Core C++ 2025

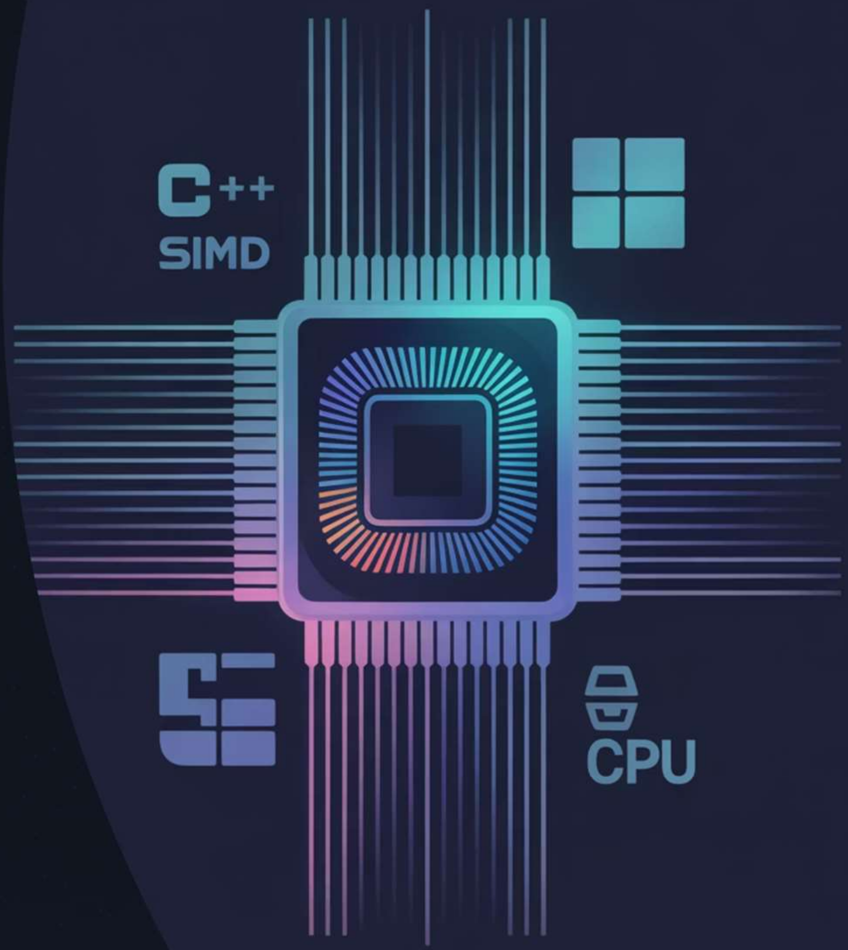
19 Oct. 2025 :: Tel-Aviv

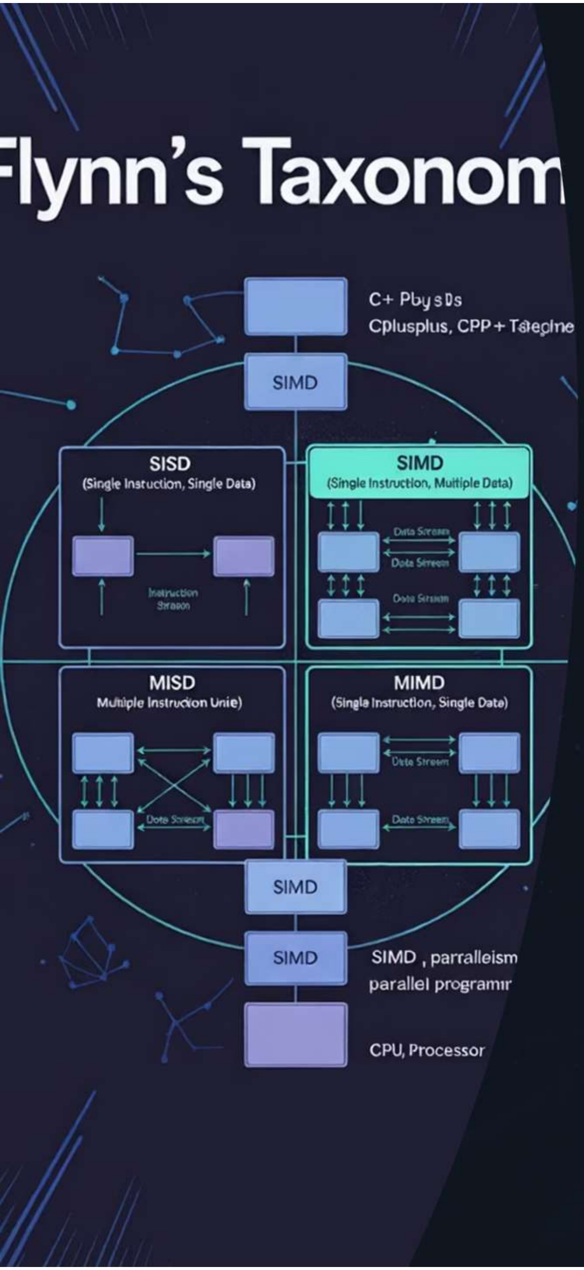
Parallel Paths: From SIMD Lanes to Memory Highways in C++

Eran Gilad

Serial code can be parallel too

- Better utilize available hardware
- Complement parallel and distributed programming
- Fewer algorithmic challenges
- More low-level issues





SIMD: Single Instruction, Multiple Data

SIMD enables processing multiple data elements within a single instruction

Intel Architecture

- MMX - multimedia extensions
- SSE - streaming SIMD extensions
- AVX - advanced vector extensions

AMD Implementation

- 3DNow! - 3D graphics acceleration
- SSE - compatible with Intel
- AVX - advanced vector support

ARM

- NEON - ARM's SIMD engine
- SVE - scalable vector extension

RISC-V

- RVV - RISC-V vector extension

Use cases: image processing, ML vector operations, data analytics, etc.

Basic SIMD Processing Flow

01

Load Data from memory

A SIMD register can hold various combinations of element numbers and types

02

Configure Lanes if needed

Set up masks or shuffling

03

Execute Operation

Perform the operation across all lanes simultaneously

04

Extract Results

Store in memory, reduce to single value, read some lanes, obtain bitmasks, etc.

Common SIMD Operations

Data Movement

Load, store, gather, scatter operations. Broadcast values across lanes, extract/insert individual elements for flexible data manipulation.

Arithmetic Operations

Element-wise unary and binary operations, fused multiply-add for improved precision and performance in mathematical computations.

Comparisons & Masks

Conditional operations using masks, enabling selective processing of elements based on comparison results.

Reductions & Horizontal

Sum, min, max operations across lanes. Horizontal operations combine elements within a single register.

Data Reorganization

Shuffle, permute, and blend operations for rearranging data elements within and between registers.

Specialized Operations

Type casting between formats, cryptographic functions, matrix operations, and bit manipulation tricks.

Talk Outline

Part 1: SIMD Deep Dive

- Use case: find max integer in array
- Raw assembly implementation
- Compiler intrinsics
- C++26 std::simd
- Auto-parallelization techniques

Part 2: Memory-Level Parallelism

- MLP vs. ILP
- Use case: aggregate unstructured data
- Bulk API design

A Word of Caution

Remember Amdahl's Law

$$Speedup = \frac{1}{seq + \frac{par}{M}}$$

Constant Factor Optimization

Speedup is limited.

Still matters IRL.

Complexity Trade-off

With great power comes great complexity!

Use case: Find Max

```
int result = *std::max_element(arr.begin(), arr.end());
```

Or, more explicitely:

```
int result = arr[0];
for (int v : arr) {
    if (result < v) result = v;
}
```

Key Characteristics:

- A single pass over the array
- One conditional branch per value to check input end
- One conditional branch per value to check max
- Array size doesn't matter

Vectorized find max - hierarchical reduction

Using SIMD ops that get the max of K values:

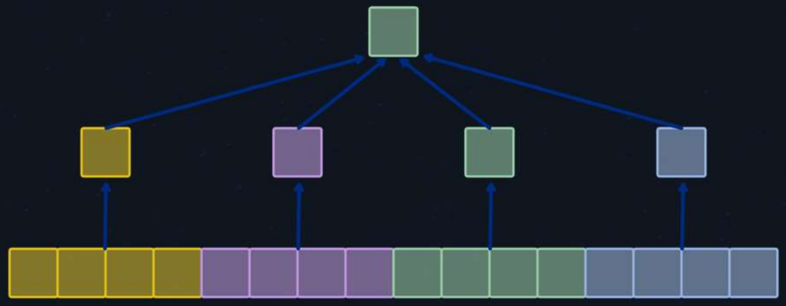
- Find a local max in each **K** elements, overall **N/K** ops.
- Find the max of each **K** local maximums, overall **(N/K)/K** ops.
- And so on, until we have a single global maximum.

But:

- Need to temporarily store local reductions
- Need to do *log_K(N)* rounds of reductions
- Need a horizontal max instruction...

The total number of SIMD operations:

$$\frac{N}{K} + \frac{N}{K^2} + \frac{N}{K^3} + \cdots + 1 \approx \frac{N-1}{K-1}$$



Vectorized reduction - lane-wise with final reduction

Phase 1: lane-wise maximum

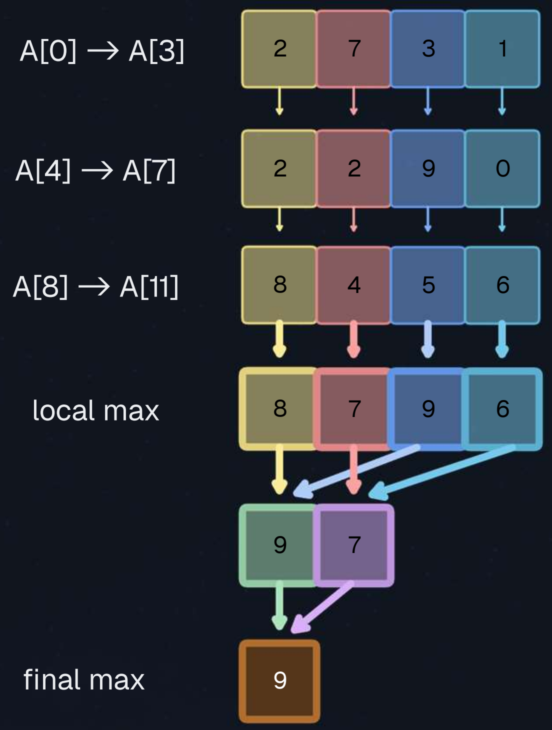
- Given SIMD instructions with K lanes, find the max value in each lane.
- Total number of operations: *N/K*

Phase 2: final reduction

- A single horizontal max (if supported).
- Or, a tree of |*log_2(K)* lane-wise reductions.

Advantages:

- Fewer overall steps (with larger inputs)
- A single pass over the array
- No need to maintain intermediate results
- Pair-wise (lane) max significantly faster than horizontal max
- Horizontal max not supported on AVX (x86)



A[0] → A[3] 2 7 3 1

A[4] → A[7] 2 2 9 0

A[8] → A[11] 8 4 5 6

local max 8 7 9 6

9 7

final max 9

Raw Assembly Implementation

Loop over an array and compute lane-wise max:

```
; rax points to the next elements
; rcx is the array end
; zmm0 holds the first elements

.L1:
vpmaxsq zmm0, zmm0, ZMMWORD PTR [rax]
add     rax, 64
cmp     rax, rcx
jne     .L1
```

vpmaxsq

Vector instruction, **P**acked, **S**igned, **Q**uad word

zmm0

AVX-512 register of size 64 bytes.

⚠ Final reduction and edge cases omitted

Intrinsics Implementation

```
__m512i vmax = _mm512_loadu_si512((__m512i*)data);
for (size_t i = 8; i < arrSize; i += 8)
{
    __m512i v = _mm512_loadu_si512((__m512i*)(data + i));
    vmax = _mm512_max_epi64(vmax, v);
}
```

__m512i

64-byte register of ints

_mm512_loadu_si512

Load **U**naligned **S**calar **I**nts

_mm512_max_epi64

Lane-wise max of 64B ints

⚠ Nicer than assembly, but still very low-level and architecture-specific.

C++26 std::simd Implementation

```
simd<int64_t> max_vec{data, element_aligned}; // data is the input array (int64_t*)

for (size_t i = simd<int64_t>::size(); i < dataSize; i += simd<int64_t>::size())
{
    simd<int64_t> vec{&data[i], element_aligned}; // load next vector
    max_vec = std::max(max_vec, vec);             // lane-wise max
}
return hmax(max_vec); // horizontal max reduction
```

std::simd<int64_t>

SIMD data type, portable width

{data, element_aligned}

Load from partially aligned data

std::max, std::hmax

Vector overload, reduction algo

⚠ Higher-level SIMD abstraction, same SIMD-aware programming model.

Auto-parallelized Implementation

```
int64_t max = data[0];

for (size_t i = 0; i < dataSize; ++i)
{
    if (max < data[i])
        max = data[i];
}
return max;
```

✓ The compiler does it all:

- Vectorize data loads
- Vectorize lane-wise max
- Vectorize final reduction
- Handle edge cases
- for_each and reduce also work

⊗ But… `max_element(execution::unseq, …)` isn't parallelized, and so are other patterns…

Auto-parallelized Limitations

```
for (size_t i = 0; i < size; ++i)
{
    v1[i] += v2[i];
}
```

⊗ Vecorization breaks if v2 = v1 - 1

The compiler can add run time checks for that

⊗ Float ops ordering and signed int overflow might limit or prevent vectorization

This is a low-hanging fruit. Other auto-parallelization issues exist (complex patterns, too many branches, dynamic data structures, infeasible transformation, etc.

Auto-parallelized Hints

```
#pragma GCC ivdep
for (size_t i = 0; i < size; ++i) {
    v1[i] = v1[i] + v2[i];
}
```

Or `#pragma clang loop vectorize(enable)`

Or `#pragma omp simd`

```
int64 getMax(
    int64_t* __restrict v1,
    int64_t* __restrict v2)
{...}
```

No `restrict` keyword in C++ 😲

`__restrict` is a common compiler extension

⚠ Back to non-portable world. Here, macros are your besties 🫂

Conditionally Using AVX-512

Not all CPUs support AVX-512.

`lscpu | grep -wo "avx512"`, or:

```
bool useAvx512() { return __builtin_cpu_supports("avx512f") != 0; }
```

Shared:

```
inline __attribute__((always_inline)) int64_t findMaxShared(...) { ... }
```

AVX-2:

```
__attribute__((target("avx2"))) int64_t findMaxAvx2(...) { return findMaxShared(arr); }
```

AVX-512:

```
__attribute__((target("avx512f"))) int64_t findMaxAvx512(...) { return findMaxShared(arr); }
```

Benchmarks

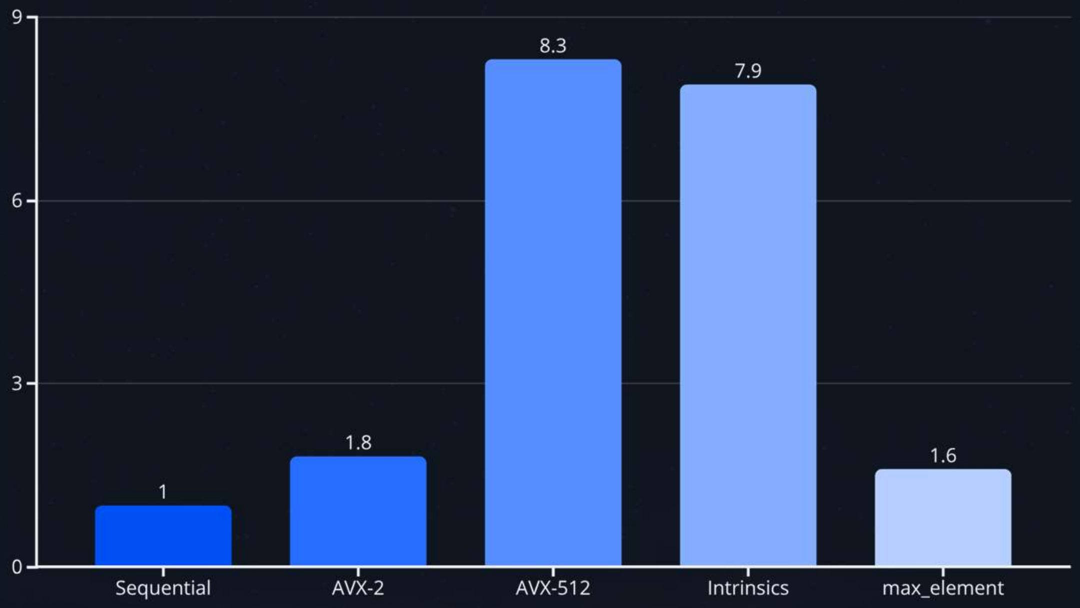
Based on google benchmark:

- 1K array of int64_t
- AVX-2 - 4 lanes, no max op
- std::max_element better optimized

Also:

- Different instruction counts
- Memory accesses, pipelineing, etc.
- YMMV

Speedup vs. sequential code



Detecting auto-vectorization

Ask the compiler to report vectorization:

1. Add the compiler flag `-fopt-info-vec=vec.log`
2. Look for messages a-la `loop vectorized using 64 byte vectors` in the log

Review disassembly:

1. Run after build`objdump -d -C my_app` (or use a debugger)
2. Look for `xmm`, `ymm`, or `zmm` registers
3. Existance indicates some use, not how optimal!

Benchmark:

1. Need a baseline
2. Linear scalability not guaranteed
3. The result is what we actually care about

SIMD Implementation Strategies: A Recap

Raw Control

- Prefer intrinsics over raw assembly
- Max control but min portability

Portable Abstraction

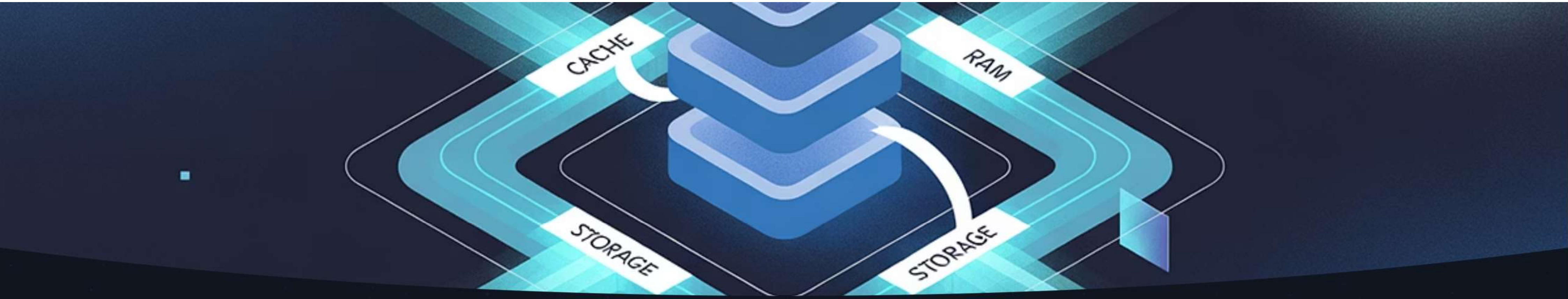
- C++26's `std::simd` nicer and portable
- Still requires "SIMD mindset"

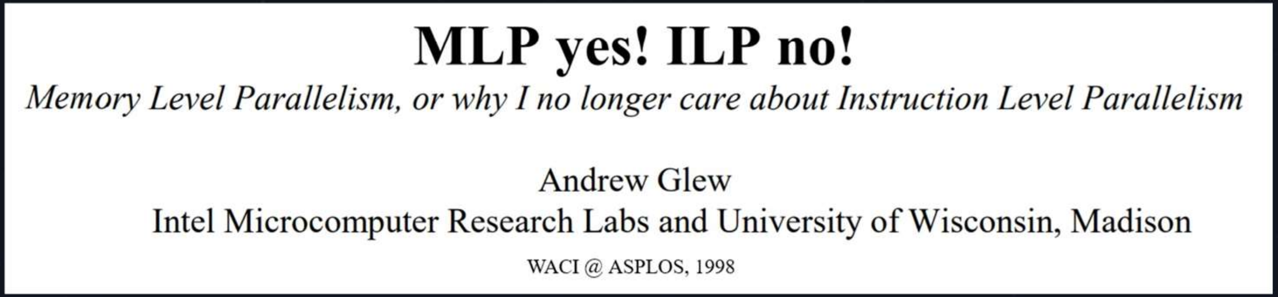
Automatic Vectorization

- Mostly portable and mostly declarative
- Unstable and unclear outcome

Beyond CPU Crunch

- SIMD accelerates compute-intensive code
- What about memory-bound work?





MLP yes! ILP no!

Memory Level Parallelism, or why I no longer care about Instruction Level Parallelism

Andrew Glew
Intel Microcomputer Research Labs and University of Wisconsin, Madison

WACI @ ASPLOS, 1998

- SIMD is very useful with packed data in L1 cache
- SIMD helps less with unstructured data or code

- Frequent cache misses matter a lot
- We have to *hide* these cache misses

Out-of-Order Executionand Memory

Exploiting Parallelism

- OOOE runs independent ops in parallel
- Memory accesses too
- CPUs *speculate* to increase OOOE

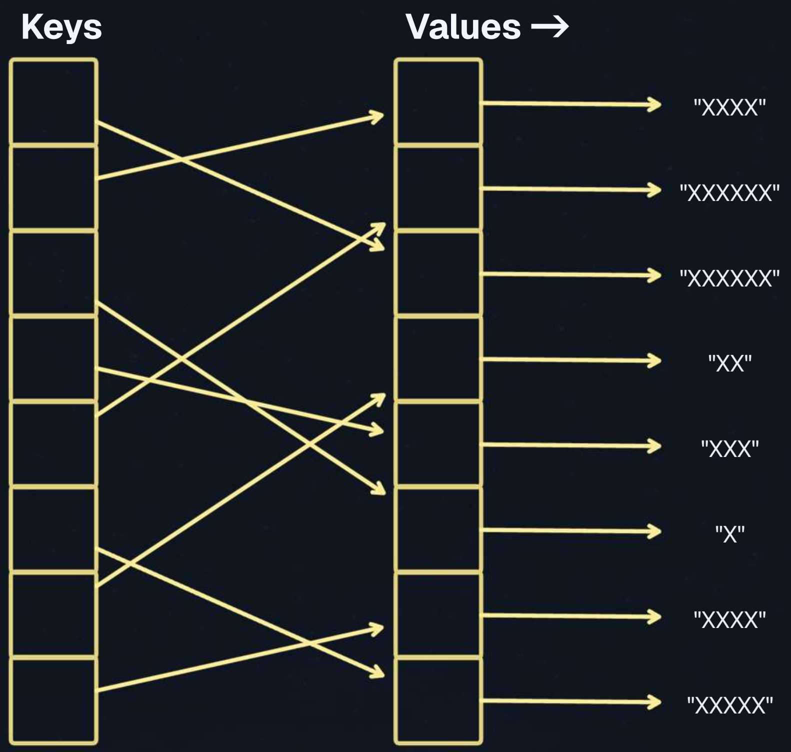
Managing Dependencies

- Memory accesses harder to speculate
- Try to algorithmically limit dependencies
- Bulk processing can help!

Use case - fully sequential

```
size_t total = 0;
for (auto k : keys)
{
    total +=
        strlen( values[k].c_str() );
}
```

- 64K keys
- 1M values, up to 1KB each
- `_mm_clflush` all values before each run

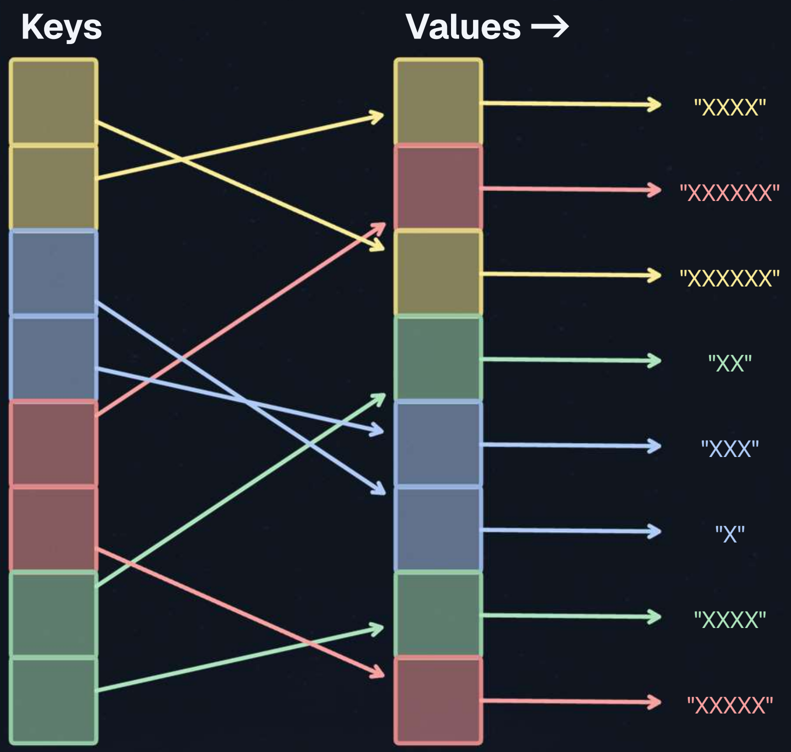


Bulk processing

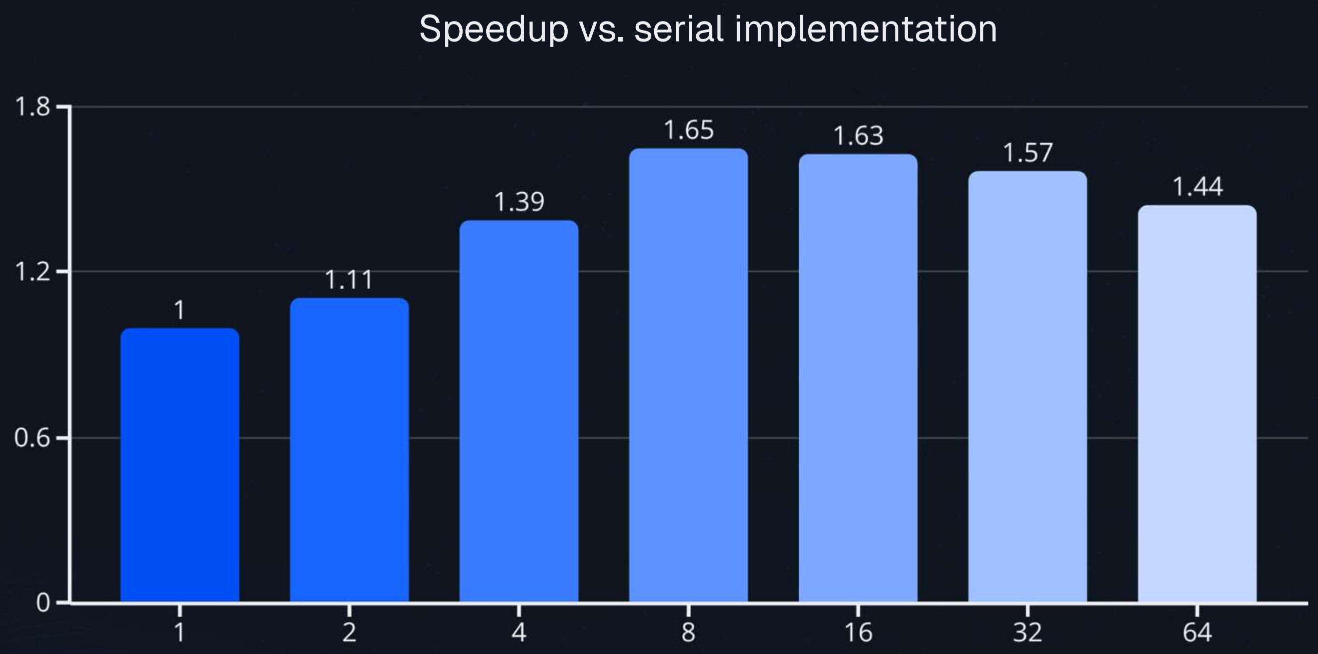
Each step of the processing performed over a bulk of inputs:

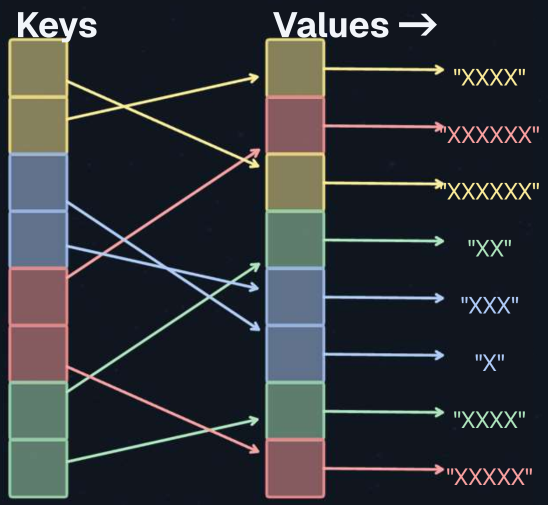
```
_mm_prefetch( &table[ keys[0] ], _MM_HINT_T0 );
_mm_prefetch( &table[ keys[1] ], _MM_HINT_T0 );

_mm_prefetch( table[ keys[0] ].c_str(), _MM_HINT_T0 );
_mm_prefetch( table[ keys[1] ].c_str(), _MM_HINT_T0 );

sum0 += strlen( table[ keys[0] ].c_str() );
sum1 += strlen( table[ keys[1] ].c_str() );
...
total = sum0 + sum1
```

- Keys access is sequential
- Prefetching parallelizes memory accesses
- No dependencies until final reduction



Benchmarks


Speedup vs. serial implementation



- Bulk gains matchs MLP limit - ~8-16. Might be accidental...
- Different params → different results. YMMV.

Bulk APIs

Enable compiler optimizations and HW utilization beyond algorithm level

Compile-time Bulk Size

- Enables loop unrolling
- Eliminating conditional branches
- Allow aggressive SIMD optimizations

Function Call Reduction

- Amortizes virtual and regular function calls

Application Constraints

- Input bulk must be available
- Complex bulk ops require complex code
- Out-of-order execution is sometimes good enough

Summary

SIMD

maximize processing power in compute-intensive loops

Memory Parallelism

Hide memory latencies and improve throughput.

Combining SIMD and MLP

Not always synergistic - address different bottlenecks

Bulk APIs

Structured approach enabling both SIMD and MLP

Talk source code: https://github.com/erangi/cpu-par-talk

Thank you! Questions?