Data Processing on Modern Hardware

Tutorial 5

Ferdinand Gruber

Michalis Georgoulakis

Assignment 3 - Hardware-Optimized Hash Joins

Hardware-Optimized Hash Joins

Homework Submission

- Deadline: **28/05** - you have one more week week
- You are allowed to use libraries (e.g., for the hash table)

Submission Instructions

- First fork the assignment repository.
- Then, in your forked repository, add:
  - Code that implements the assignment
  - A 1-page report with your findings

Report Expectation

Part 1: Implement the classic in-memory hash join (`hash_join`).

- **Benchmark** it: measure *time per tuple* as you scale `|R|` and `|S|` from L1-sized up to well beyond LLC.
- **Discuss** cache effects (e.g. why performance degrades once you overflow L2/L3).

Part 2: Implement each of your three partitioned-join variants (`naive`, `softwareManaged`, `multiPass`).

- **Benchmark** them across the same size sweep.
- **Analyze** which variant wins for small, medium, and large relations, and use perf to find out *why* (tie back to histogram cost, buffer management, write-combining, TLB misses, etc.).

Part 3: Implement the full radix join (`radix_join`) that combines one of your partitioners with a per-bucket hash join.

- **Benchmark & compare** its *per-tuple* performance against the baseline from Part 1 and the best partitioned variant from Part 2.
- **Explain** how it fuses partitioning+join to give overall improvement (or where it still falls short).

Non-Temporal Stores → Key Requirements

What the Hardware Needs	What Your Code Must Do
• 256-bit (AVX2) streams require 32-byte alignment • 512-bit (AVX-512) streams require 64-byte alignment	1. **Align your buffers** - Base output array: 64-byte aligned (aligned_allocator, posix_memalign, or pointer bump) - Each `SoftwareManagedBuffer`: `alignas(64)` - `static_assert(sizeof(Buffer) >= stream_width)`
• Streams always write exactly one full cache line • No cache-line fetch ahead of write (bypass L1)	2. **Pad each partition** - Round up `histogram[i]` to a multiple of `tuplesPerCL` so every streamed write is exactly 64 bytes - Compute `startPositions[i]` from these **padded** sizes
• Don’t inadvertently read or hash the padding	3. **Remember the real counts** - Return a second vector of “real” tuple counts per partition - In the build/probe phase only touch `[start, start + real_count)`

Non-Temporal Stores → Padding & Alignment

1. **Alignment** = where the buffer *starts* in memory
   - You need the buffer's base address (and each little `SoftwareManagedBuffer`) to lie on a cache-line boundary (e.g. a 64-byte multiple) so that a VMOVDQA store won't fault or fall back to a slow path.
   - That's solved by using `alignas(64)`, or an aligned allocator (`posix_memalign`/`aligned_alloc`).
2. **Padding** = offset between *computed* address and required alignment for a partition
   - Once you've got a 64-byte-aligned base, you must ensure that every *write* you do with a streaming store covers exactly one full cache line—no half-lines.
   - You accomplish this by rounding each partition's tuple count *up* to the next multiple of `tuplesPerCL` (so its byte length is a multiple of 64).
   - The extra "padded" slots are never *read*—you track the real tuple count separately.

In short:

- **Alignment** makes the *address* legal for cache-line writes.
- **Padding** makes the *size* legal for whole-line writes.
  You need *both* to safely and efficiently use non-temporal stores.

Q&A

Q2. Compile Flags for non-temporal write avx instructions

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -O3 -march=native")

Q3. Correct Syntax for temporal writes.

_mm256_stream_si256(__m256i p, __m256i a);*

_mm512_stream_si512(__m512i p, __m512i a);*

Q4. Where can i see if i have passed an assignment or not?

Questions?