High Peformance Computing: Research Motivation

Modern CNN architectures frequently store and process data in flattened 1D arrays, which places heavy demands on memory bandwidth and cache efficiency. While I've already explored optimizations in threading and parallel processing, I now want to focus more deeply on SIMD access patterns and their interaction with caching and prefetching.

At a low level, performance depends not only on computation but also on how data flows through the memory hierarchy. Sequential versus non-sequential access, cache line alignment, and compiler vectorization strategies all play a role in how effectively hardware prefetchers detect and optimize memory usage. I also want to explore experimental information (such as data types), and how I could/if I can leverage them for CNNs.

This research aims to clarify:

A. What optimizations the compiler performs automatically versus what requires manual intervention, also look more deeply into OpenMP and threading.
B. How prefetching behavior is influenced by access patterns, algorithms, and compilation flags.
C. Whether forward sequential array traversal is more cache- and prefetch-friendly than backward traversal on modern CPUs.
D. What are brainfloats and can we use them?

From A-C: I hope to understand: How SIMD instructions and multithreading interact with caches and prefetchers, and how to exploit them for efficient data processing.

By investigating these questions, I hope to build a clearer mental model of how compilers, caches, prefetchers, SIMD, and threading all work together — and where manual optimization can provide the most benefit.

Compilation Research:

AVX-512 Library Reference: https://docs.oracle.com/cd/E37838_01/html/E61064/gsesq.html

Compilers

Conditions for Variables and Register Trimming

- **Compiler can remove variables:** When code is used briefly, or not at all.
- **Compiler will store as registers:** Loop variables are often stored as temp registers (typically up to 16 general purpose?).
- **Compiler cannot remove variables:** When you grab its address, give keyword volatile, or print its value is passed as pointer and register pressure.

Loop Optimizations

```
for(int i=n; i > 0; i--);  // Better bc, zero register, easier to optimize, less comparison.
for(int i=n; i != 0; i--); // Optimal micro-optimization.
```

Rule of Thumb

Focus your optimizations on areas where the compiler cannot automatically optimize. You can also improve performance by aligning your data to encourage vectorization and by providing hints to the compiler, such as `#pragma unroll` for loop unrolling.

Cases That Compiler Does Not Optimize

When Compiler Cannot Prove Safety

```
// Compiler might not vectorize this (aliasing uncertainty)
void add_arrays(double *a, double *b, double *c, int n) {
    for (int i = 0; i < n; i++) {
        a[i] = b[i] + c[i];
    }
}

// Manual fix: use restrict
void add_arrays(double *restrict a, double *restrict b, double *restrict c, int n) {
    #pragma omp simd
    for (int i = 0; i < n; i++) {
        a[i] = b[i] + c[i];
    }
}
```

Complex Data Usage

```
// Compiler might not optimize this well
for (int i = 0; i < n; i++) {
    result += data[indices[i]] * weights[i];
}

// Manual: prefetch hints, blocking
for (int i = 0; i < n; i += 4) {
    __builtin_prefetch(&data[indices[i+8]]);
    // ... process 4 elements
}
```

When You Know Something the Compiler Doesn't

```
// You know n is always multiple of 8
#pragma GCC unroll 8
for (int i = 0; i < n; i++) {
    // ...
}
```

Manual Alignment for SIMD

```
double *array = aligned_alloc(64, n * sizeof(double));
#pragma GCC ivdep
for (int i = 0; i < n; i++) {
    array[i] = ...;
}
```

Optimization Flags

- **-O1:** Simple optimization: remove dead code, and improve loops.
- **-O2:** Loop unrolling, instruction scheduling, inlining functions.
- **-O3:** +SIMD, function cloning, aggressive loop transformations.

- **-Ofast:** Even more aggressive. Ignores strict standards compliance. Will have different floating point behavior (e.g., may assume no NaNs or infinities in floating-point math).
- **--march=native:** Enables all instruction sets supported by your current CPU (AVX2, AVX-512, SSE4, etc.).
- **-mfma:** Enable Fused Multiply-Add (FMA) instructions, which compute `a * b + c` in one step with higher precision and lower latency. Common in linear algebra, ML, and scientific code.
- **-ffast-math:** Aggressively optimize floating-point math by relaxing strict IEEE rules. ENABLES: `-fno-trapping-math`, `-fno-math-errno`, `-funsafe-math-optimizations`
- **-fstrict-aliasing:** Assumes that pointers of different types do not alias the same memory.
- **-fopenmp:** Enables OpenMP pragmas (e.g., `#pragma omp parallel for`) for multithreading. Without this flag, OpenMP directives are ignored.

OpenMP Research and Comparisons

Pthreads allow explicit control of threads, including stack size and scheduling. They are best when threads must be reused or customized.

OpenMP provides a higher-level approach using compiler pragmas, ideal for simple parallelism where thread management is handled automatically.

Vectorized operations often follow the pattern load → compute → store across multiple elements. OpenMP maps these efficiently, while Pthreads enable more specialized or persistent threading models.

Cache & Prefetch Behavior

What Affects the Prefetcher?

Essentially, the prefetcher gets weak, or non-existent, if you make stuff that the CPU can't predict, thus you lose out on prefetching the next cache line. Here are some conditions:

1. A memory access becomes "irregular" when:
2. Stride ≠ 1 (or not aligned to cache lines)
3. Access depends on data (indirect/pointer-chasing)
4. Cross-iteration dependencies serialize accesses
5. Branching skips memory writes/reads unpredictably
6. Multiple threads scatter writes across memory

Things That Make Prefetcher Worse

Strided or Non-Unit Access

If your loop accesses memory with a stride that isn't 1 (or the size of the data type), addresses aren't sequential by 7 * sizeof(int) each iteration → prefetcher sees gaps. Small strides (like 2 or 4) can sometimes still be prefetched, but large or prime-number strides usually break it.

```
for (int i = 0; i < N; i++)
    sum += data[i*7];  // stride = 7 ints. Memory addresses jump
```

Indirect or Pointer-Chasing Accesses

Access patterns where you go through another array or pointer. Now the next memory address depends on the content of `indices[i]`. Hardware can't predict which cache line will be needed next → prefetch fails.

```
int idx = indices[i];
sum += data[idx];
```

Cross-Iteration Dependencies

When each iteration's memory location depends on previous iterations. Compiler may serialize the loop → memory accesses aren't contiguous anymore. Prefetchers rely on contiguous/sequential patterns, so efficiency drops.

```
data[i+1] = data[i] + something;
```

Loops with Branches Affecting Memory

Conditional access inside a loop can break predictability. If `condition[i]` is unpredictable, some iterations may skip a memory access → prefetcher sees "holes" in the stream.

```
if (condition[i])
    data[i] += 1;
```

Multithreading Interleaving

If multiple threads write to different parts of memory in non-contiguous patterns, prefetching can be less effective because the hardware sees scattered access patterns.

General Conclusion on Caches/Prefetchers

Conclusion on Caches/Prefetchers: Caches are direction-agnostic, but prefetchers are typically better optimized for forward sequential access due to its prevalence in code. Backward access is still well-supported on modern CPUs, but prefetching may be slightly less efficient (e.g., higher latency to detect the pattern or fewer prefetched lines).

SIMD's rely heavily on prefetching! They do best under contignous memory accesses when vectorizing!

Conclusion... apparently to AI: Evidence from Hardware Documentation: Intel's optimization manuals (e.g., Intel 64 and IA-32 Architectures Optimization Reference Manual) note that prefetchers like the L2 streaming prefetcher can handle both directions but are tuned for forward access. AMD's Zen documentation similarly indicates that prefetchers adapt to descending patterns, though forward access is the primary optimization target.

Experimentations

For very large arrays (e.g., ~471M doubles), forward access is consistently faster, especially with vectorization, since prefetching and SIMD optimizations are designed for it. This effect becomes most noticeable at larger sizes where prefetch algorithms are more effective.

For mid-sized arrays, in non-vectorized code, backward access can trend faster, though large strides degrade performance. Then for small, cache-resident arrays, access direction makes little difference.

So IG: Use forward access for vectorized code. For non-vectorized code, backward access may help on mid-sized arrays. On small arrays, it's negligible.

Bfloat

The **bfloat16 (brain floating point)** [1][2] format is a computer number format occupying 16 bits in memory. It represents a wide dynamic range of numeric values by using a floating radix point. This format is a shortened (16-bit) version of the 32-bit IEEE 754 single-precision floating-point format (binary32), with the intent of accelerating machine learning and near-sensor computing.

Accumulation

When I say a vectorized operation *"accumulates in float"* in the context of bfloat16 (bf16) operations with hardware like **AVX-512BF16**, I mean: The operation (e.g., a dot product or matrix multiplication) performs computations using **bfloat16 inputs** directly. The result of the operation (or intermediate results, like sums in a dot product) is stored and managed in **32-bit float (fp32)** format to maintain numerical stability and precision.

General Notes

In all languages and abstraction levels, if you don't have special hardware, **bfloat will be simulated**, often accumulating into `f32`.

C and C++: If hardware supports it, you can use some bfloat vectorized operations, but they will still accumulate.

C: Does not have a native bfloat type. You would need custom assembly for general arithmetic. You must explicitly know when to use the right intrinsics for bfloats that accumulate, or when conversions are required.

C++: Has a native bfloat type and abstracts away some of the details, giving you less control/knowledge of when/if conversion happens. However, it can perform general arithmetic and potentially enable auto-vectorization. But the **low-level behavior** still mirrors C's when using intrinsics.

GPU and OpenBLAS: Also suffer from the same issues. They may implement optimized workarounds.

PyTorch, NumPy: Also face similar problems. However, they most likely optimize for ML algorithms.

My Thoughts

- Most likely, conversions occur at the **register level**, so you're probably not adding new things to the stack. To optimize, you may need to **pad vectors** to avoid unnecessary conversions. You could analyze load/store patterns better. Ideally, you want all your bfloats to remain in vectors.

Practical takeaway:

- To use bfloats effectively in C or C++: Ensure caching benefits outweigh the extra instructions. Make sure the time spent converting is still less than fetching from higher levels of cached memory.

C++ Thoughts: General arithmetic support helps handle leftover operations that don't fit vectorized shapes. Yet... Idk. As a higher-level language, it may be less optimal than carefully tuned C.