
  - They solve real-world problems elegantly.
  - But they are not magic.
- Abstractions Are Key
  - Elegant abstractions are the "magic" enabling usability and performance.
- Declarativity Rules
  - Declarative query models make complex systems usable.
  - Taken to the extreme -- Google search

Big Ideas from the Course

- Building Systems Is More Than Hacking:
  - It's about design, principles, and reusability.
- Recurring Patterns:
  - Recognizing motifs like parallelism, caching, and transactions.
- Intellectual Contributions:
  - Computer Science is evolving, and you can contribute to its history.

Looking Ahead: What's Next?

- CS 6423 (Advanced Database Implementation)
  - Query Optimization
  - Concurrency Control
  - Logging and Recovery
- Building on this course
  - Deeper insights into **how databases optimize for efficiency**.
  - Expanding your knowledge of **system-level guarantees**.

Looking Ahead: What's Next?

- CS 6423 (Advanced Database Implementation)
  - Logging and Recovery
  - Concurrency Control
  - Query Optimization
- Building on this course
  - Deeper insights into **how databases optimize for efficiency**.
  - Expanding your knowledge of **system-level guarantees**.

Logging and Recovery

- **Example:** A system crash during a transfer:

```
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
```

Logging and Recovery

- Mechanisms to restore the database to a consistent state after crashes.
- **Write-Ahead Logging (WAL)**
  - Log changes before applying them.
- **Checkpointing and Crash Recovery**
  - Periodically save the database state to reduce recovery time.
  - Redo/Undo logs to reconstruct committed transactions and roll back uncommitted ones.
- **ARIES Algorithm**
  - Advanced Recovery Algorithm (Analysis, Redo, Undo).

Concurrency Control

- **Example:** Two users simultaneously trying to update the same row

```
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
```

Concurrency Control

- Ensuring **correctness** and **consistency** when multiple users access the database simultaneously.
- **Locks and Latches**:
  - Types of locks (shared, exclusive); Deadlock detection and prevention.
- **Multi-Version Concurrency Control (MVCC)**
  - Readers don't block writers; writers don't block readers.
- **Isolation Levels**:
  - Read Committed, Repeatable Read, Serializable.
- **Distributed Transactions**:
  - Two-phase commit (2PC), distributed locking.

Query Optimization

- **Example:** Push filters before the join. Use an indexed join if available.

```
SELECT *
FROM orders
JOIN customers
ON orders.customer_id = customers.id
WHERE customers.city = 'Atlanta';
```

Query Optimization

- Selecting the **best execution plan** for a query.
- Goal: Minimize **execution cost** (e.g., time, memory, I/O).
- **Execution Plans**
  - Logical vs. physical plans.
- **Cost Models**
  - Estimating costs for different plans.
- **Heuristics and Rules**
  - Simplifying query trees and reordering joins.
- **Advanced Techniques**
  - **Dynamic Programming** (e.g., System R algorithm).
  - **Cardinality Estimation**

Feedback and Project Presentations

- Please share your feedback via CIOS
- +1% extra credit for entire class if we get 80%+ participation
- In-class presentations on **Nov 21**
- Tentatively prepare a 5-minute presentation

Conclusion

- Vectorized Execution
- Course Retrospective