Technische
Universität
Braunschweig

Institut für Betriebssysteme
und Rechnerverbund
Algorithmik

Algorithm Engineering
Lecture 2: CPU Architecture 2

Room for the Tutorial

Room for the Tutorial

Hopefully:

- no more standing in the lecture/tutorial,
- no more sitting on the window sills,
- no more PK 3.3; we have the SN 23.2 (70 official seats) instead.

Optimization Hierarchy

Optimization Hierarchy

Correctness comes first! Then:

- choose the right algorithm, then
- choose the right data structures and memory layout, then
- measure performance, help branch prediction, etc.

Today

Today

Topics today:

Today

Topics today:

- Finishing caching/memory

Today

Topics today:

- Finishing caching/memory
- Compiler optimizations & single instruction, multiple data (SIMD)

Today

Topics today:

- Finishing caching/memory
- Compiler optimizations & single instruction, multiple data (SIMD)
- Parallelization

Caching: SoA vs AoS

Caching: SoA vs AoS

Striding:

Caching: SoA vs AoS

Striding:

- Iterating an array of structures skips past unused fields

Caching: SoA vs AoS

Striding:

- Iterating an array of structures skips past unused fields
- Unless we use most of the parts of the structure

Caching: SoA vs AoS

Striding:

- Iterating an array of structures skips past unused fields
- Unless we use most of the parts of the structure
- Fundamental issue in many situations:
  - Array of structures (AoS) is much cleaner from a software engineering perspective
  - Structure of arrays (SoA) can be more performant for several reasons

```cpp
struct Particle {
    float pos[2];
    float speed[2];
    float color[4];
    float shinyness;
    float rotation;
    float age;
    float springiness;
    float density;
    float area;
    float thickness;
    // ...
};
std::vector<Particle> particles;
```

```cpp
struct Particles {
    std::vector<std::array<float,2>> positions;
    std::vector<std::array<float,2>> speeds;
    std::vector<std::array<float,4>> colors;
    std::vector<float> shinyness;
    std::vector<float> rotation;
    std::vector<float> age;
    std::vector<float> springiness;
    std::vector<float> density;
    std::vector<float> area;
    std::vector<float> thickness;
    // ...
};
Particles particles;
```

SoA vs. AoS: Experiment

```cpp
struct Data {
    float touched[2];
    char untouched[N];
};
std::vector<Data> aos;
struct SoA {
    std::vector<std::array<float,2>> touched;
    std::vector<std::array<char,N>> untouched;
};
SoA soa;
```

SoA vs. AoS: Experiment

```cpp
struct Data {
    float touched[2];
    char untouched[N];
};
std::vector<Data> aos;
struct SoA {
    std::vector<std::array<float,2>> touched;
    std::vector<std::array<char,N>> untouched;
};
SoA soa;
```

Each element:

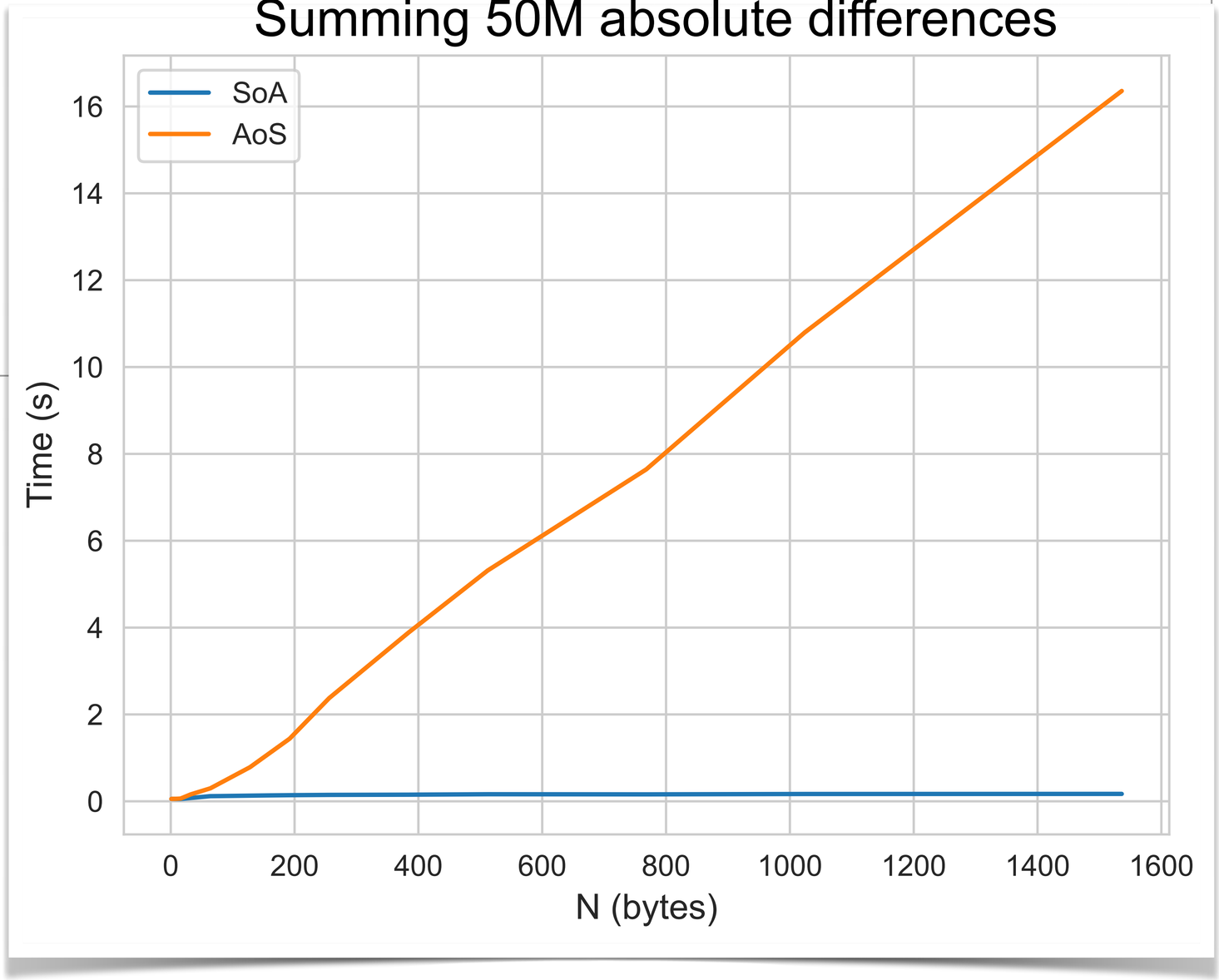
```cpp
s += (std::abs)(touched[0] - touched[1]);
```

SoA vs. AoS: Experiment

```cpp
struct Data {
    float touched[2];
    char untouched[N];
};
std::vector<Data> aos;
struct SoA {
    std::vector<std::array<float,2>> touched;
    std::vector<std::array<char,N>> untouched;
};
SoA soa;
```

Each element:

```cpp
s += (std::abs)(touched[0] - touched[1]);
```



Cost of Memory Allocations

Cost of Memory Allocations

Another pitfall of linked/pointer-heavy data structures: Allocation cost

Cost of Memory Allocations

Another pitfall of linked/pointer-heavy data structures: Allocation cost

- Single malloc/free pair: reasonably cheap (50-500 cycles, more for fresh memory)

Cost of Memory Allocations

Another pitfall of linked/pointer-heavy data structures: Allocation cost

- Single malloc/free pair: reasonably cheap (50-500 cycles, more for fresh memory)
- But much more expensive than other common operations

Cost of Memory Allocations

Another pitfall of linked/pointer-heavy data structures: Allocation cost

- Single malloc/free pair: reasonably cheap (50-500 cycles, more for fresh memory)
- But much more expensive than other common operations
- Can often be avoided without going to extreme lengths

Cost of Memory Allocations

Another pitfall of linked/pointer-heavy data structures: Allocation cost

- Single malloc/free pair: reasonably cheap (50-500 cycles, more for fresh memory)
- But much more expensive than other common operations
- Can often be avoided without going to extreme lengths
- If you find significant malloc/free in your profiles, check for this

Cost of Memory Allocations

Another pitfall of linked/pointer-heavy data structures: Allocation cost

- Single malloc/free pair: reasonably cheap (50-500 cycles, more for fresh memory)
- But much more expensive than other common operations
- Can often be avoided without going to extreme lengths
- If you find significant malloc/free in your profiles, check for this

```
1 // Allocates a new vector EVERY iteration — heap allocation + deallocation each time
2 for (int iter = 0; iter < 1000000; iter++) {
3     std::vector<int> neighbors;            // malloc → heap allocation
4     find_neighbors(node, neighbors);       // fills it
5     best = evaluate(neighbors);            // uses it
6 }   // free → heap deallocation, then repeat
```

Cost of Memory Allocations

Another pitfall of linked/pointer-heavy data structures: Allocation cost

- Single malloc/free pair: reasonably cheap (50-500 cycles, more for fresh memory)
- But much more expensive than other common operations
- Can often be avoided without going to extreme lengths
- If you find significant malloc/free in your profiles, check for this

```cpp
// Allocates a new vector EVERY iteration — heap allocation + deallocation each time
for (int iter = 0; iter < 1000000; iter++) {
    std::vector<int> neighbors;            // malloc → heap allocation
    find_neighbors(node, neighbors);       // fills it
    best = evaluate(neighbors);            // uses it
}   // free → heap deallocation, then repeat
```

```cpp
// Allocate once, reuse — no allocation in the hot loop
std::vector<int> neighbors;
for (int iter = 0; iter < 1000000; iter++) {
    neighbors.clear();                     // size → 0, but buffer stays
    find_neighbors(node, neighbors);       // reuses existing capacity
    best = evaluate(neighbors);
}
```

Cost of Memory Allocations

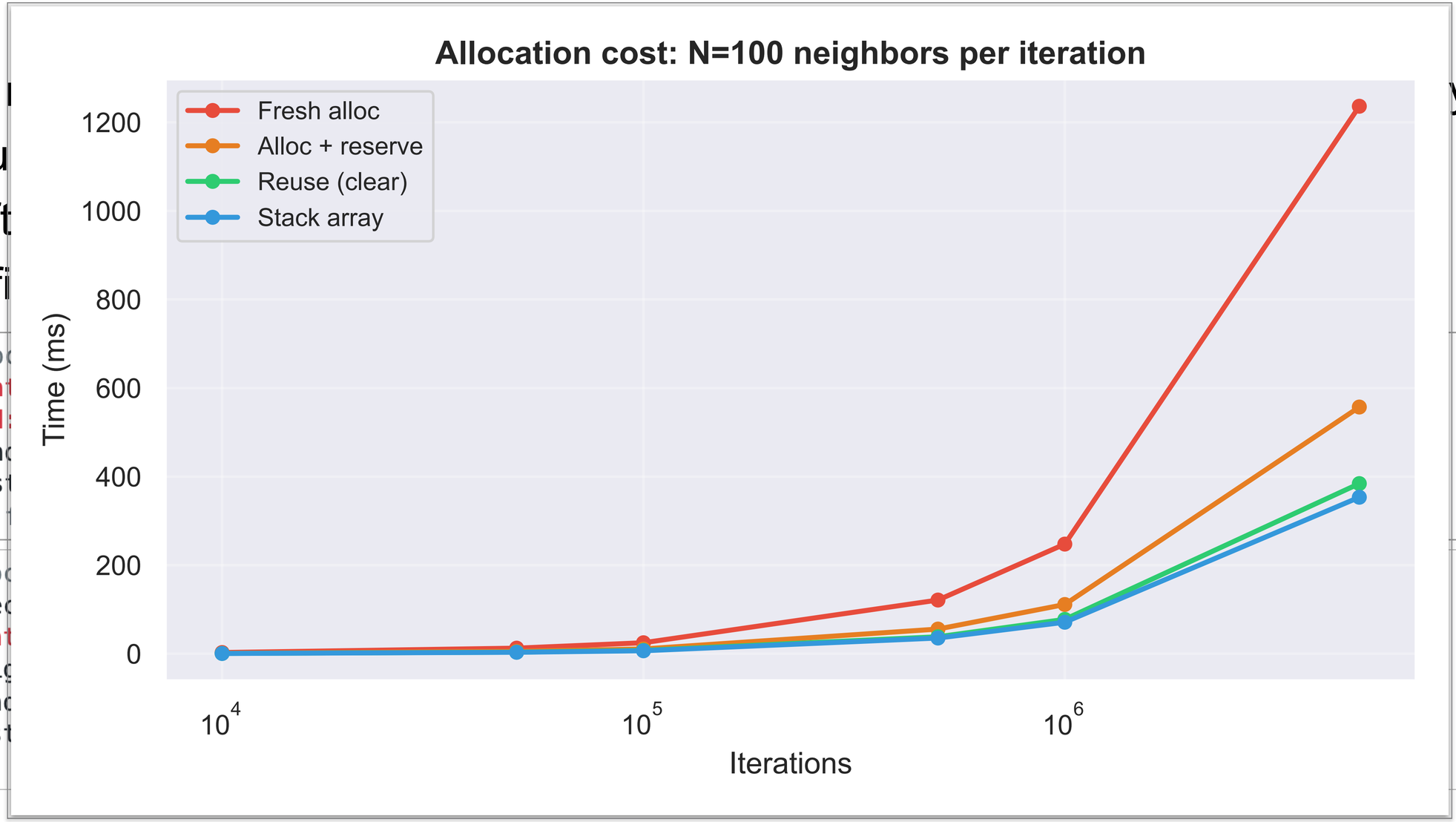
Another

- Single
- But mu
- Can oft
- If you fi

y)

```
1 // Alloc
2 for (int
3     std:
4     finc
5     best
6 }   //
```

```
1 // Alloc
2 std::vec
3 for (int
4     neig
5     finc
6     best
7 }
```



Cost of Memory Allocations

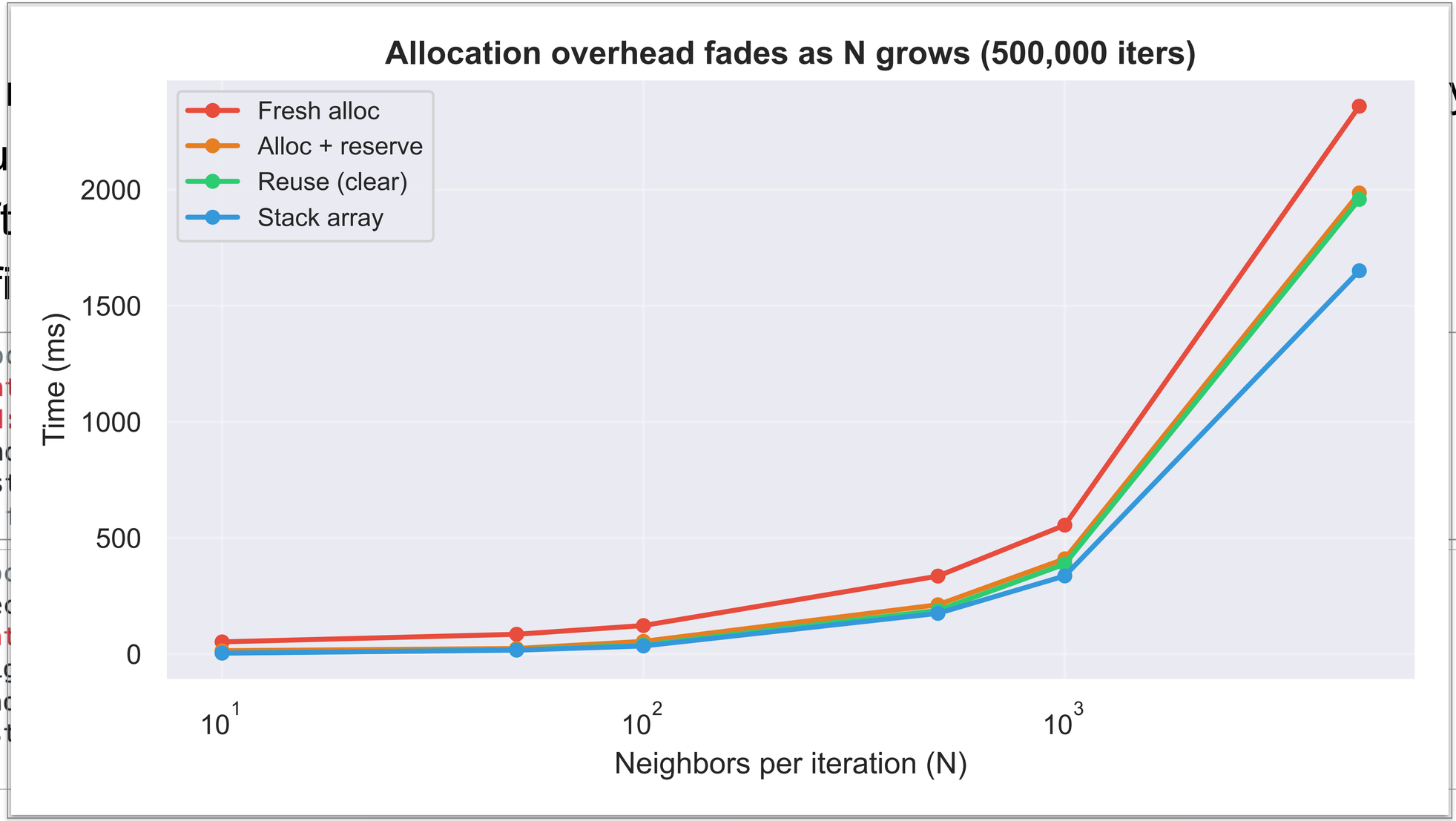
Another

- Single
- But mu
- Can oft
- If you fi

y)

```
1 // Alloc
2 for (int
3     std:
4     finc
5     best
6 }   //
```

```
1 // Alloc
2 std::vec
3 for (int
4     neig
5     finc
6     best
7 }
```



Memory Allocation: More Examples

Memory Allocation: More Examples

Individual allocations (expensive):

```
1  // Branch-and-bound: new node per branch
2  for (auto& child : branches) {
3      auto* node = new BBNode(child);
4      queue.push(node);
5  }
6  // ... later: delete every node
7
8  // String building per iteration
9  for (int i = 0; i < N; i++) {
10     std::string key = build_key(i);
11     map[key] = compute(i);
12 }
```

Memory Allocation: More Examples

Individual allocations (expensive):

```
1  // Branch-and-bound: new node per branch
2  for (auto& child : branches) {
3      auto* node = new BBNode(child);
4      queue.push(node);
5  }
6  // ... later: delete every node
7
8  // String building per iteration
9  for (int i = 0; i < N; i++) {
10     std::string key = build_key(i);
11     map[key] = compute(i);
12 }
```

Allocate once & reuse (cheaper):

```
1  // Object pool: pre-allocate, recycle
2  std::vector<BBNode> pool(max_nodes);
3  int next = 0;
4  for (auto& child : branches) {
5      pool[next] = BBNode(child);
6      queue.push(&pool[next++]);
7  }
8
9  // Reserve + reuse string buffer
10 std::string key;
11 key.reserve(64);
12 for (int i = 0; i < N; i++) {
13     build_key_into(i, key);  // writes into key
14     map[key] = compute(i);
15 }
```

Memory Allocation: More Examples

Individual allocations (expensive):

```cpp
// Branch-and-bound: new node per branch
for (auto& child : branches) {
    auto* node = new BBNode(child);
    queue.push(node);
}
// ... later: delete every node

// String building per iteration
for (int i = 0; i < N; i++) {
    std::string key = build_key(i);
    map[key] = compute(i);
}
```

Allocate once & reuse (cheaper):

```cpp
// Object pool: pre-allocate, recycle
std::vector<BBNode> pool(max_nodes);
int next = 0;
for (auto& child : branches) {
    pool[next] = BBNode(child);
    queue.push(&pool[next++]);
}

// Reserve + reuse string buffer
std::string key;
key.reserve(64);
for (int i = 0; i < N; i++) {
    build_key_into(i, key);  // writes into key
    map[key] = compute(i);
}
```

Hidden allocations: accidental copying/construction of containers, temporaries, …

Memory Allocation: More Examples

Individual allocations (expensive):

```cpp
// Branch-and-bound: new node per branch
for (auto& child : branches) {
    auto* node = new BBNode(child);
    queue.push(node);
}
// ... later: delete every node

// String building per iteration
for (int i = 0; i < N; i++) {
    std::string key = build_key(i);
    map[key] = compute(i);
}
```

Allocate once & reuse (cheaper):

```cpp
// Object pool: pre-allocate, recycle
std::vector<BBNode> pool(max_nodes);
int next = 0;
for (auto& child : branches) {
    pool[next] = BBNode(child);
    queue.push(&pool[next++]);
}

// Reserve + reuse string buffer
std::string key;
key.reserve(64);
for (int i = 0; i < N; i++) {
    build_key_into(i, key);  // writes into key
    map[key] = compute(i);
}
```

Hidden allocations: accidental copying/construction of containers, temporaries, …
Misunderstanding of C++: Objects are passed around by value by default

Memory Allocation: More Examples

Individual allocations (expensive):

```cpp
// Branch-and-bound: new node per branch
for (auto& child : branches) {
    auto* node = new BBNode(child);
    queue.push(node);
}
// ... later: delete every node

// String building per iteration
for (int i = 0; i < N; i++) {
    std::string key = build_key(i);
    map[key] = compute(i);
}
```

Allocate once & reuse (cheaper):

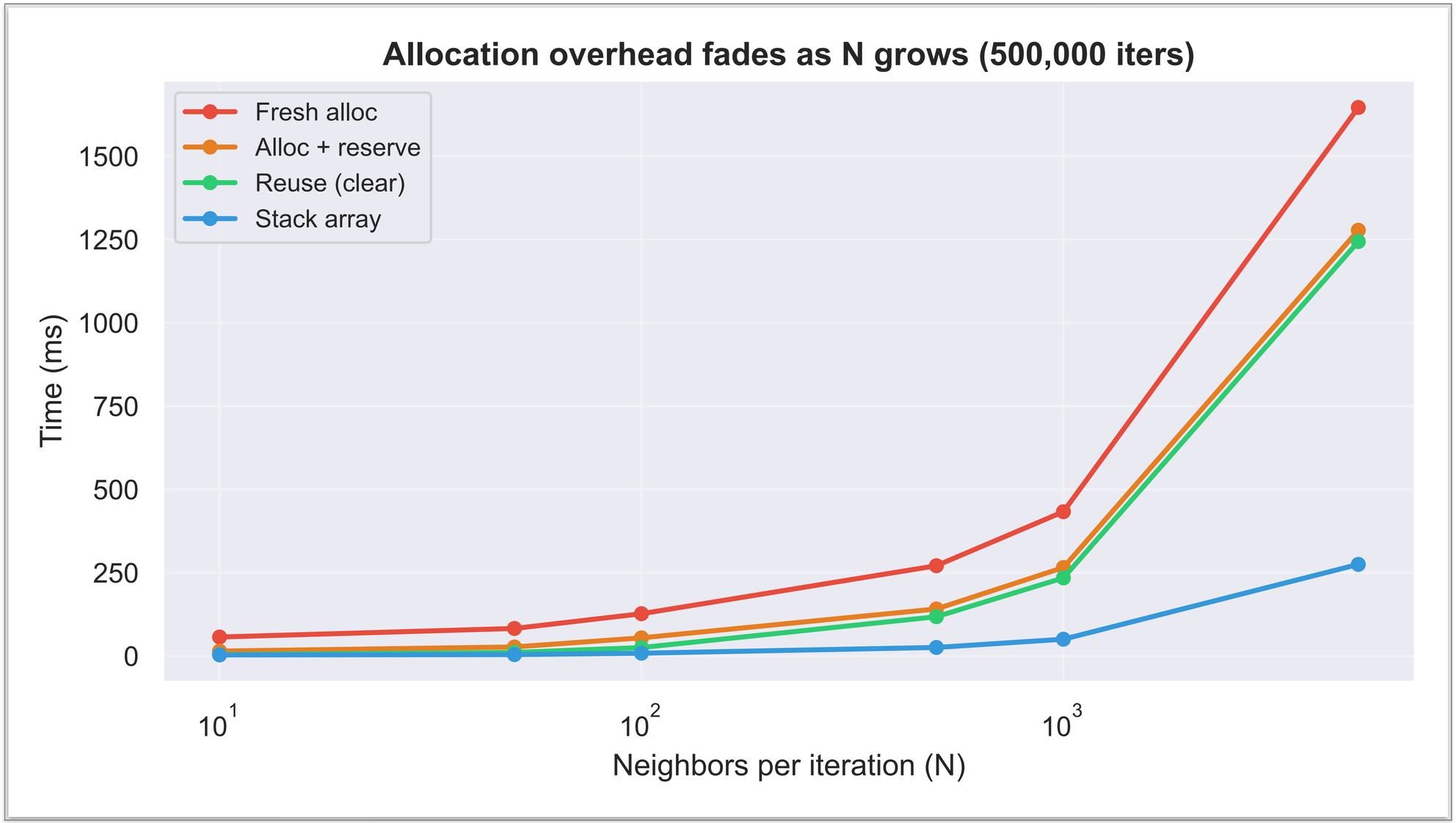
```cpp
// Object pool: pre-allocate, recycle
std::vector<BBNode> pool(max_nodes);
int next = 0;
for (auto& child : branches) {
    pool[next] = BBNode(child);
    queue.push(&pool[next++]);
}

// Reserve + reuse string buffer
std::string key;
key.reserve(64);
for (int i = 0; i < N; i++) {
    build_key_into(i, key);  // writes into key
    map[key] = compute(i);
}
```

Hidden allocations: accidental copying/construction of containers, temporaries, …
Misunderstanding of C++: Objects are passed around by value by default
Profile/measure to find such situations!

Why is the stack version cheaper?

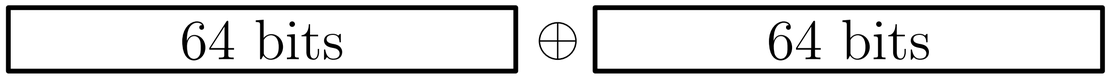
Why is the stack version cheaper?



SIMD: Doing more with one instruction

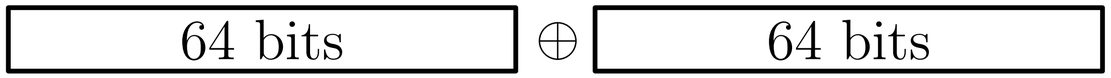
SIMD: Doing more with one instruction

Normal instructions:

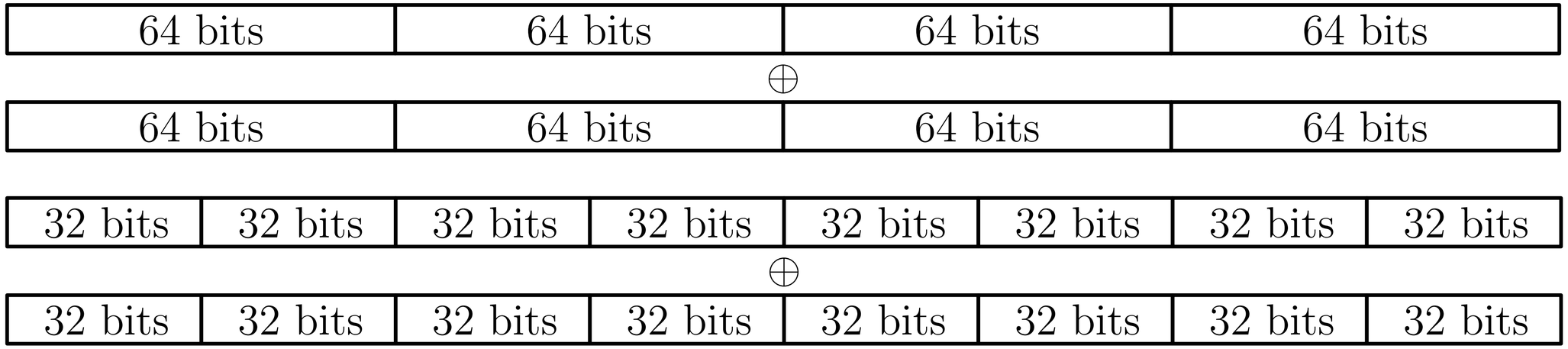


SIMD: Doing more with one instruction

Normal instructions:

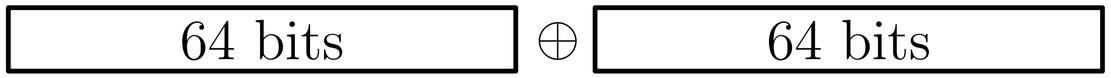


SIMD (single instruction, multiple data) instructions:

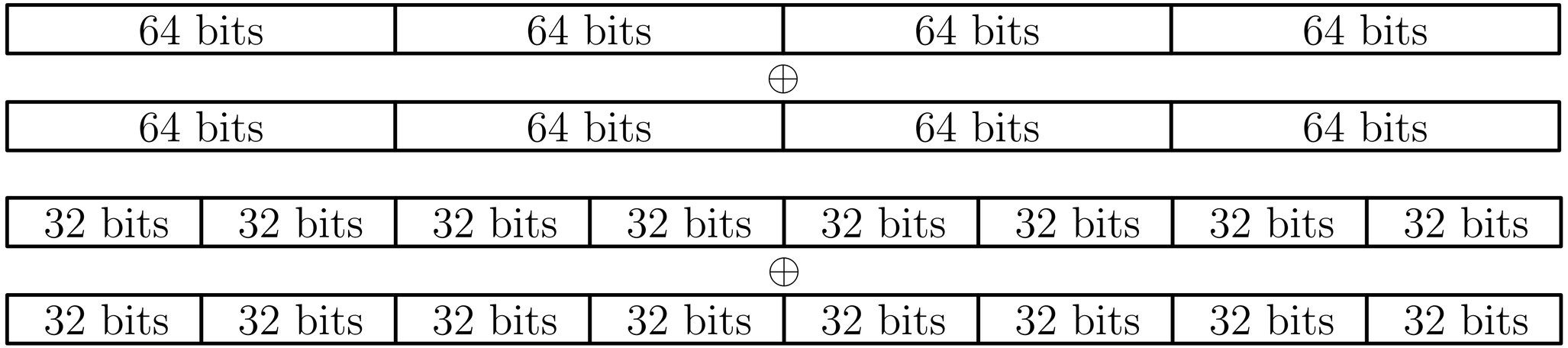


SIMD: Doing more with one instruction

Normal instructions:



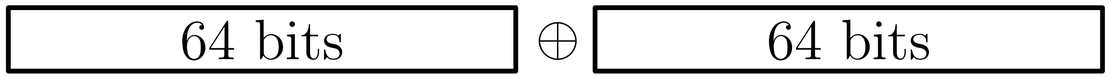
SIMD (single instruction, multiple data) instructions:



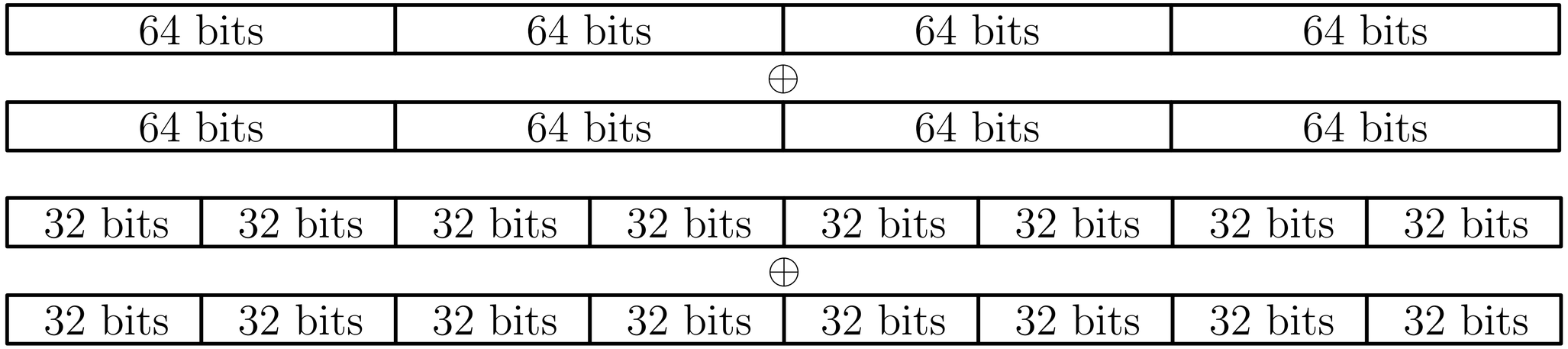
Exact available instructions: platform dependent: mov, add, sub, mul, div, xor, and, …

SIMD: Doing more with one instruction

Normal instructions:



SIMD (single instruction, multiple data) instructions:

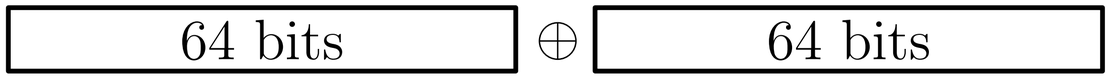


Exact available instructions: platform dependent: mov, add, sub, mul, div, xor, and, …

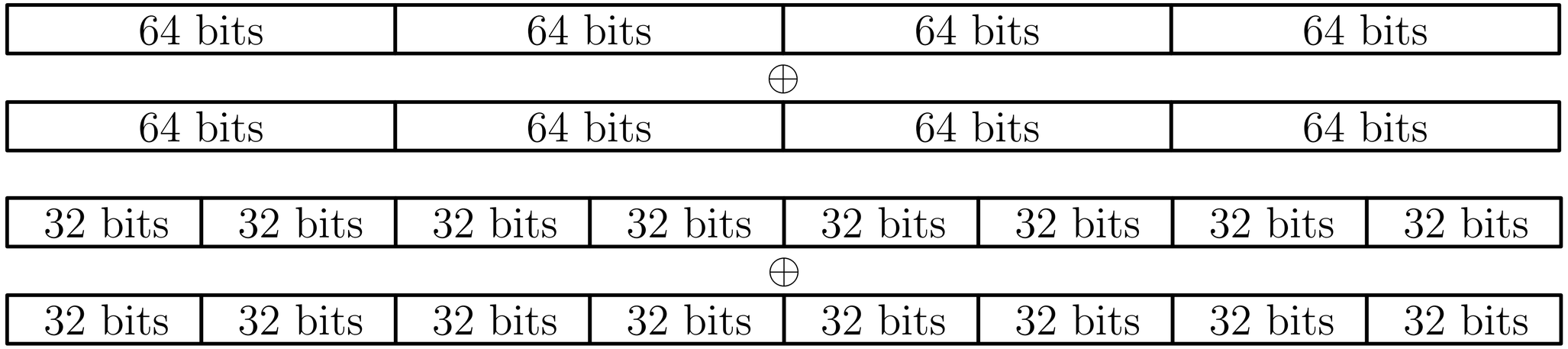
Comparisons, masking, permuting entries and other horizontal operations, extraction, …

SIMD: Doing more with one instruction

Normal instructions:



SIMD (single instruction, multiple data) instructions:



Exact available instructions: platform dependent: mov, add, sub, mul, div, xor, and, …

Comparisons, masking, permuting entries and other horizontal operations, extraction, …

Both integer and floating point, various operand and suboperand sizes (128, 256, 512 bits).

SIMD: Simplest Example

```cpp
#include <cstdint>
#include <vector>
#include <numeric>

std::int32_t sum(const std::vector<std::int32_t>& input) {
    return std::reduce(input.begin(), input.end());
}
```

SIMD: Simplest Example

```cpp
#include <cstdint>
#include <vector>
#include <numeric>

std::int32_t sum(const std::vector<std::int32_t>& input) {
    return std::reduce(input.begin(), input.end());
}
```

unvectorized (`-O1`)

```
[...]
.LBB0_1:
        add     eax, dword ptr [rcx]
        add     rcx, 4
        cmp     rcx, rdx
        jne     .LBB0_1
[...]
```

SIMD: Simplest Example

```cpp
#include <cstdint>
#include <vector>
#include <numeric>

std::int32_t sum(const std::vector<std::int32_t>& input) {
    return std::reduce(input.begin(), input.end());
}
```

vectorized (`-O2 -mavx`)

```asm
[...]
.LBB0_8:
        vpaddd  xmm0, xmm0, xmmword ptr [rdx + 4*rax]
        vpaddd  xmm1, xmm1, xmmword ptr [rdx + 4*rax + 16]
        vpaddd  xmm2, xmm2, xmmword ptr [rdx + 4*rax + 32]
        vpaddd  xmm3, xmm3, xmmword ptr [rdx + 4*rax + 48]
        add     rax, 16
        cmp     r9, rax
        jne     .LBB0_8
[...]
```

unvectorized (`-O1`)

```asm
[...]
.LBB0_1:
        add     eax, dword ptr [rcx]
        add     rcx, 4
        cmp     rcx, rdx
        jne     .LBB0_1
[...]
```

SIMD: Simplest Example

```cpp
#include <cstdint>
#include <vector>
#include <numeric>

std::int32_t sum(const std::vector<std::int32_t>& input) {
    return std::reduce(input.begin(), input.end());
}
```

vectorized (`-O2 -mavx`)

```asm
[...]
.LBB0_8:
        vpaddd  xmm0, xmm0, xmmword ptr [rdx + 4*rax]
        vpaddd  xmm1, xmm1, xmmword ptr [rdx + 4*rax + 16]
        vpaddd  xmm2, xmm2, xmmword ptr [rdx + 4*rax + 32]
        vpaddd  xmm3, xmm3, xmmword ptr [rdx + 4*rax + 48]
        add     rax, 16
        cmp     r9, rax
        jne     .LBB0_8
[...]
```

vectorized (`-O2 -mavx2`)

```asm
[...]
.LBB0_8:
        vpaddd  ymm0, ymm0, ymmword ptr [rdx + 4*rax]
        vpaddd  ymm1, ymm1, ymmword ptr [rdx + 4*rax + 32]
        vpaddd  ymm2, ymm2, ymmword ptr [rdx + 4*rax + 64]
        vpaddd  ymm3, ymm3, ymmword ptr [rdx + 4*rax + 96]
        add     rax, 32
        cmp     r9, rax
        jne     .LBB0_8
[...]
```

unvectorized (`-O1`)

```asm
[...]
.LBB0_1:
        add     eax, dword ptr [rcx]
        add     rcx, 4
        cmp     rcx, rdx
        jne     .LBB0_1
[...]
```

SIMD: Simplest Example

```cpp
#include <cstdint>
#include <vector>
#include <numeric>

std::int32_t sum(const std::vector<std::int32_t>& input) {
    return std::reduce(input.begin(), input.end());
}
```

vectorized (`-O2 -mavx`)

```asm
[...]
.LBB0_8:
        vpaddd  xmm0, xmm0, xmmword ptr [rdx + 4*rax]
        vpaddd  xmm1, xmm1, xmmword ptr [rdx + 4*rax + 16]
        vpaddd  xmm2, xmm2, xmmword ptr [rdx + 4*rax + 32]
        vpaddd  xmm3, xmm3, xmmword ptr [rdx + 4*rax + 48]
        add     rax, 16
        cmp     r9, rax
        jne     .LBB0_8
[...]
```

vectorized (`-O2 -mavx2`)

```asm
[...]
.LBB0_8:
        vpaddd  ymm0, ymm0, ymmword ptr [rdx + 4*rax]
        vpaddd  ymm1, ymm1, ymmword ptr [rdx + 4*rax + 32]
        vpaddd  ymm2, ymm2, ymmword ptr [rdx + 4*rax + 64]
        vpaddd  ymm3, ymm3, ymmword ptr [rdx + 4*rax + 96]
        add     rax, 32
        cmp     r9, rax
        jne     .LBB0_8
[...]
```

unvectorized (`-O1`)

```asm
[...]
.LBB0_1:
        add     eax, dword ptr [rcx]
        add     rcx, 4
        cmp     rcx, rdx
        jne     .LBB0_1
[...]
```

vectorized (`-O2 -mavx512f`)

```asm
[...]
.LBB0_8:
        vpaddd  zmm0, zmm0, zmmword ptr [rdx + 4*rax]
        vpaddd  zmm1, zmm1, zmmword ptr [rdx + 4*rax + 64]
        vpaddd  zmm2, zmm2, zmmword ptr [rdx + 4*rax + 128]
        vpaddd  zmm3, zmm3, zmmword ptr [rdx + 4*rax + 192]
        add     rax, 64
        cmp     r9, rax
        jne     .LBB0_8
[...]
```

SIMD: When to do it

SIMD: When to do it

Simple to medium-complexity cases:

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

Manual intervention:

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

Manual intervention:

- Often complex

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

Manual intervention:

- Often complex
- Platform- and compiler-dependent

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

Manual intervention:

- Often complex
- Platform- and compiler-dependent
- Needs special compiler/library support (e.g., `<immintrin.h>`)

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

Manual intervention:

- Often complex
- Platform- and compiler-dependent
- Needs special compiler/library support (e.g., `<immintrin.h>`)
- Rarely worth it

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

Manual intervention:

- Often complex
- Platform- and compiler-dependent
- Needs special compiler/library support (e.g., `<immintrin.h>`)
- Rarely worth it
- Measure first!

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

Manual intervention:

- Often complex
- Platform- and compiler-dependent
- Needs special compiler/library support (e.g., `<immintrin.h>`)
- Rarely worth it
- Measure first!
- Verify the compiler is not doing it (properly) & try to get the compiler to do it second!

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

Manual intervention:

- Often complex
- Platform- and compiler-dependent
- Needs special compiler/library support (e.g., `<immintrin.h>`)
- Rarely worth it
- Measure first!
- Verify the compiler is not doing it (properly) & try to get the compiler to do it second!
- Only then think about potentially doing it yourself

SIMD: When to do it

Simple to medium-complexity cases:

- Usually handled well by the compiler (`-O2` and upwards, depending on extra flags)

Manual intervention:

- Often complex
- Platform- and compiler-dependent
- Needs special compiler/library support (e.g., `<immintrin.h>`)
- Rarely worth it
- Measure first!
- Verify the compiler is not doing it (properly) & try to get the compiler to do it second!
- Only then think about potentially doing it yourself

Auto-vectorization: Why might the compiler not be doing it?

Auto-vectorization: Why might the compiler not be doing it?

Simple setup issues:

Auto-vectorization: Why might the compiler not be doing it?

Simple setup issues:

- Missing optimization flags (CMake RelWithDebInfo `-O2` vs. Release `-O3`)

Auto-vectorization: Why might the compiler not be doing it?

Simple setup issues:

- Missing optimization flags (CMake RelWithDebInfo `-O2` vs. Release `-O3`)
  - clang has less drastic differences between `-O2` and `-O3` than gcc

Auto-vectorization: Why might the compiler not be doing it?

Simple setup issues:

- Missing optimization flags (CMake RelWithDebInfo `-O2` vs. Release `-O3`)
  - clang has less drastic differences between `-O2` and `-O3` than gcc
  - In my compiler explorer tests, gcc often failed even for simple loops with `-O2`

Auto-vectorization: Why might the compiler not be doing it?

Simple setup issues:

- Missing optimization flags (CMake RelWithDebInfo `-O2` vs. Release `-O3`)
  - clang has less drastic differences between `-O2` and `-O3` than gcc
  - In my compiler explorer tests, gcc often failed even for simple loops with `-O2`
  - Only enables ‘very cheap’ model at `-O2` (balance code size vs. potential speed-up)

Auto-vectorization: Why might the compiler not be doing it?

Simple setup issues:

- Missing optimization flags (CMake RelWithDebInfo `-O2` vs. Release `-O3`)
  - clang has less drastic differences between `-O2` and `-O3` than gcc
  - In my compiler explorer tests, gcc often failed even for simple loops with `-O2`
  - Only enables ‘very cheap’ model at `-O2` (balance code size vs. potential speed-up)
- Some compilers can print diagnostics (`-fopt-info-vec-optimized` for gcc)

Auto-vectorization: Why might the compiler not be doing it?

Simple setup issues:

- Missing optimization flags (CMake RelWithDebInfo `-O2` vs. Release `-O3`)
  - clang has less drastic differences between `-O2` and `-O3` than gcc
  - In my compiler explorer tests, gcc often failed even for simple loops with `-O2`
  - Only enables ‘very cheap’ model at `-O2` (balance code size vs. potential speed-up)
- Some compilers can print diagnostics (`-fopt-info-vec-optimized` for gcc)
- Missing assumed hardware support (e.g., `-mavx2` (gcc, clang), `/arch:AVX2`)

Auto-vectorization: Why might the compiler not be doing it?

Simple setup issues:

- Missing optimization flags (CMake RelWithDebInfo `-O2` vs. Release `-O3`)
  - clang has less drastic differences between `-O2` and `-O3` than gcc
  - In my compiler explorer tests, gcc often failed even for simple loops with `-O2`
  - Only enables ‘very cheap’ model at `-O2` (balance code size vs. potential speed-up)
- Some compilers can print diagnostics (`-fopt-info-vec-optimized` for gcc)
- Missing assumed hardware support (e.g., `-mavx2` (gcc, clang), `/arch:AVX2`)
- Implicit assumptions for x86_64: at least SSE2 (128 bit vector registers)

Auto-vectorization: Why might the compiler not be doing it?

Simple setup issues:

- Missing optimization flags (CMake RelWithDebInfo `-O2` vs. Release `-O3`)
  - clang has less drastic differences between `-O2` and `-O3` than gcc
  - In my compiler explorer tests, gcc often failed even for simple loops with `-O2`
  - Only enables ‘very cheap’ model at `-O2` (balance code size vs. potential speed-up)
- Some compilers can print diagnostics (`-fopt-info-vec-optimized` for gcc)
- Missing assumed hardware support (e.g., `-mavx2` (gcc, clang), `/arch:AVX2`)
- Implicit assumptions for x86_64: at least SSE2 (128 bit vector registers)
- Implicit assumptions for, e.g., M1 offer more (newer hardware baseline)

Auto-vectorization: Let’s play a game.

Auto-vectorization: Why might the compiler not be doing it?

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)

```
 1 std::int64_t sum(const std::vector<std::int64_t>& v) {
 2     std::int64_t r = 0;
 3     for(std::size_t i = 0, n = v.size(); i < n; ++i) {
 4         r += v.at(i); // contains a bounds check
 5     }
 6     return r;
 7 }
 8
 9 std::int64_t sum2(const std::vector<std::int64_t>& v) {
10     std::int64_t r = 0;
11     for(std::size_t i = 0, n = v.size(); i < n; ++i) {
12         r += v[i]; // contains no bounds check
13     }
14     return r;
15 }
```

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)

```cpp
std::int64_t sum(const std::vector<std::int64_t>& v) {
    std::int64_t r = 0;
    for(std::size_t i = 0, n = v.size(); i < n; ++i) {
        r += v.at(i); // contains a bounds check
    }
    return r;
}

std::int64_t sum2(const std::vector<std::int64_t>& v) {
    std::int64_t r = 0;
    for(std::size_t i = 0, n = v.size(); i < n; ++i) {
        r += v[i]; // contains no bounds check
    }
    return r;
}
```

Both vectorized (clang at `-O2`, gcc only at `-O3`) & essentially identical assembly.

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)

```cpp
std::int64_t sum(const std::vector<std::int64_t>& v) {
    std::int64_t r = 0;
    for(std::size_t i = 0, n = v.size(); i < n; ++i) {
        r += v.at(i); // contains a bounds check
    }
    return r;
}

std::int64_t sum2(const std::vector<std::int64_t>& v) {
    std::int64_t r = 0;
    for(std::size_t i = 0, n = v.size(); i < n; ++i) {
        r += v[i]; // contains no bounds check
    }
    return r;
}
```

Both vectorized (clang at `-O2`, gcc only at `-O3`) & essentially identical assembly.
MSVC (`/O2 /arch:AVX2`) only vectorizes `sum2` (but eliminates bounds check…)

Auto-vectorization: Why might the compiler not be doing it?

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns

```
1 std::int64_t telescopic_sum(const std::vector<std::int64_t>& v) {
2     std::int64_t r = 0;
3     for(std::size_t i = 1, n = v.size(); i < n; ++i) {
4         r += (v[i] - v[i-1]);
5     }
6     return r;
7 }
```

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns

```
1 std::int64_t telescopic_sum(const std::vector<std::int64_t>& v) {
2     std::int64_t r = 0;
3     for(std::size_t i = 1, n = v.size(); i < n; ++i) {
4         r += (v[i] - v[i-1]);
5     }
6     return r;
7 }
```

No compiler found the hidden $O(1)$.

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns

```
1 std::int64_t telescopic_sum(const std::vector<std::int64_t>& v) {
2     std::int64_t r = 0;
3     for(std::size_t i = 1, n = v.size(); i < n; ++i) {
4         r += (v[i] - v[i-1]);
5     }
6     return r;
7 }
```

No compiler found the hidden $O(1)$.

Vectorization: clang (`-O2`): ✅, gcc (`-O3`): ✅, MSVC (`/O2 /Ob3`): ❌

Auto-vectorization: Why might the compiler not be doing it?

```cpp
void sequence2(std::vector<std::int64_t>& v,
               const std::vector<std::int64_t>& w)
{
    if(w.size() != v.size()) {
        return;
    }
    for(std::size_t i = 1, n = v.size(); i < n; ++i) {
        v[i] = v[i-1] * w[i];
    }
}
```

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns
- Inter-iteration dependencies

```
 1 void sequence2(std::vector<std::int64_t>& v,
 2                 const std::vector<std::int64_t>& w)
 3 {
 4     if(w.size() != v.size()) {
 5         return;
 6     }
 7     for(std::size_t i = 1, n = v.size(); i < n; ++i) {
 8         v[i] = v[i-1] * w[i];
 9     }
10 }
```

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns
- Inter-iteration dependencies

```
 1 void sequence2(std::vector<std::int64_t>& v,
 2                 const std::vector<std::int64_t>& w)
 3 {
 4     if(w.size() != v.size()) {
 5         return;
 6     }
 7     for(std::size_t i = 1, n = v.size(); i < n; ++i) {
 8         v[i] = v[i-1] * w[i];
 9     }
10 }
```

No way to vectorize - dependency chain `v[n-1]` - `v[n-2]` - ... - `v[0]`!

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns
- Inter-iteration dependencies

```
 1 void sequence3(std::vector<std::int64_t>& v,
 2                 const std::vector<std::int64_t>& w)
 3 {
 4     if(w.size() != v.size()) {
 5         return;
 6     }
 7     for(std::size_t i = 0, n = v.size(); i < n - 1; ++i) {
 8         v[i] = v[i+1] * w[i];
 9     }
10 }
```

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns
- Inter-iteration dependencies

```
 1 void sequence3(std::vector<std::int64_t>& v,
 2                 const std::vector<std::int64_t>& w)
 3 {
 4     if(w.size() != v.size()) {
 5         return;
 6     }
 7     for(std::size_t i = 0, n = v.size(); i < n - 1; ++i) {
 8         v[i] = v[i+1] * w[i];
 9     }
10 }
```

gcc, clang and MSVC: ✅

Auto-vectorization: Why might the compiler not be doing it?

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns
- Inter-iteration dependencies
- Pointer-chasing (linked lists, trees, …), irregular memory access (hash tables, …)
- Non-uniform operations

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns
- Inter-iteration dependencies
- Pointer-chasing (linked lists, trees, …), irregular memory access (hash tables, …)
- Non-uniform operations

```
1 void collatz_update(std::vector<std::int64_t>& v) {
2     for(std::size_t i = 0, n = v.size(); i < n; ++i) {
3         v[i] = (v[i] & 1) ? (v[i] * 3 + 1) : v[i] / 2;
4     }
5 }
```

Auto-vectorization: Why might the compiler not be doing it?

'Obvious' code issues:

- AoS with lots of extra fields (large gaps in accessed memory)
- Control flow in loop is too complex (if/else that remains branching, non-inlined calls)
- Complex access patterns
- Inter-iteration dependencies
- Pointer-chasing (linked lists, trees, …), irregular memory access (hash tables, …)
- Non-uniform operations

```
1 void collatz_update(std::vector<std::int64_t>& v) {
2     for(std::size_t i = 0, n = v.size(); i < n; ++i) {
3         v[i] = (v[i] & 1) ? (v[i] * 3 + 1) : v[i] / 2;
4     }
5 }
```

clang (`-O2`): ✅, gcc (`-O3`): ✅, MSVC (`/O2 /Ob3`): ❌ (remains branching)

Auto-vectorization: Subtle code issues

```
 1 float floatplus(float y, float z) { return y + z; }
 2
 3 float sum1(const std::vector<float>& x) {
 4     return std::accumulate(x.begin(), x.end());
 5 }
 6
 7 float sum2(const std::vector<float>& x) {
 8     return std::reduce(x.begin(), x.end(),
 9                        0.0f, floatplus);
10 }
11
12 float sum3(const std::vector<float>& x) {
13     float r = 0.0f;
14     for(float f : x) { r += f; }
15     return r;
16 }
```

Quiz: Which compiler can vectorize which function?

Auto-vectorization: Subtle code issues

```cpp
float floatplus(float y, float z) { return y + z; }

float sum1(const std::vector<float>& x) {
    return std::accumulate(x.begin(), x.end());
}

float sum2(const std::vector<float>& x) {
    return std::reduce(x.begin(), x.end(),
                       0.0f, floatplus);
}

float sum3(const std::vector<float>& x) {
    float r = 0.0f;
    for(float f : x) { r += f; }
    return r;
}
```

Quiz: Which compiler can vectorize which function?

MSVC & AppleClang: none

Auto-vectorization: Subtle code issues

```cpp
float floatplus(float y, float z) { return y + z; }

float sum1(const std::vector<float>& x) {
    return std::accumulate(x.begin(), x.end());
}

float sum2(const std::vector<float>& x) {
    return std::reduce(x.begin(), x.end(),
                       0.0f, floatplus);
}

float sum3(const std::vector<float>& x) {
    float r = 0.0f;
    for(float f : x) { r += f; }
    return r;
}
```

Quiz: Which compiler can vectorize which function?

MSVC & AppleClang: none **GCC:** `sum2`

Auto-vectorization: Subtle code issues

```
 1 float floatplus(float y, float z) { return y + z; }
 2
 3 float sum1(const std::vector<float>& x) {
 4     return std::accumulate(x.begin(), x.end());
 5 }
 6
 7 float sum2(const std::vector<float>& x) {
 8     return std::reduce(x.begin(), x.end(),
 9                        0.0f, floatplus);
10 }
11
12 float sum3(const std::vector<float>& x) {
13     float r = 0.0f;
14     for(float f : x) { r += f; }
15     return r;
16 }
```

Quiz: Which compiler can vectorize which function?

MSVC & AppleClang: none **GCC:** `sum2` **clang:** `sum2`

Auto-vectorization: Subtle code issues

```cpp
float floatplus(float y, float z) { return y + z; }

float sum1(const std::vector<float>& x) {
    return std::accumulate(x.begin(), x.end());
}

float sum2(const std::vector<float>& x) {
    return std::reduce(x.begin(), x.end(),
                       0.0f, floatplus);
}

float sum3(const std::vector<float>& x) {
    float r = 0.0f;
    for(float f : x) { r += f; }
    return r;
}
```

Quiz: Which compiler can vectorize which function?

MSVC & AppleClang: none **GCC:** `sum2` **clang:** `sum2` **But:** why?

Auto-vectorization: Subtle code issues

```
1  float floatplus(float y, float z) { return y + z; }
2
3  float sum1(const std::vector<float>& x) {
4      return std::accumulate(x.begin(), x.end());
5  }
6
7  float sum2(const std::vector<float>& x) {
8      return std::reduce(x.begin(), x.end(),
9                         0.0f, floatplus);
10 }
11
12 float sum
13     float
14     for(f
15     retur
16 }
```

Hint:

Difference from `std::accumulate`: `std::accumulate` performs strict left-to-right summation. `std::reduce` can reorder operations, which is why it requires associative/commutative operations, but permits parallel execution.

Quiz: Which compiler can vectorize which function?

MSVC & AppleClang: none **GCC:** `sum2` **clang:** `sum2` **But:** why?

Auto-vectorization: Subtle code issues

```
1  float floatplus(float y, float z) { return y + z; }
2
3  float sum1(const std::vector<float>& x) {
4      return std::accumulate(x.begin(), x.end());
5  }
6
7  float sum2(const std::vector<float>& x) {
8      return std::reduce(x.begin(), x.end(),
9                         0.0f, floatplus);
10 }
11
12 float sum
13     float
14     for(f
15     retur
16 }
```

Hint:

Difference from `std::accumulate`: `std::accumulate` performs strict left-to-right summation. `std::reduce` can reorder operations, which is why it requires associative/commutative operations, but permits parallel execution.

Quiz: Which compiler can vectorize which function?

MSVC & AppleClang: none **GCC:** `sum2` **clang:** `sum2` **But:** why?

As-if rule prohibits reordering of float additions (enforces dependency chain)

Auto-vectorization: Subtle code issues

```
1  float floatplus(float y, float z) { return y + z; }
2
3  float sum1(const std::vector<float>& x) {
4      return std::accumulate(x.begin(), x.end());
5  }
6
7  float sum2(const std::vector<float>& x) {
8      return std::reduce(x.begin(), x.end(),
9                         0.0f, floatplus);
10 }
11
12 float sum
13     float
14     for(f
15     retur
16 }
```

Hint:

Difference from `std::accumulate`: `std::accumulate` performs strict left-to-right summation. `std::reduce` can reorder operations, which is why it requires associative/commutative operations, but permits parallel execution.

Quiz: Which compiler can vectorize which function?

MSVC & AppleClang: none **GCC:** `sum2` **clang:** `sum2` **But:** why?

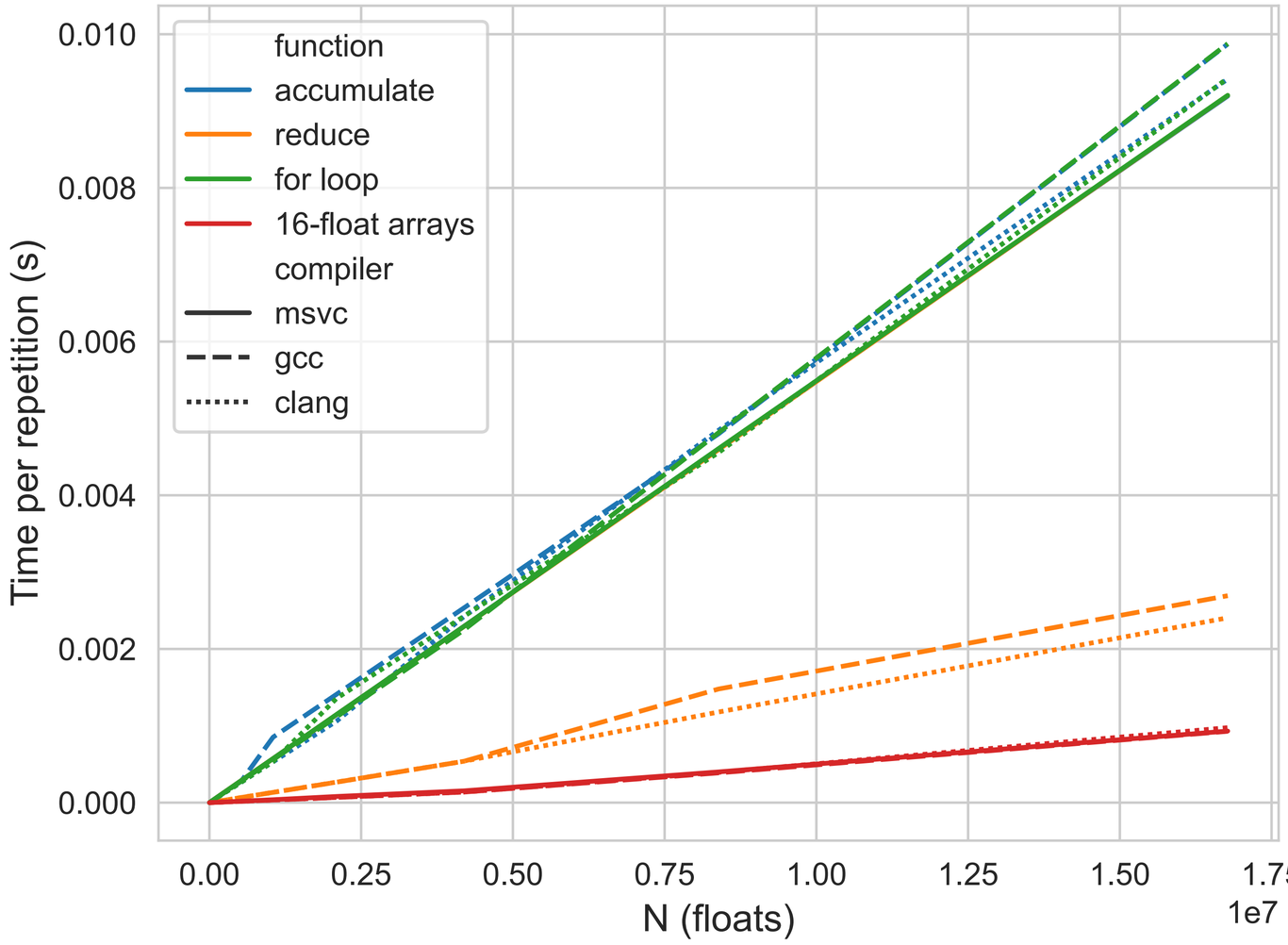
As-if rule prohibits reordering of float additions (enforces dependency chain)
Float addition is commutative, but not strictly speaking associative!

Auto-vectorization: Alternative Version

```cpp
float sum4(const std::vector<float>& x) {
    const std::size_t n = x.size(), i = 0;
    if(n < 32) return std::accumulate(x.begin(), x.end());
    alignas(64) std::array<float, 16> subsums, next;
    std::fill_n(&subsums[0], 16, 0.0f);
    for(; i + 15 < n; i += 16) {
        for(int j = 0; j < 16; ++j) {
            next[j] = x[i + j];
        }
        for(int j = 0; j < 16; ++j) {
            subsums[j] += next[j];
        }
    }
    // handle <= 15 remaining values
    for(std::size_t j = 0; i + j < n; ++j)
        subsums[j] += x[i + j];
    }
    // sum intermediate values
    for(int k = 8; k > 0; k /= 2) {
        for(int j = 0; j < k; ++j) {
            subsums[j] += subsums[j + k];
        }
    }
    return subsums[0];
}
```

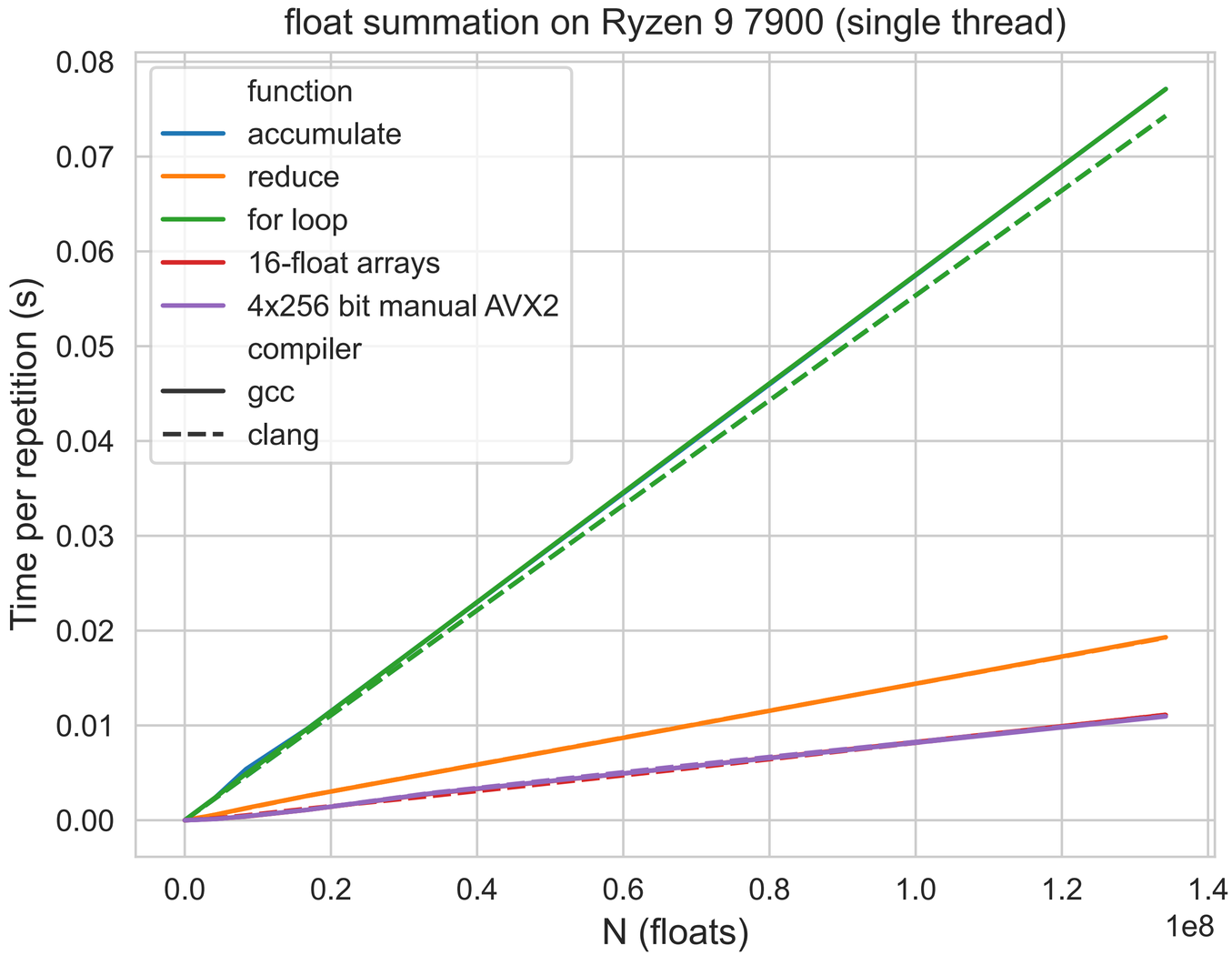
Auto-vectorization: Alternative Version

```cpp
float sum4(const std::vector<float>& x) {
    const std::size_t n = x.size(), i = 0;
    if(n < 32) return std::accumulate(x.begin(), x.end());
    alignas(64) std::array<float, 16> subsums, next;
    std::fill_n(&subsums[0], 16, 0.0f);
    for(; i + 15 < n; i += 16) {
        for(int j = 0; j < 16; ++j) {
            next[j] = x[i + j];
        }
        for(int j = 0; j < 16; ++j) {
            subsums[j] += next[j];
        }
    }
    // handle <= 15 remaining values
    for(std::size_t j = 0; i + j < n; ++j)
        subsums[j] += x[i + j];
    }
    // sum intermediate values
    for(int k = 8; k > 0; k /= 2) {
        for(int j = 0; j < k; ++j) {
            subsums[j] += subsums[j + k];
        }
    }
    return subsums[0];
}
```



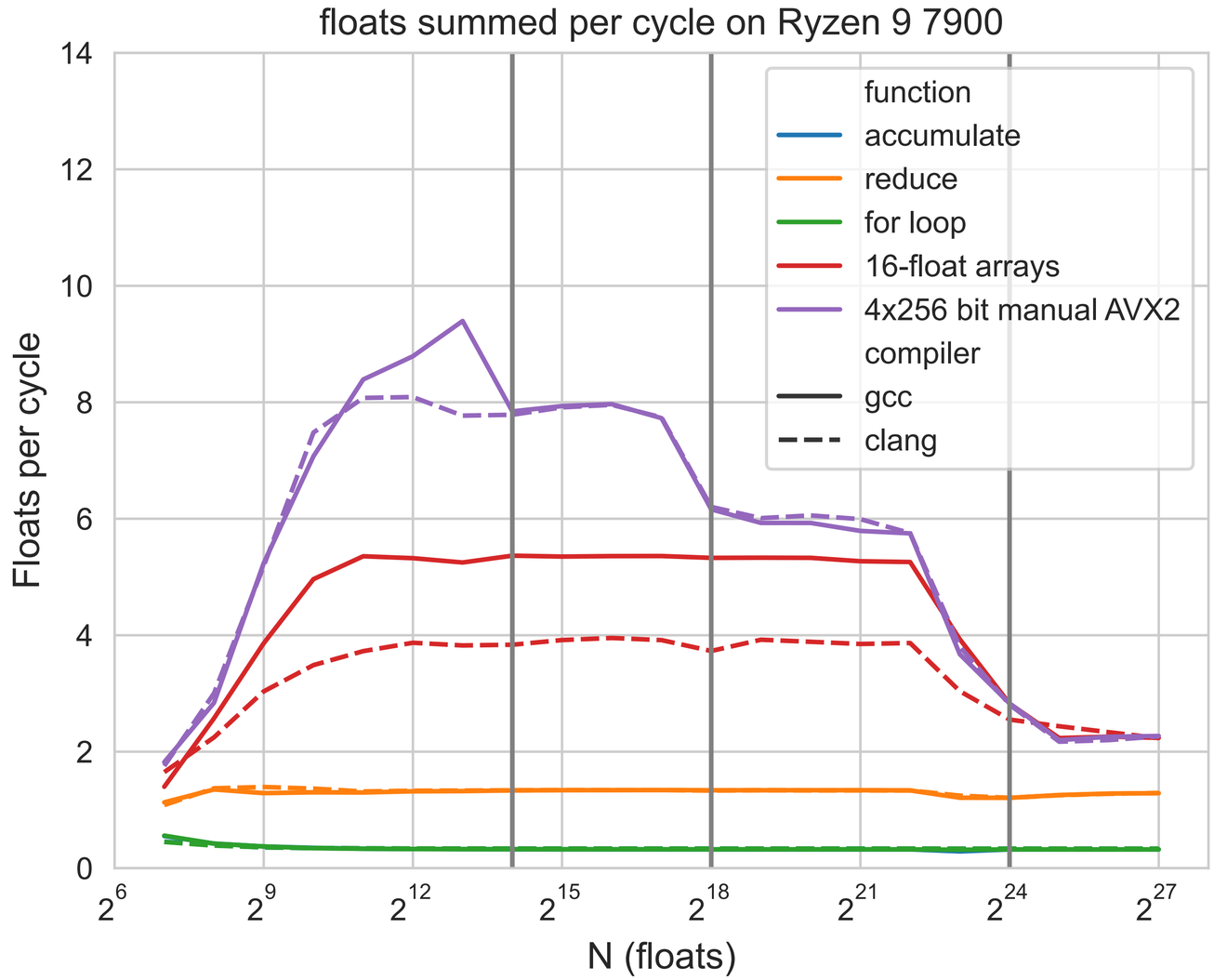
Auto-vectorization: Alternative Version

```cpp
float sum4(const std::vector<float>& x) {
    const std::size_t n = x.size(), i = 0;
    if(n < 32) return std::accumulate(x.begin(), x.end());
    alignas(64) std::array<float, 16> subsums, next;
    std::fill_n(&subsums[0], 16, 0.0f);
    for(; i + 15 < n; i += 16) {
        for(int j = 0; j < 16; ++j) {
            next[j] = x[i + j];
        }
        for(int j = 0; j < 16; ++j) {
            subsums[j] += next[j];
        }
    }
    // handle <= 15 remaining values
    for(std::size_t j = 0; i + j < n; ++j)
        subsums[j] += x[i + j];
    }
    // sum intermediate values
    for(int k = 8; k > 0; k /= 2) {
        for(int j = 0; j < k; ++j) {
            subsums[j] += subsums[j + k];
        }
    }
    return subsums[0];
}
```



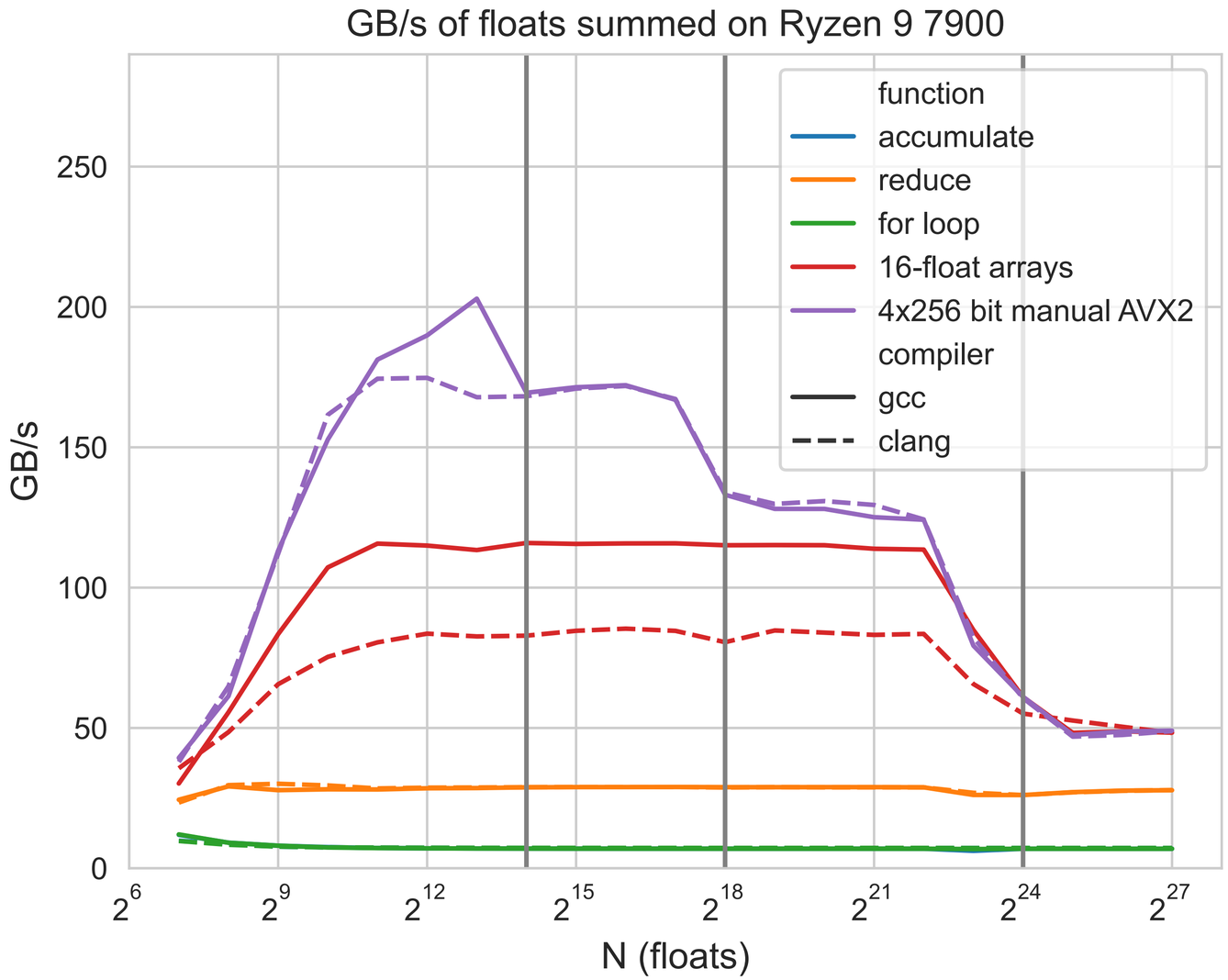
Auto-vectorization: Alternative Version

```cpp
float sum4(const std::vector<float>& x) {
    const std::size_t n = x.size(), i = 0;
    if(n < 32) return std::accumulate(x.begin(), x.end());
    alignas(64) std::array<float, 16> subsums, next;
    std::fill_n(&subsums[0], 16, 0.0f);
    for(; i + 15 < n; i += 16) {
        for(int j = 0; j < 16; ++j) {
            next[j] = x[i + j];
        }
        for(int j = 0; j < 16; ++j) {
            subsums[j] += next[j];
        }
    }
    // handle <= 15 remaining values
    for(std::size_t j = 0; i + j < n; ++j)
        subsums[j] += x[i + j];
    }
    // sum intermediate values
    for(int k = 8; k > 0; k /= 2) {
        for(int j = 0; j < k; ++j) {
            subsums[j] += subsums[j + k];
        }
    }
    return subsums[0];
}
```



Auto-vectorization: Alternative Version

```cpp
float sum4(const std::vector<float>& x) {
    const std::size_t n = x.size(), i = 0;
    if(n < 32) return std::accumulate(x.begin(), x.end());
    alignas(64) std::array<float, 16> subsums, next;
    std::fill_n(&subsums[0], 16, 0.0f);
    for(; i + 15 < n; i += 16) {
        for(int j = 0; j < 16; ++j) {
            next[j] = x[i + j];
        }
        for(int j = 0; j < 16; ++j) {
            subsums[j] += next[j];
        }
    }
    // handle <= 15 remaining values
    for(std::size_t j = 0; i + j < n; ++j)
        subsums[j] += x[i + j];
    }
    // sum intermediate values
    for(int k = 8; k > 0; k /= 2) {
        for(int j = 0; j < k; ++j) {
            subsums[j] += subsums[j + k];
        }
    }
    return subsums[0];
}
```



More subtleties: Alignment

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

- Many (SIMD) operations have strict alignment requirements: the address of x should be divisible by `sizeof(`x`)`

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

- Many (SIMD) operations have strict alignment requirements:
  the address of x should be divisible by `sizeof(`x`)`
- `malloc/new` return sufficiently aligned memory; possible exception: array `new`

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

- Many (SIMD) operations have strict alignment requirements:
  the address of x should be divisible by `sizeof(`x`)`
- `malloc/new` return sufficiently aligned memory; possible exception: array `new`
- Compiler has to handle alignment requirements; can hamper auto-vectorization

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

- Many (SIMD) operations have strict alignment requirements:
  the address of x should be divisible by `sizeof(`x`)`
- `malloc/new` return sufficiently aligned memory; possible exception: array `new`
- Compiler has to handle alignment requirements; can hamper auto-vectorization
- Sometimes only runtime penalties; these once were much harsher than they are now

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

- Many (SIMD) operations have strict alignment requirements:
  the address of x should be divisible by `sizeof(`x`)`
- `malloc/new` return sufficiently aligned memory; possible exception: array `new`
- Compiler has to handle alignment requirements; can hamper auto-vectorization
- Sometimes only runtime penalties; these once were much harsher than they are now
- Particularly bad: loads that split cache lines or even pages

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

- Many (SIMD) operations have strict alignment requirements:
  the address of x should be divisible by `sizeof(`x`)`
- `malloc/new` return sufficiently aligned memory; possible exception: array `new`
- Compiler has to handle alignment requirements; can hamper auto-vectorization
- Sometimes only runtime penalties; these once were much harsher than they are now
- Particularly bad: loads that split cache lines or even pages

Padding: `p->b, p[1].b` should be aligned (if we assume `p` is)

```
1 struct DataUnsorted {
2     char a;
3     long long b;
4     bool c;
5     double d;
6     int e;
7 };
8 // sizeof(DataUnsorted) == ?
```

```
1 struct DataSorted {
2         long long b;
3         double d;
4         int e;
5         char a;
6         bool c;
7 };
8 // sizeof(DataSorted) == ?
```

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

- Many (SIMD) operations have strict alignment requirements:
  the address of x should be divisible by `sizeof(`x`)`
- `malloc/new` return sufficiently aligned memory; possible exception: array `new`
- Compiler has to handle alignment requirements; can hamper auto-vectorization
- Sometimes only runtime penalties; these once were much harsher than they are now
- Particularly bad: loads that split cache lines or even pages

Padding: `p->b, p[1].b` should be aligned (if we assume `p` is)

```
1 struct DataUnsorted {
2     char a;
3     long long b;
4     bool c;
5     double d;
6     int e;
7 };
8 // sizeof(DataUnsorted) == 40
```

```
1 struct DataSorted {
2         long long b;
3         double d;
4         int e;
5         char a;
6         bool c;
7 };
8 // sizeof(DataSorted) == ?
```

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

- Many (SIMD) operations have strict alignment requirements:
  the address of x should be divisible by `sizeof(`x`)`
- `malloc/new` return sufficiently aligned memory; possible exception: array `new`
- Compiler has to handle alignment requirements; can hamper auto-vectorization
- Sometimes only runtime penalties; these once were much harsher than they are now
- Particularly bad: loads that split cache lines or even pages

Padding: `p->b, p[1].b` should be aligned (if we assume `p` is)

```cpp
struct DataUnsorted {
    char a;
    long long b;
    bool c;
    double d;
    int e;
};
// sizeof(DataUnsorted) == 40
```

```cpp
struct DataSorted {
        long long b;
        double d;
        int e;
        char a;
        bool c;
};
// sizeof(DataSorted) == 24
```

More subtleties: Alignment

Alignment: Trivial-looking changes can make a difference

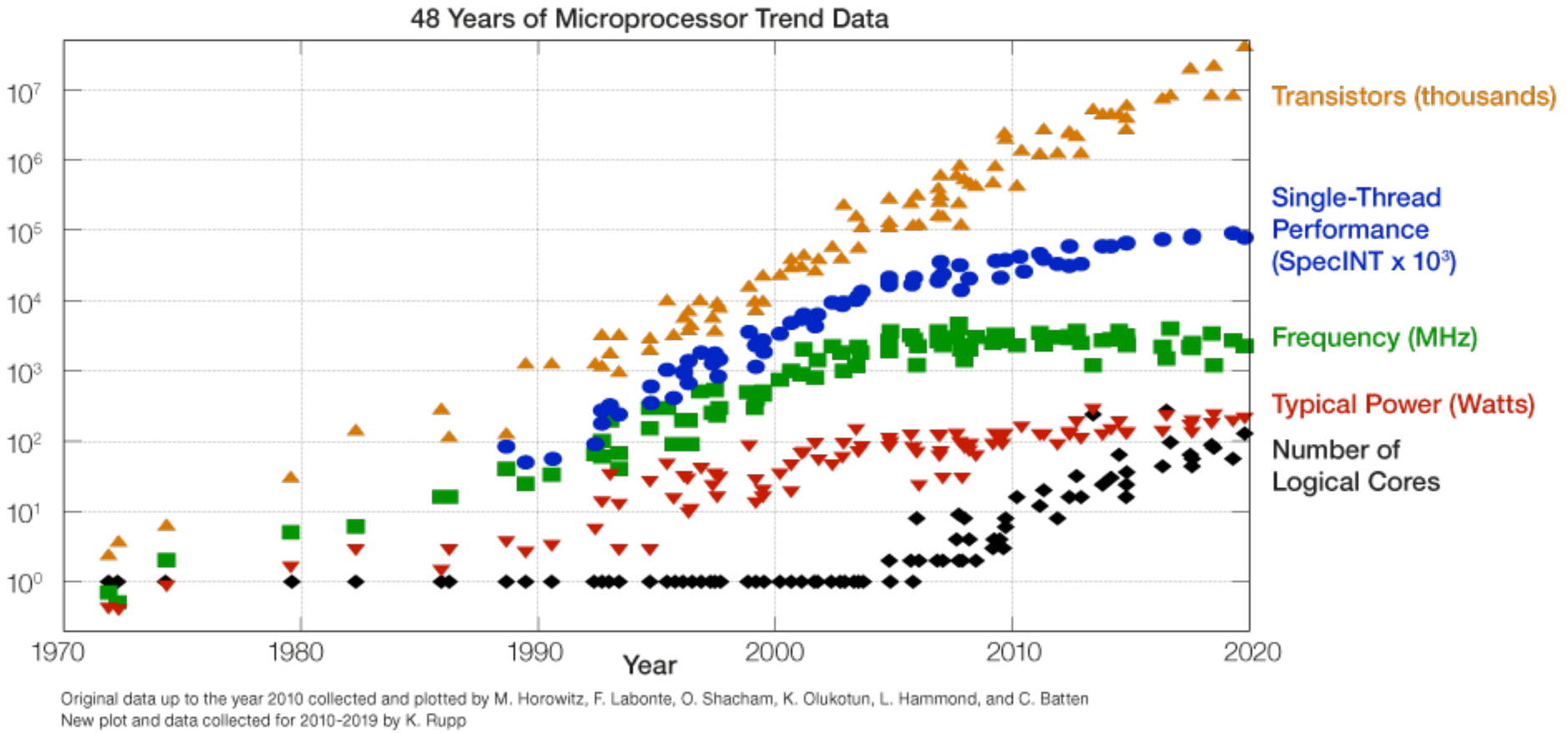
- Many (SIMD) operations have strict alignment requirements:
  the address of x should be divisible by `sizeof(`x`)`
- `malloc/new` return sufficiently aligned memory; possible exception: array `new`
- Compiler has to handle alignment requirements; can hamper auto-vectorization
- Sometimes only runtime penalties; these once were much harsher than they are now
- Particularly bad: loads that split cache lines or even pages

Padding: `p->b, p[1].b` should be aligned (if we assume `p` is)

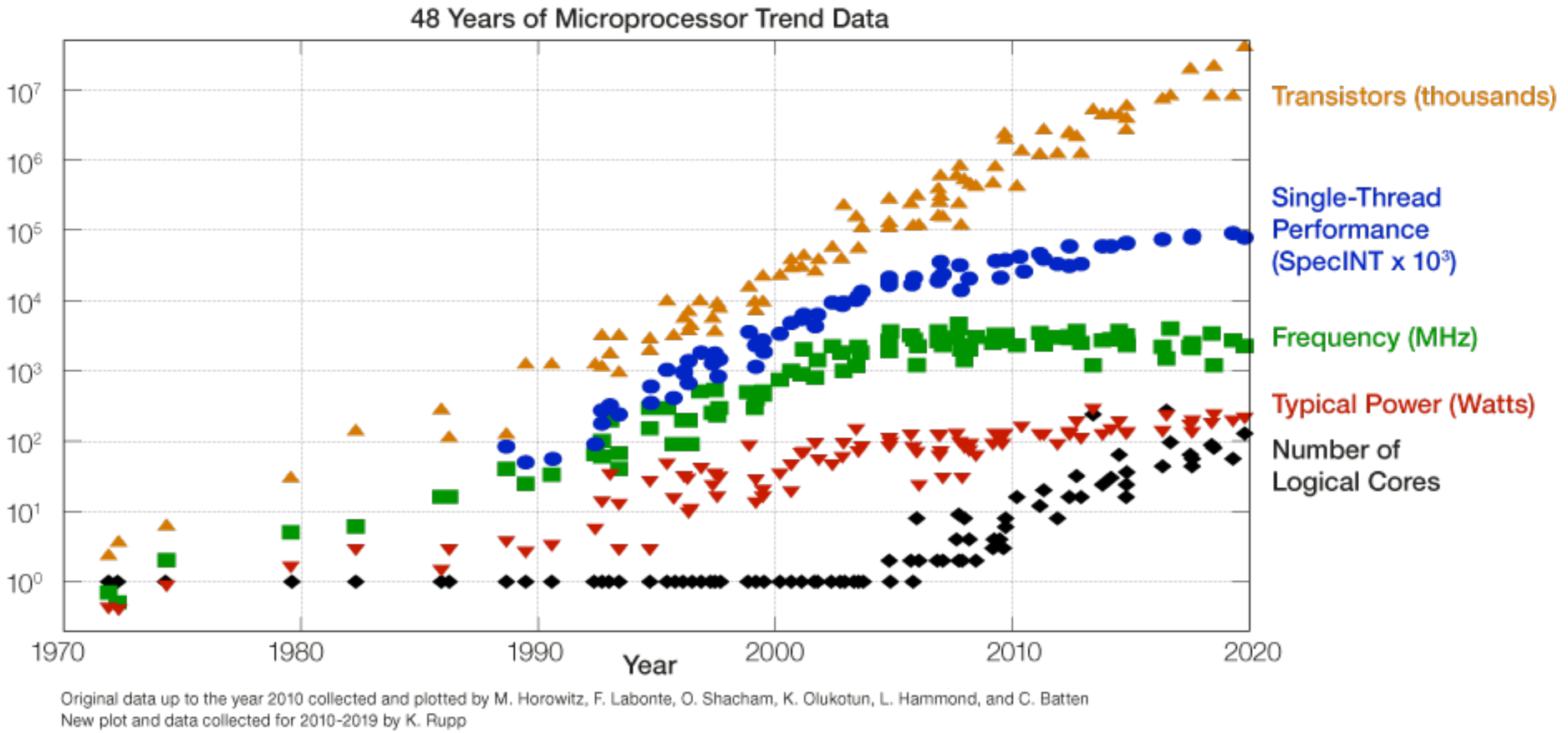
```
 1 struct DataUnsorted {
 2     char a;
 3     char __pad1[7];
 4     long long b;
 5     bool c;
 6     char __pad2[7];
 7     double d;
 8     int e;
 9     char __pad3[4];
10 };
11 // sizeof(DataUnsorted) == 40
```

```
1 struct DataSorted {
2     long long b;
3     double d;
4     int e;
5     char a;
6     bool c;
7     char __pad1[2];
8 };
9 // sizeof(DataSorted) == 24
```

Parallelization



Parallelization



So far, everything we did used a single thread (instruction-level parallelism).
Fully utilizing a modern CPU requires parallelization!

Amdahl's Law

Usually, only a part of your program is parallelizable.

Let p be that fraction, and N be the number of processors.

Then, the best runtime reduction by parallelization you can hope for is

$$\frac{1}{(1-p)+\frac{p}{N}}.$$

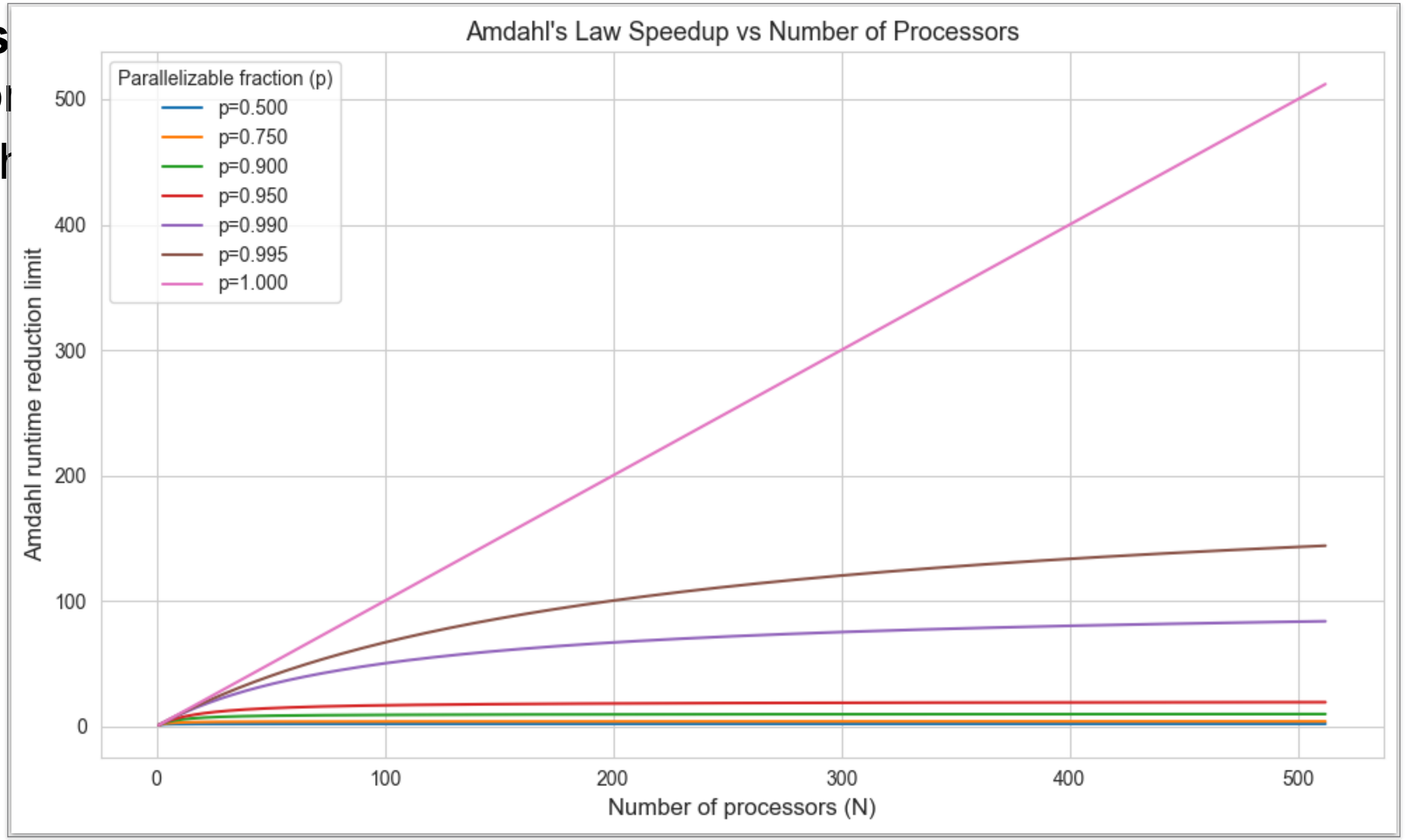
Theoretical Limits: Amdahl's Law

Amdahl's

Usually, o

Let p be t

Then, the



Parallelization Hurdles

Parallelization Hurdles

Two fundamental issues: synchronization and **shared resources.**

Parallelization Hurdles

Two fundamental issues: synchronization and **shared resources.**

Synchronization:

- Parallel (potential) access of shared data (except for read-only access)

Parallelization Hurdles

Two fundamental issues: synchronization and **shared resources.**

Synchronization:

- Parallel (potential) access of shared data (except for read-only access)
- Requires locking or lock-free data structures & atomic operations

Parallelization Hurdles

Two fundamental issues: synchronization and **shared resources.**

Synchronization:

- Parallel (potential) access of shared data (except for read-only access)
- Requires locking or lock-free data structures & atomic operations
- Such data is typically much more expensive to access than with a single thread

Parallelization Hurdles

Two fundamental issues: synchronization and **shared resources.**

Synchronization:

- Parallel (potential) access of shared data (except for read-only access)
- Requires locking or lock-free data structures & atomic operations
- Such data is typically much more expensive to access than with a single thread

Shared resources:

- Even completely independent threads that shared no data share some resources

Parallelization Hurdles

Two fundamental issues: synchronization and **shared resources.**

Synchronization:

- Parallel (potential) access of shared data (except for read-only access)
- Requires locking or lock-free data structures & atomic operations
- Such data is typically much more expensive to access than with a single thread

Shared resources:

- Even completely independent threads that shared no data share some resources
- Power supply and thermal resources

Parallelization Hurdles

Two fundamental issues: synchronization and **shared resources.**

Synchronization:

- Parallel (potential) access of shared data (except for read-only access)
- Requires locking or lock-free data structures & atomic operations
- Such data is typically much more expensive to access than with a single thread

Shared resources:

- Even completely independent threads that shared no data share some resources
- Power supply and thermal resources
- Memory (both capacity and bandwidth)

Parallelization Hurdles

Two fundamental issues: synchronization and **shared resources.**

Synchronization:

- Parallel (potential) access of shared data (except for read-only access)
- Requires locking or lock-free data structures & atomic operations
- Such data is typically much more expensive to access than with a single thread

Shared resources:

- Even completely independent threads that shared no data share some resources
- Power supply and thermal resources
- Memory (both capacity and bandwidth)
- Storage, Network, …

Shared Resource: Memory Bandwidth

Shared Resource: Memory Bandwidth

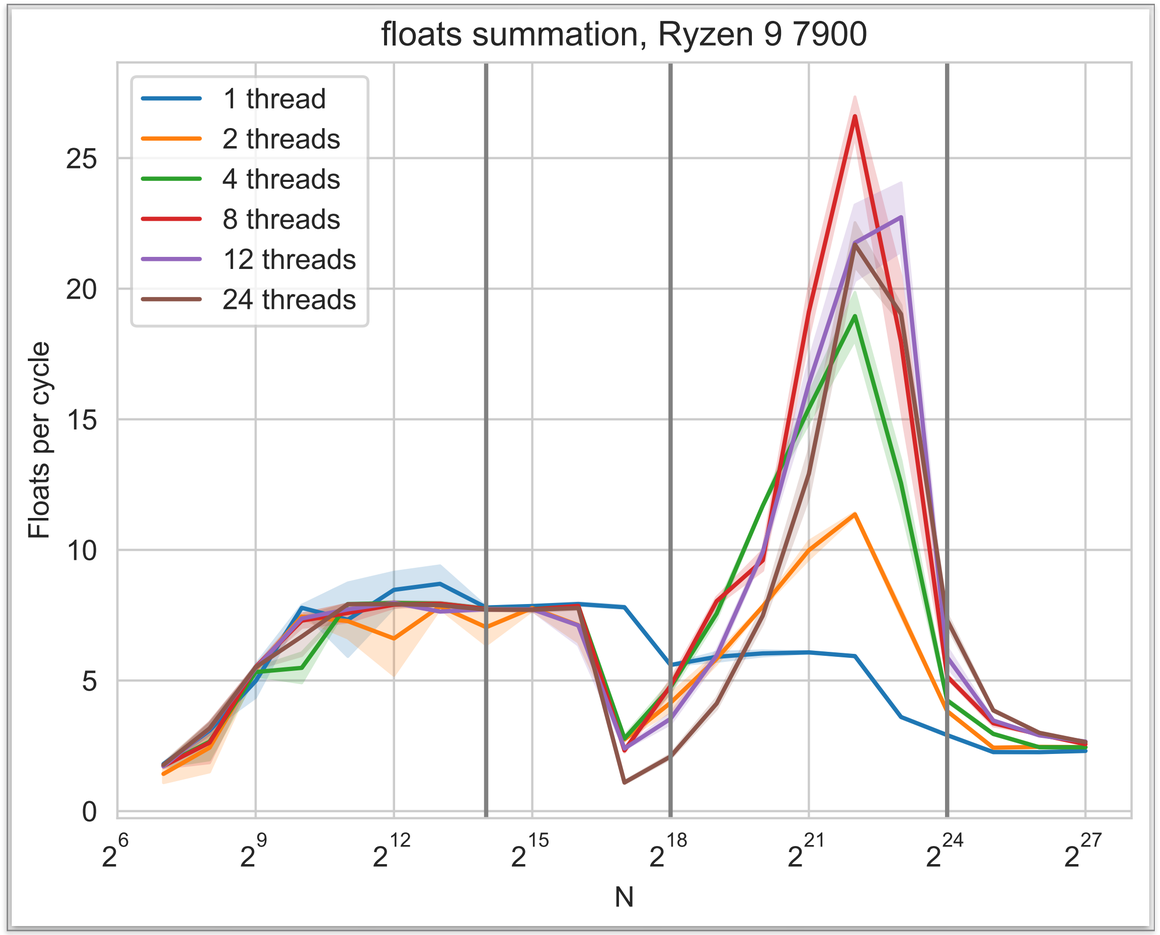
Let’s parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)

Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

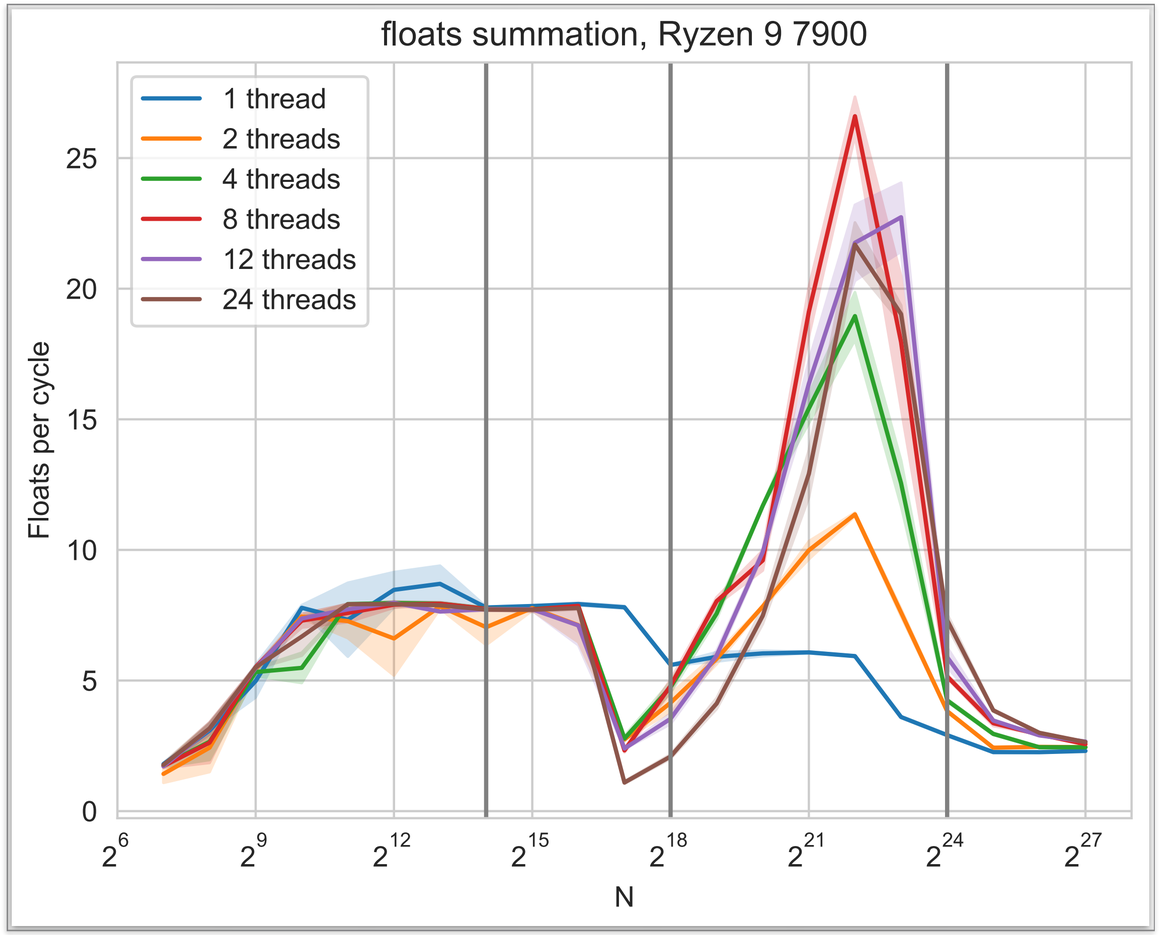
Synchronization: each thread works on contiguous chunk of the input array (trivial!)



Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)



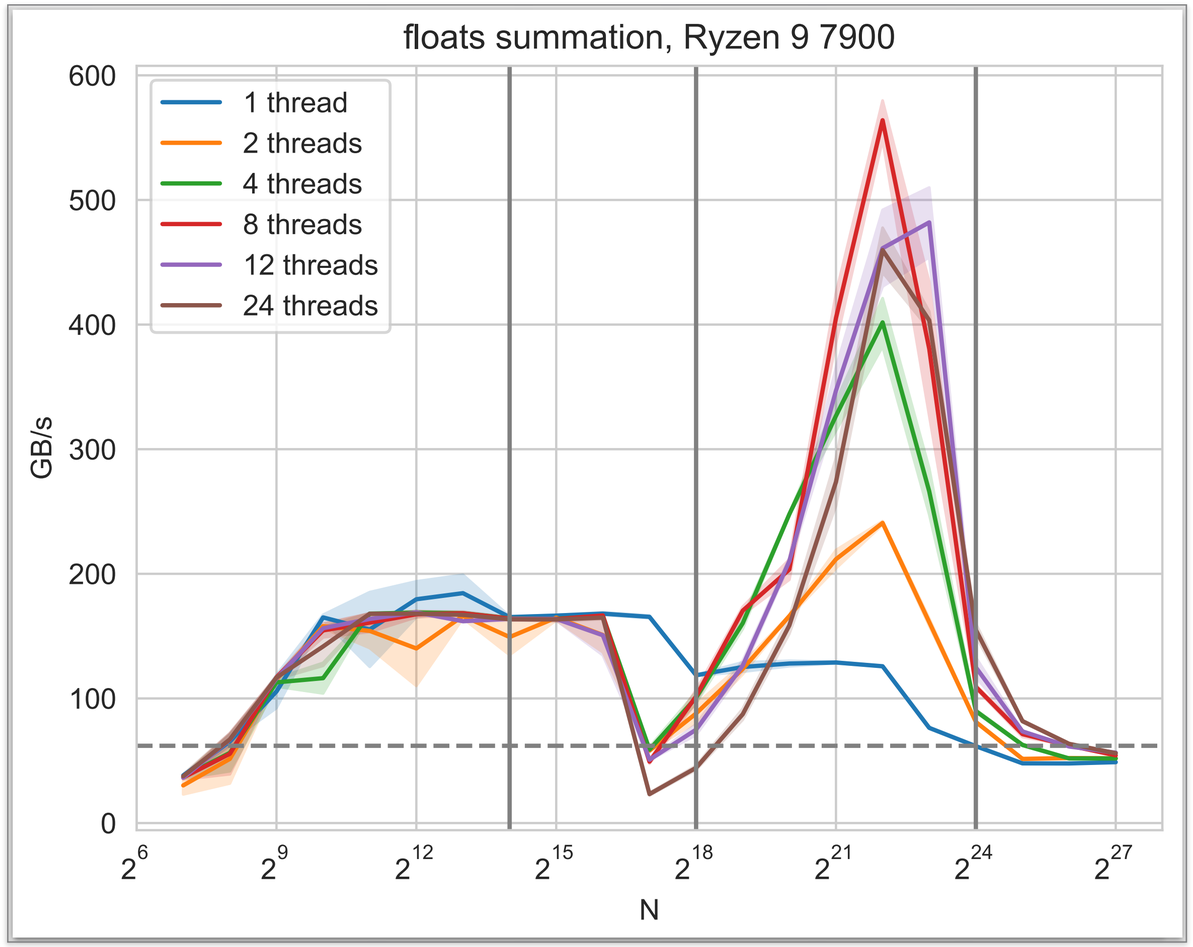
At the peak: $5\times$ faster

Almost no effect for large buffers

Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)



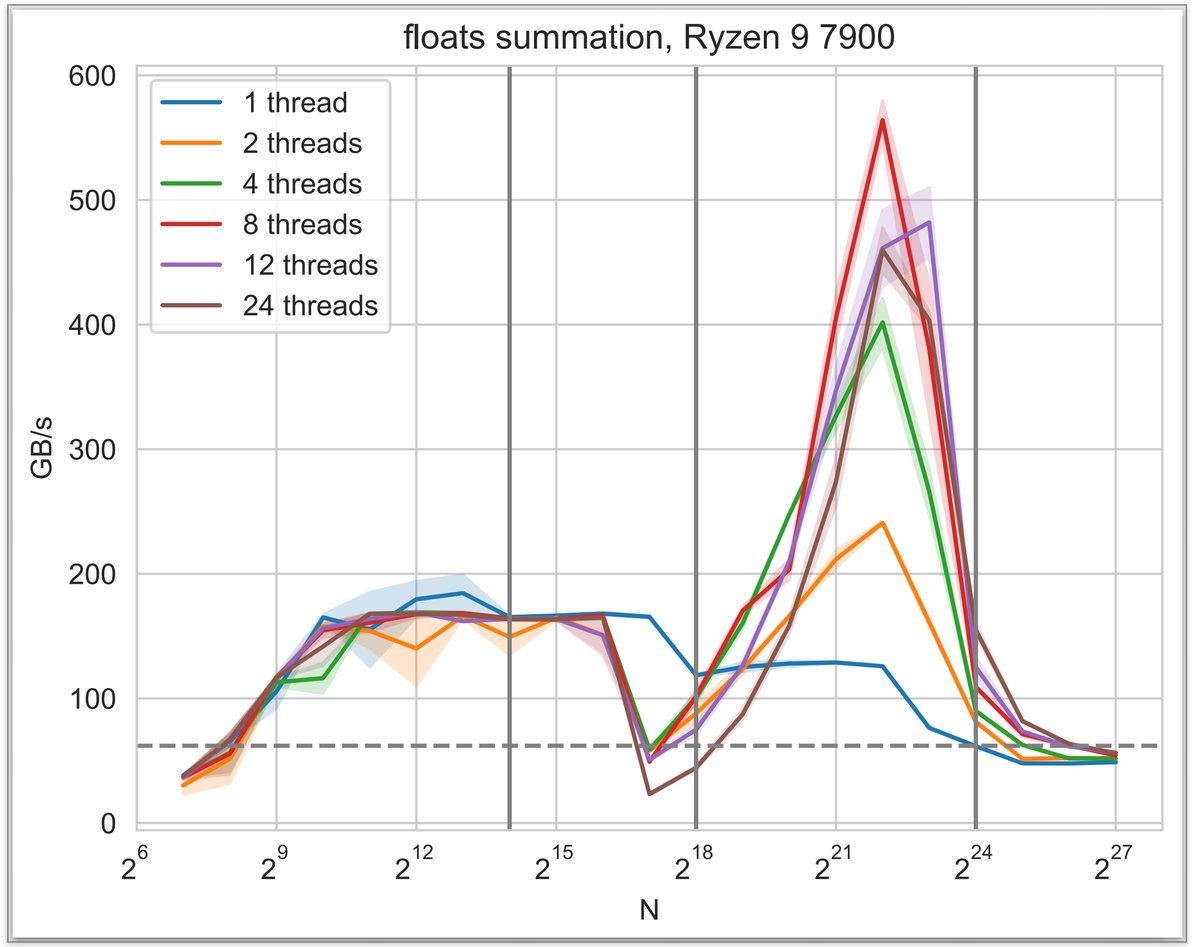
At the peak: $5\times$ faster

Almost no effect for large buffers

Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)



At the peak: $5\times$ faster

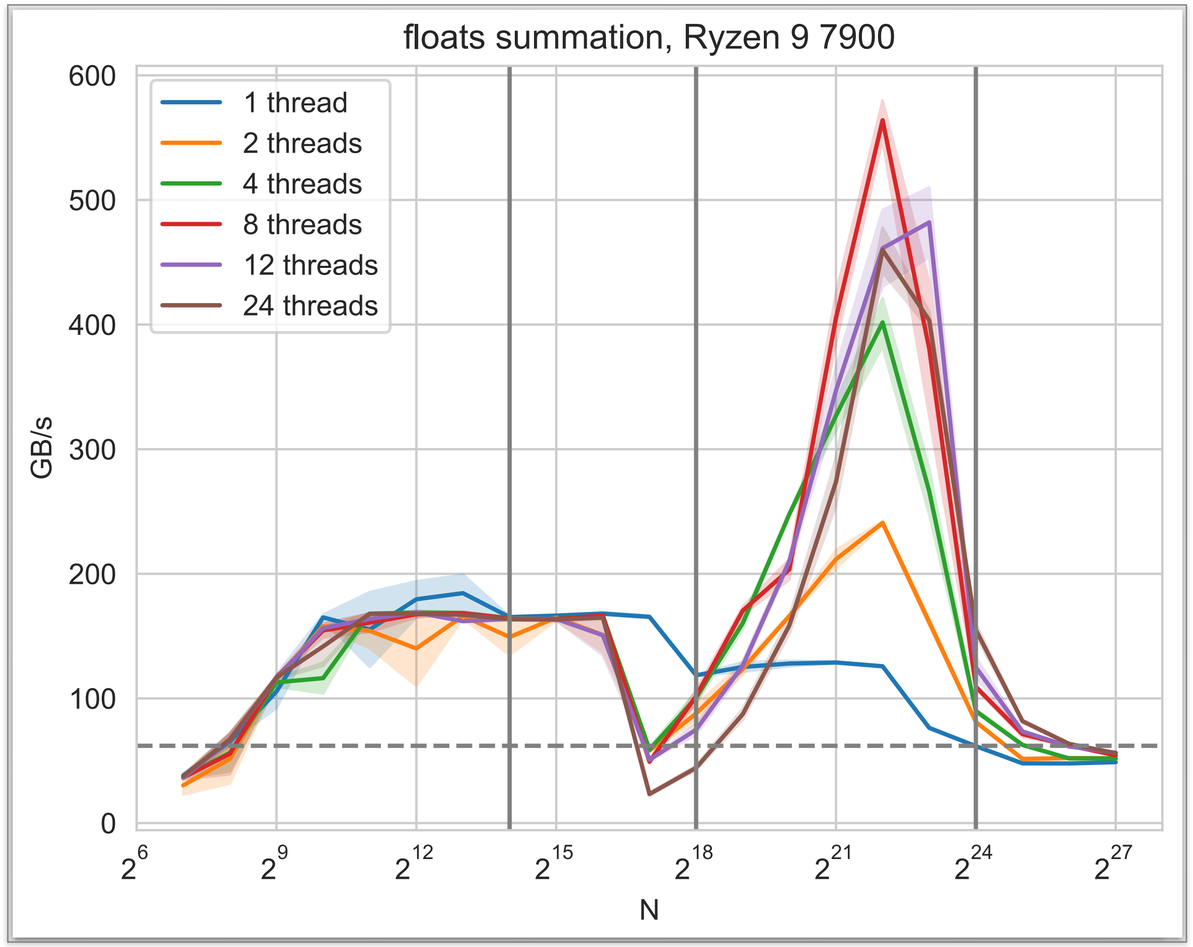
Almost no effect for large buffers

Very close to the memory bandwidth limit;

Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)



At the peak: $5\times$ faster

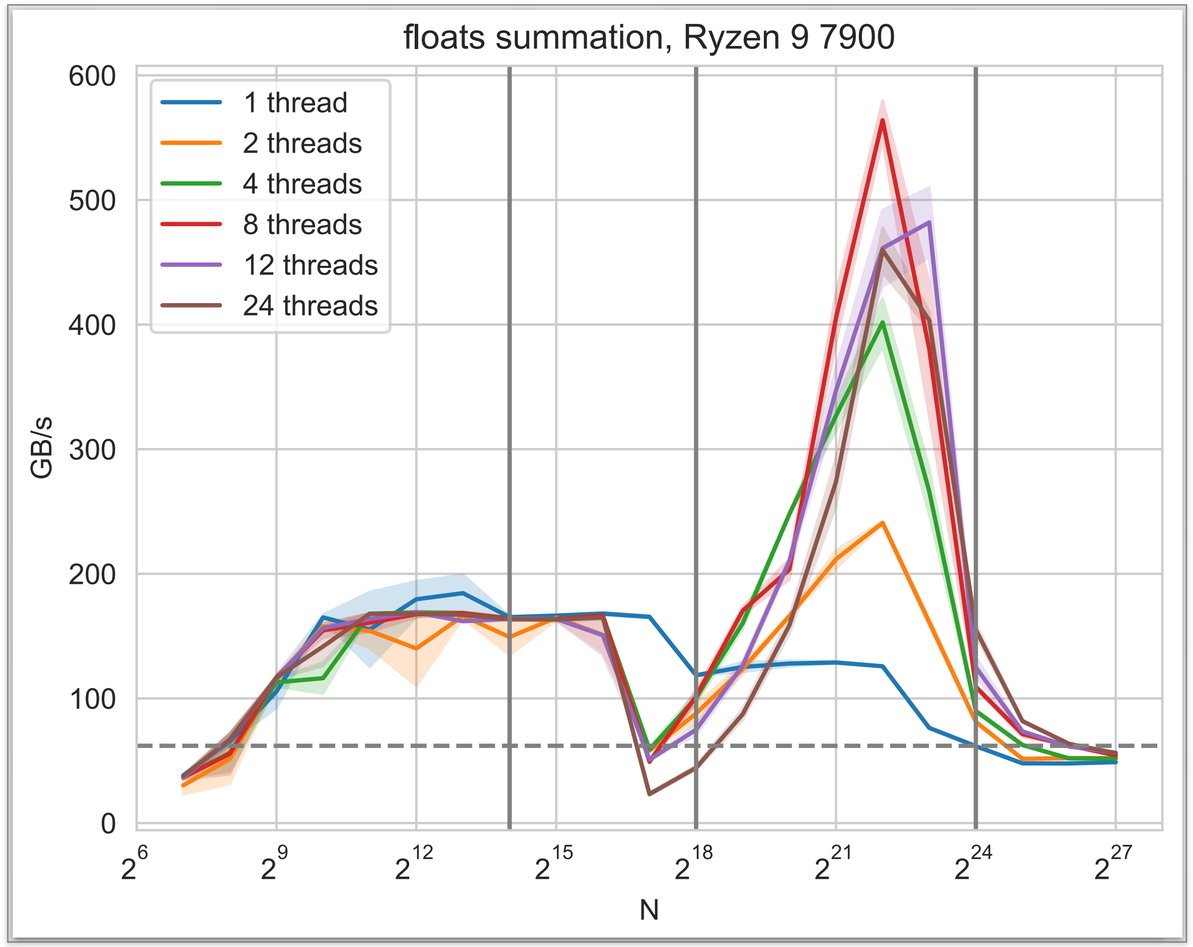
Almost no effect for large buffers

Very close to the memory bandwidth limit;
need not much more than one thread for this!

Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)



At the peak: $5\times$ faster

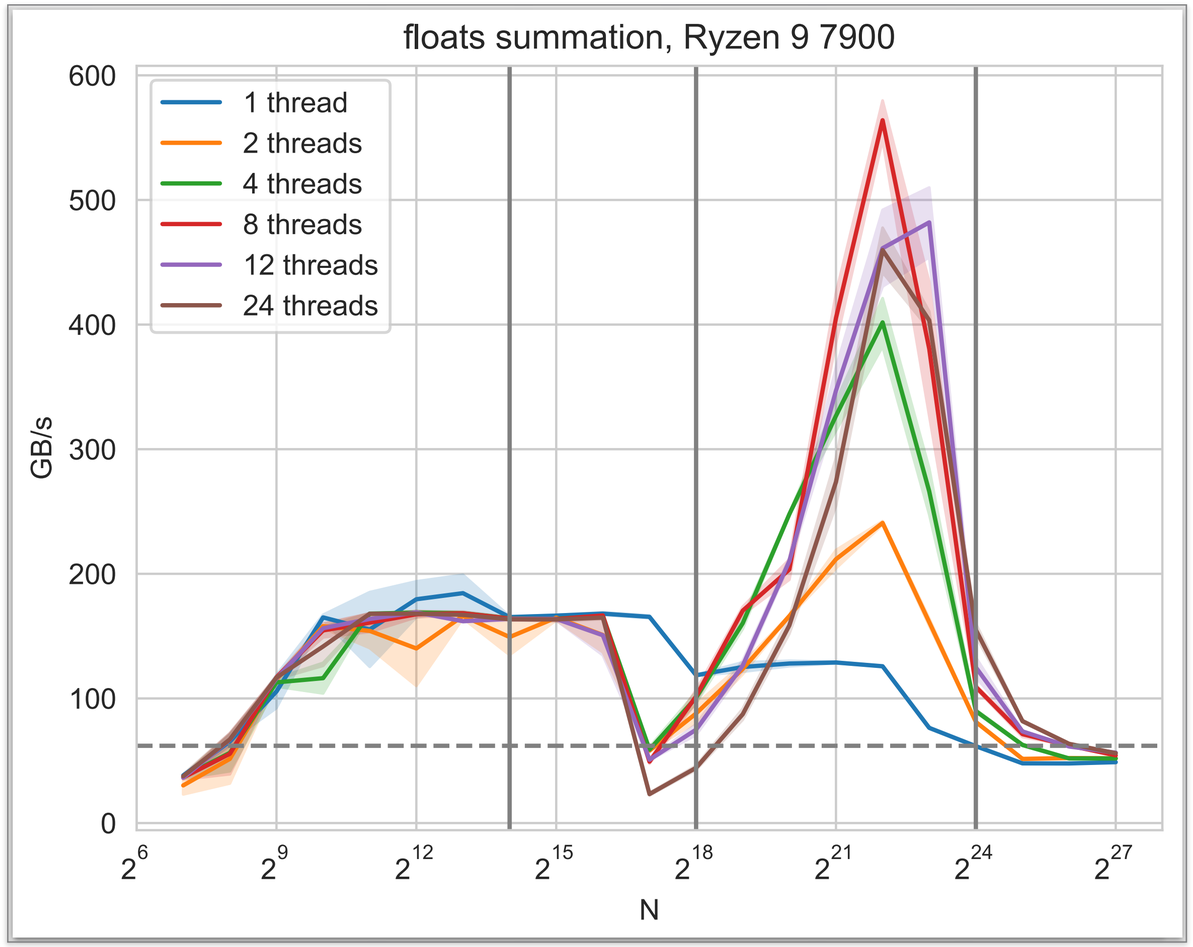
Almost no effect for large buffers

Very close to the memory bandwidth limit;
need not much more than one thread for this!
Likely for:

Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)



At the peak: $5\times$ faster

Almost no effect for large buffers

Very close to the memory bandwidth limit;
need not much more than one thread for this!

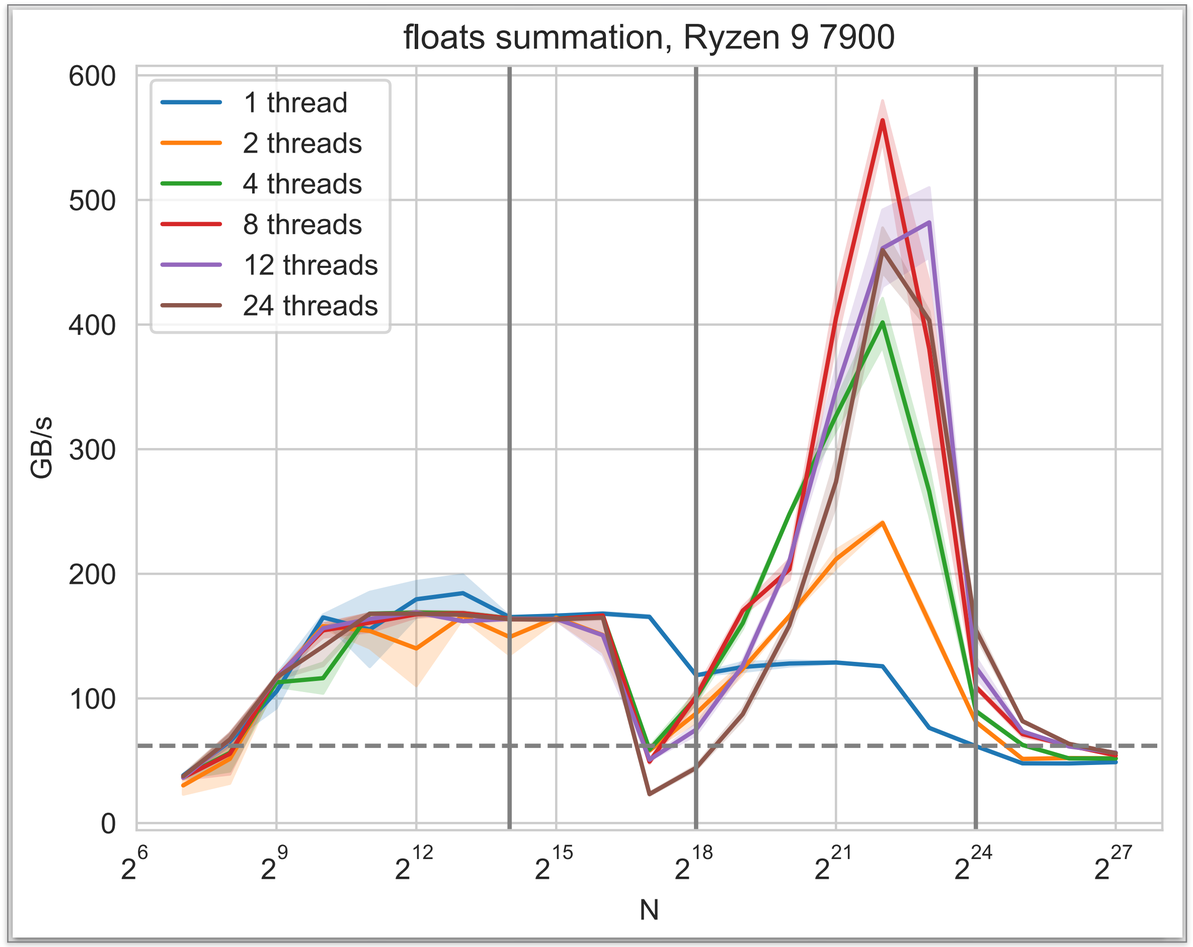
Likely for:

- reading lots of memory once,

Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)



At the peak: $5\times$ faster

Almost no effect for large buffers

Very close to the memory bandwidth limit;
need not much more than one thread for this!

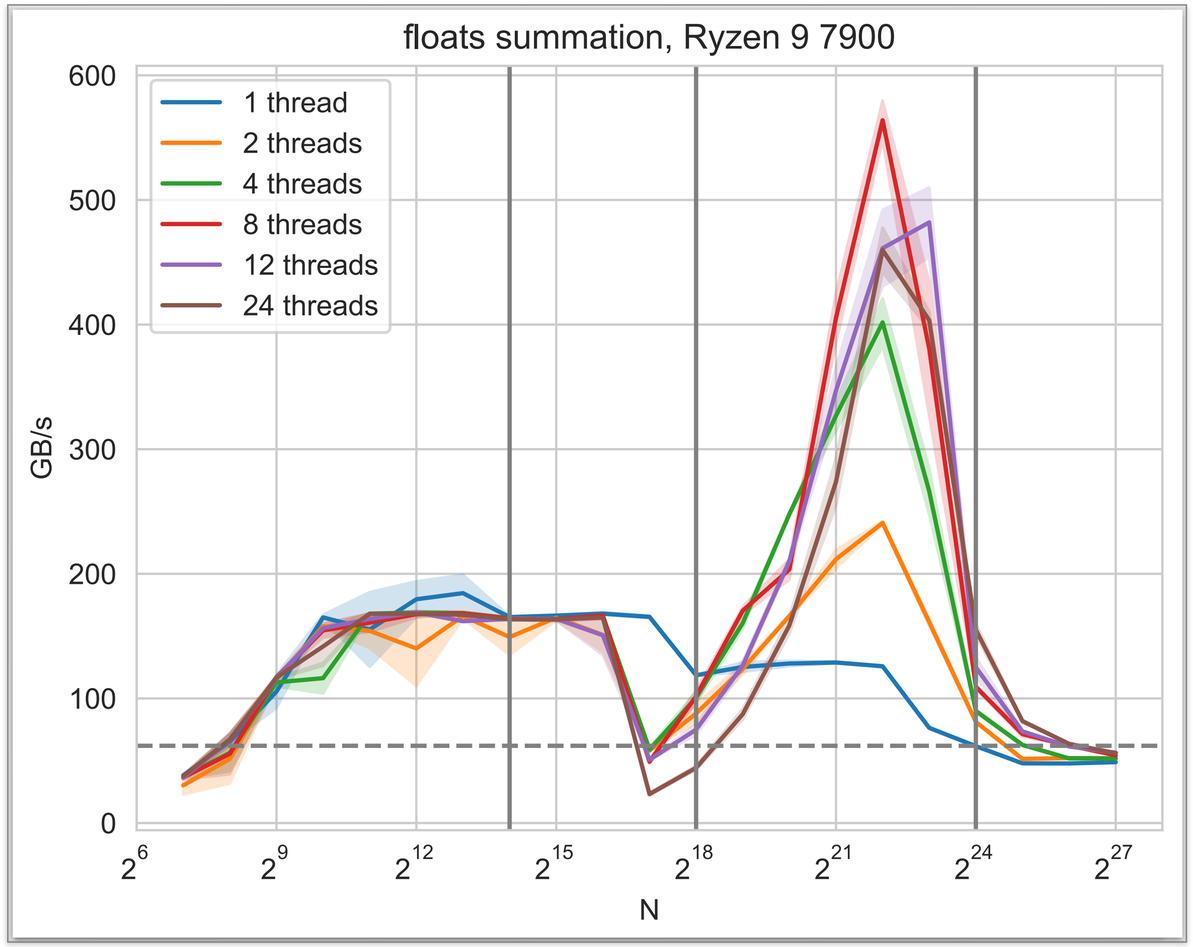
Likely for:

- reading lots of memory once,
- predictable linear scan pattern,

Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)



At the peak: $5\times$ faster

Almost no effect for large buffers

Very close to the memory bandwidth limit;
need not much more than one thread for this!

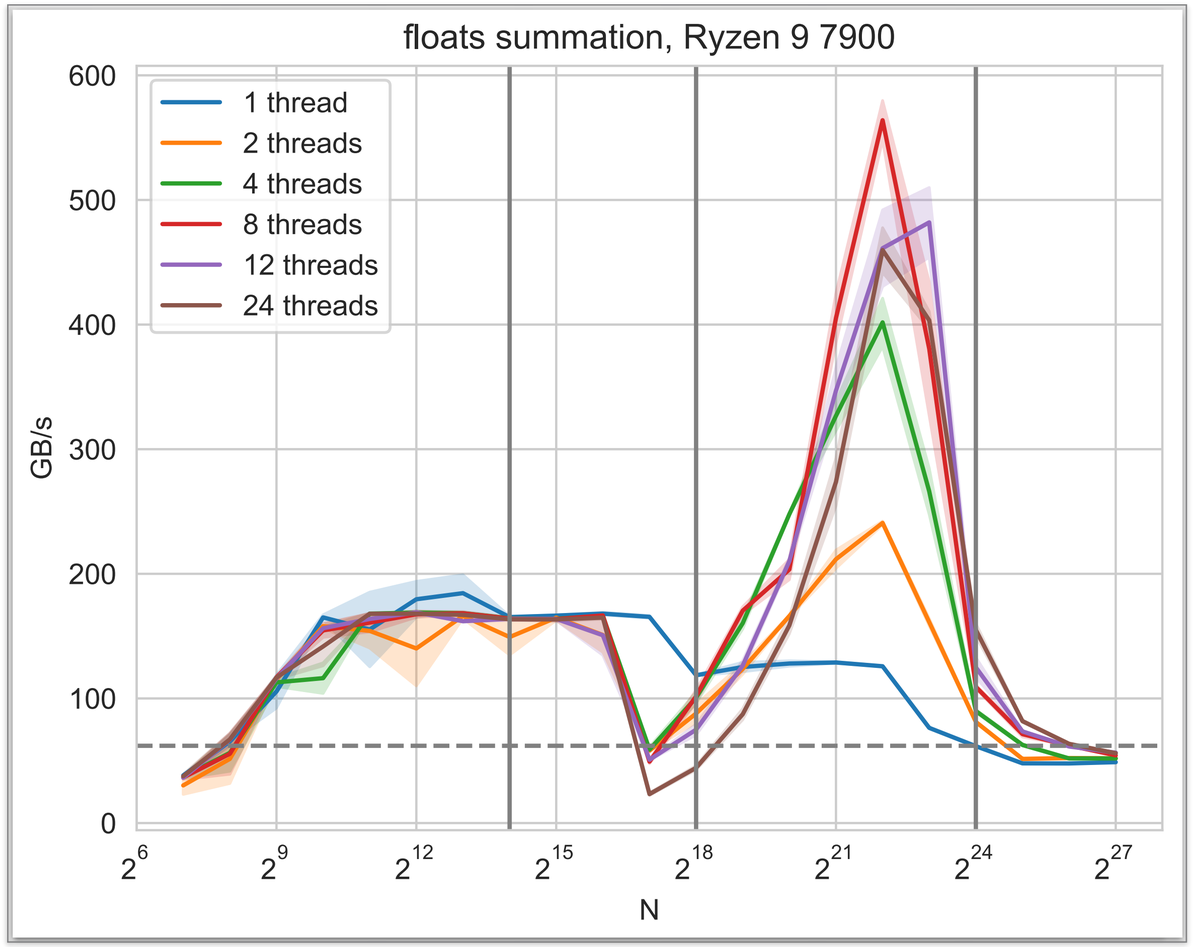
Likely for:

- reading lots of memory once,
- predictable linear scan pattern,
- significantly more than available cache,

Shared Resource: Memory Bandwidth

Let's parallelize our summation function (thread pool + manual SIMD/fallback).

Synchronization: each thread works on contiguous chunk of the input array (trivial!)



At the peak: $5\times$ faster

Almost no effect for large buffers

Very close to the memory bandwidth limit;
need not much more than one thread for this!

Likely for:

- reading lots of memory once,
- predictable linear scan pattern,
- significantly more than available cache,
- relatively light computational load.

Synchronization

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost
- Waiting threads

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost
- Waiting threads

Typical sources:

- Distribute work

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost
- Waiting threads

Typical sources:

- Distribute work
- Signaling (termination, new inputs, …)

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost
- Waiting threads

Typical sources:

- Distribute work
- Signaling (termination, new inputs, …)
- Shared data structures

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost
- Waiting threads

Typical sources:

- Distribute work
- Signaling (termination, new inputs, …)
- Shared data structures

Strategies:

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost
- Waiting threads

Typical sources:

- Distribute work
- Signaling (termination, new inputs, …)
- Shared data structures

Strategies:

- Have each thread work on its own chunk and combine results at the end

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost
- Waiting threads

Typical sources:

- Distribute work
- Signaling (termination, new inputs, …)
- Shared data structures

Strategies:

- Have each thread work on its own chunk and combine results at the end
- Keep critical sections short

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost
- Waiting threads

Typical sources:

- Distribute work
- Signaling (termination, new inputs, …)
- Shared data structures

Strategies:

- Have each thread work on its own chunk and combine results at the end
- Keep critical sections short
- Distribute sufficiently large tasks — balance with uneven workload distribution

Synchronization

Synchronization: only adds costs, so the goal is to keep it to a minimum.

- Direct locking/atomic operation cost
- Waiting threads

Typical sources:

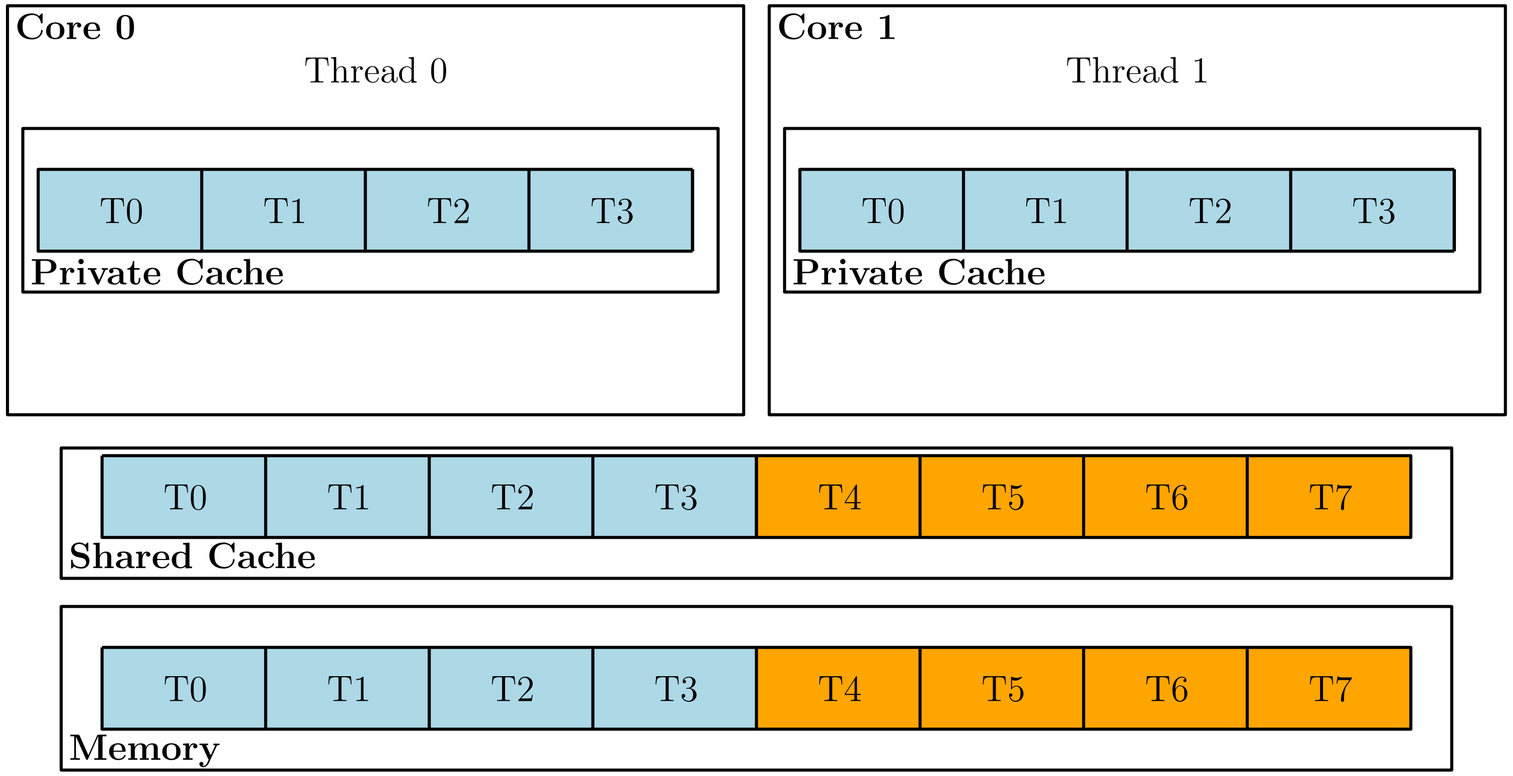
- Distribute work
- Signaling (termination, new inputs, …)
- Shared data structures

Strategies:

- Have each thread work on its own chunk and combine results at the end
- Keep critical sections short
- Distribute sufficiently large tasks — balance with uneven workload distribution
- Tools such as TBB can help

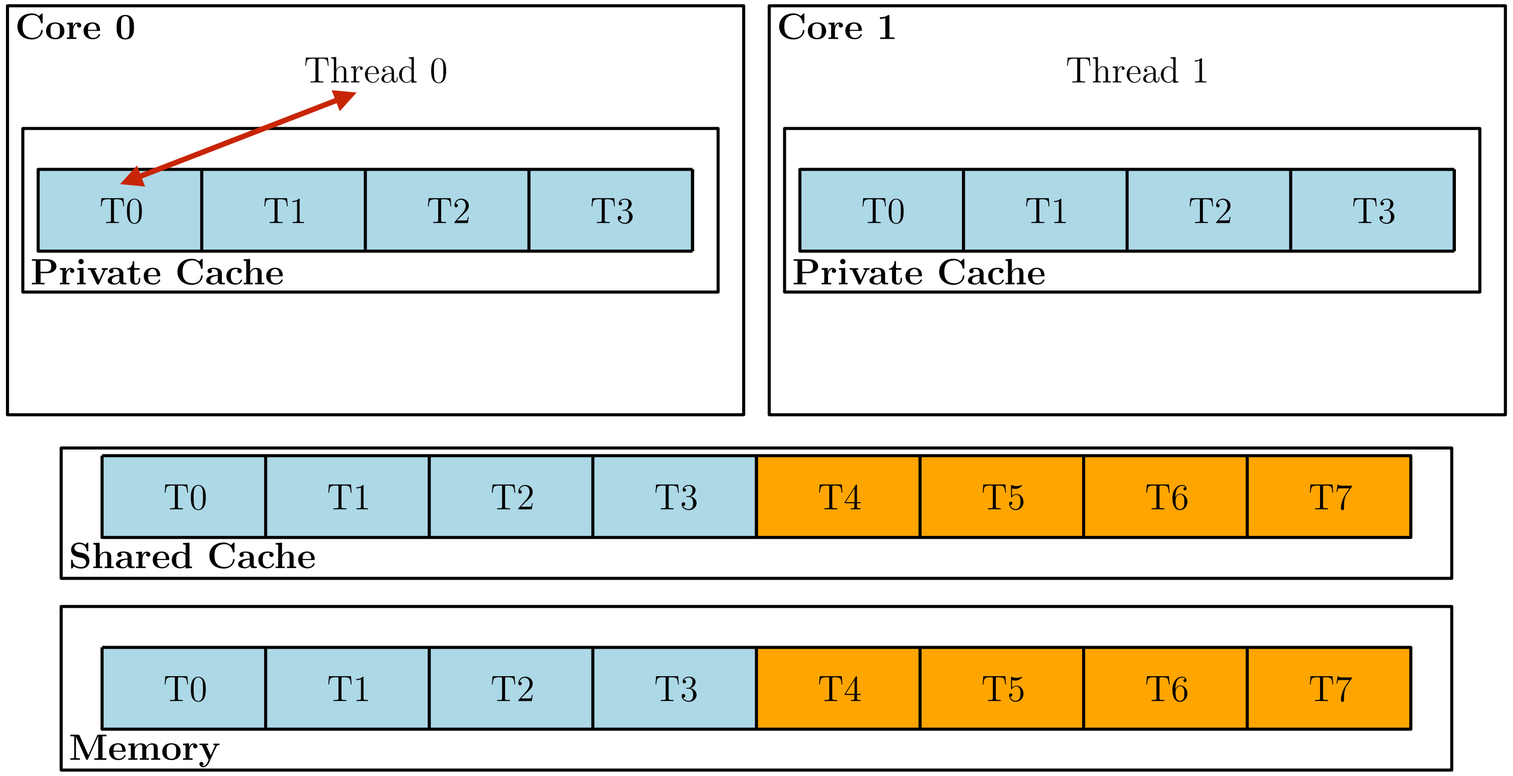
False Sharing

Encountering synchronization costs without synchronization/shared data:



False Sharing

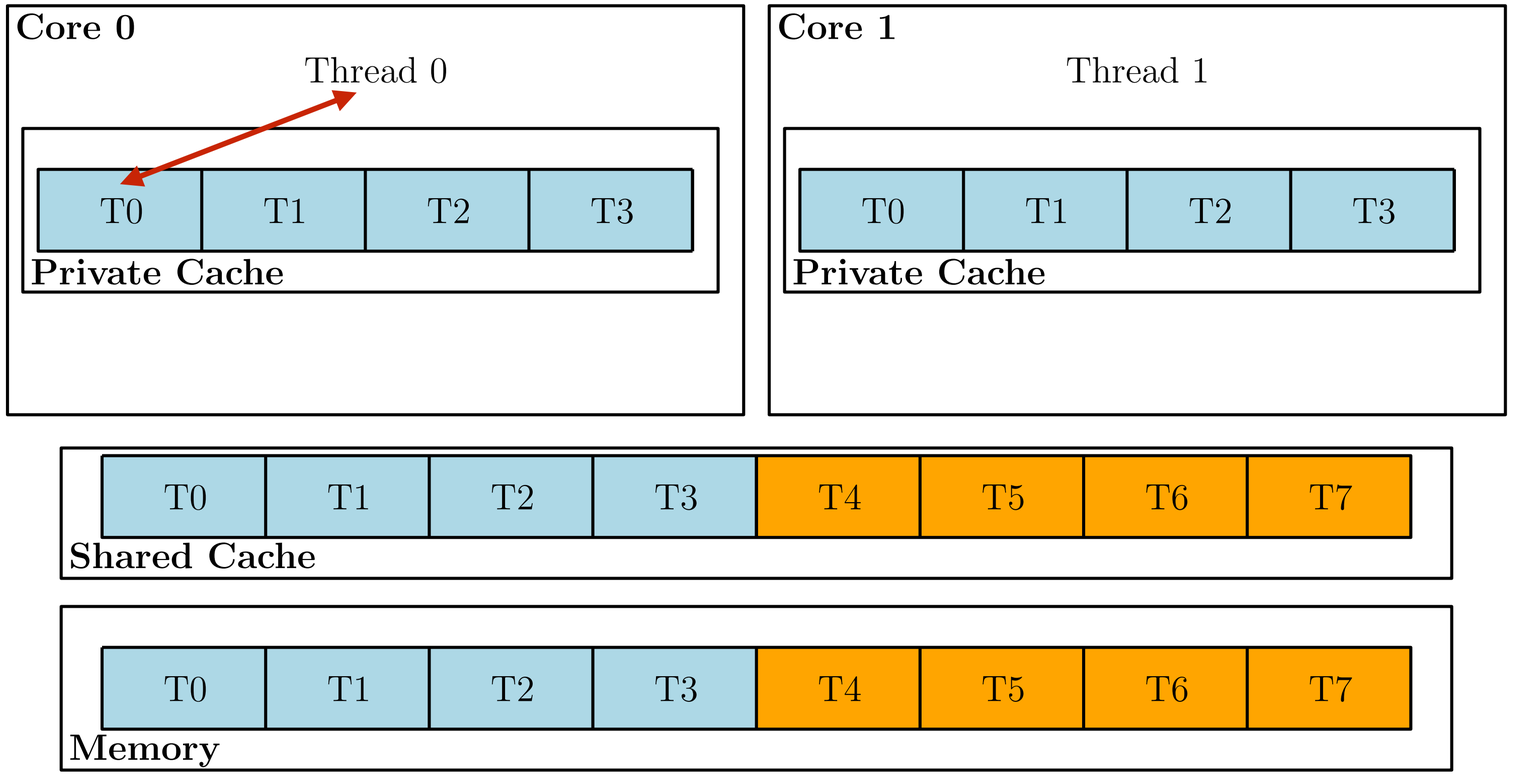
Encountering synchronization costs without synchronization/shared data:



- Thread 0 writes `T0`

False Sharing

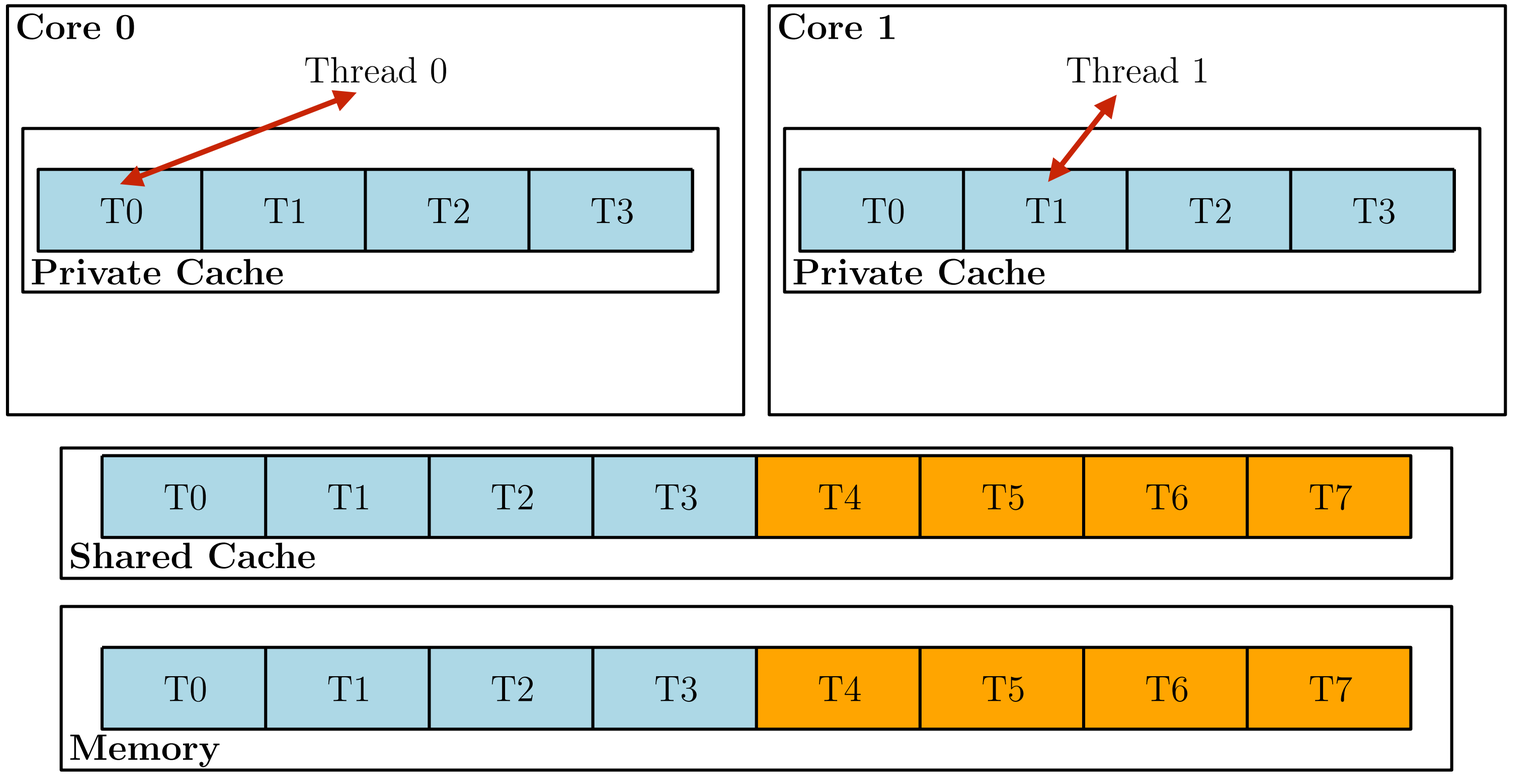
Encountering synchronization costs without synchronization/shared data:



- Thread 0 writes `T0`
- Invalidates copy on Core 1

False Sharing

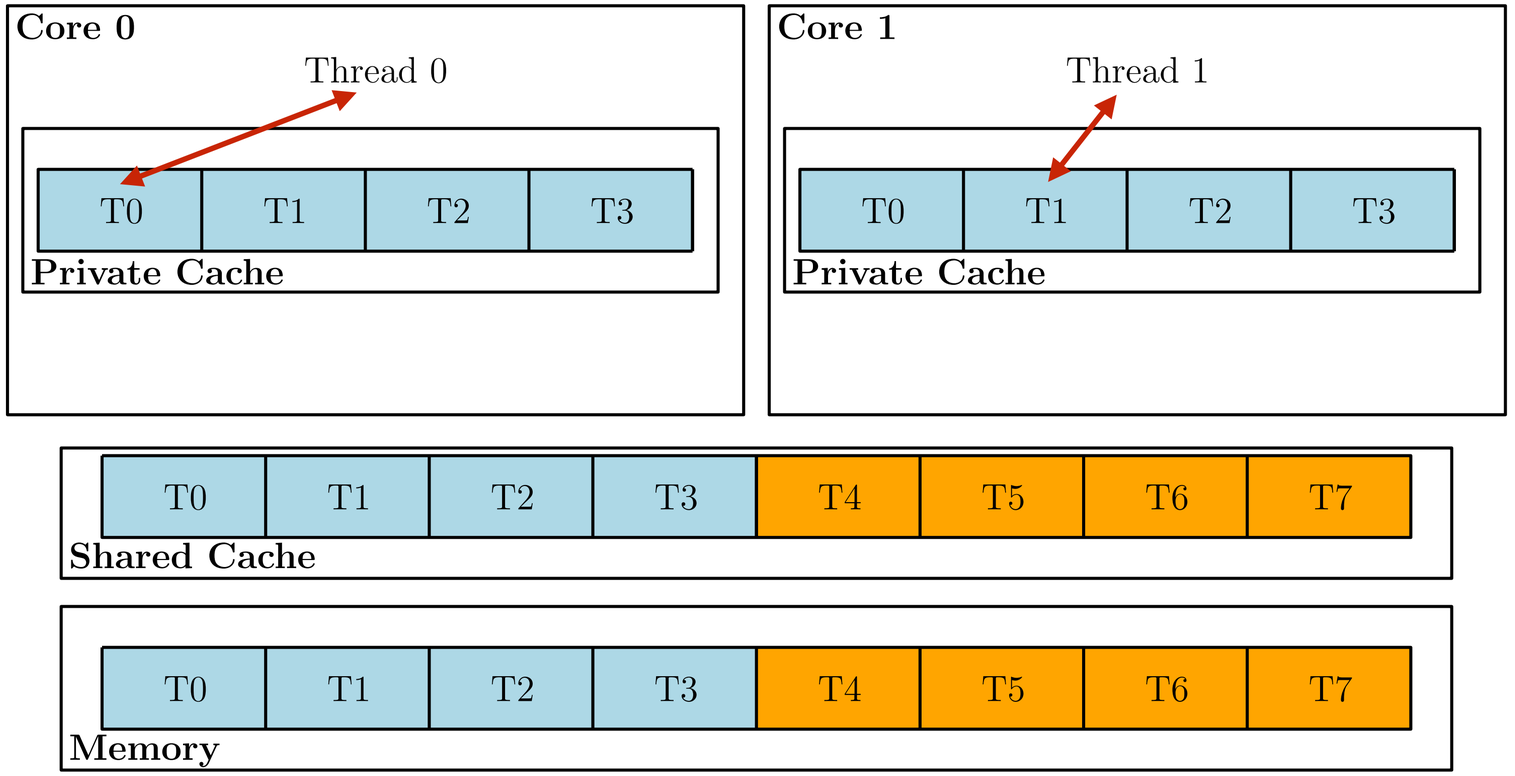
Encountering synchronization costs without synchronization/shared data:



- Thread 0 writes `T0`
- Invalidates copy on Core 1
- Thread 1 writes `T1`
- Invalidates copy on Core 0

False Sharing

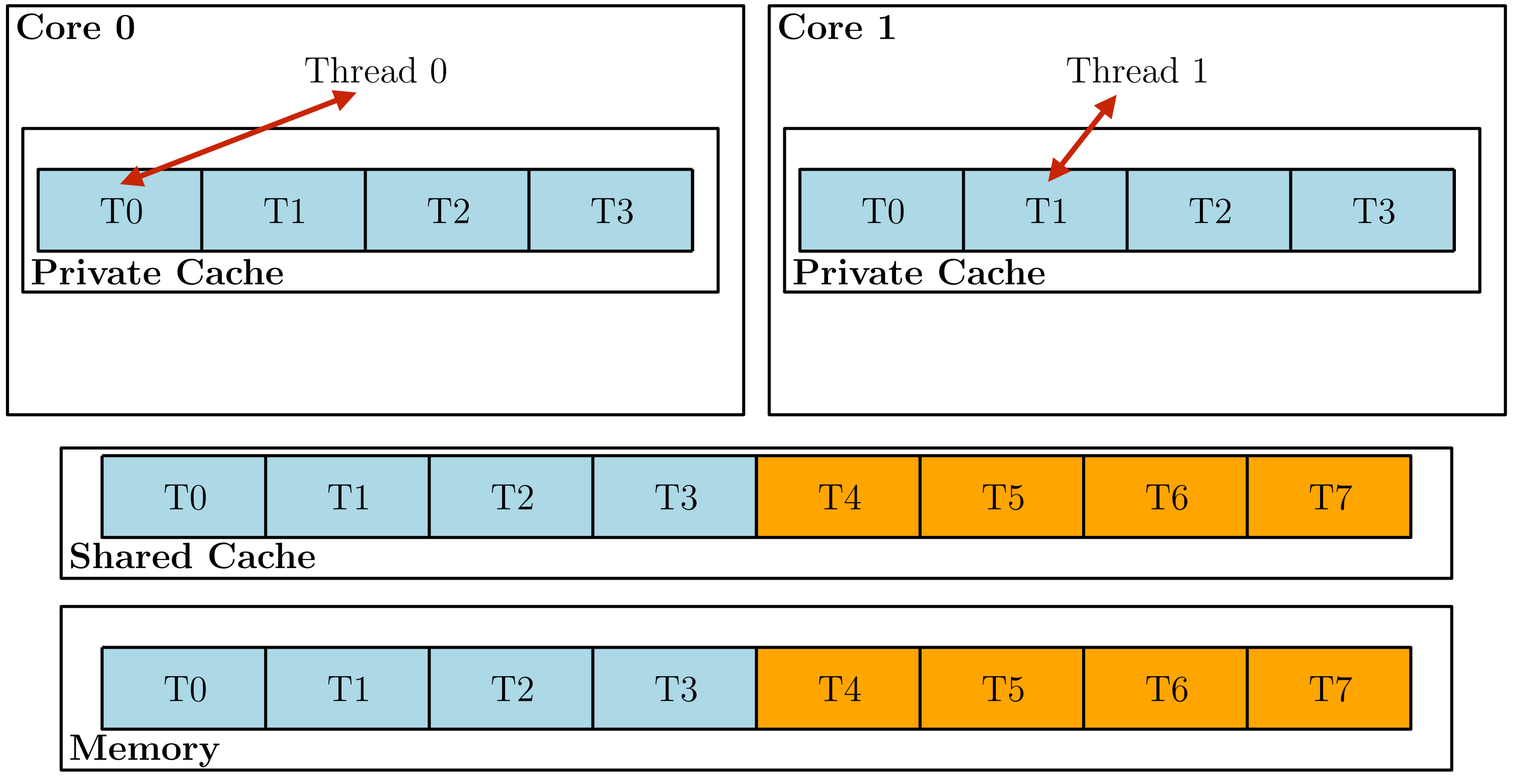
Encountering synchronization costs without synchronization/shared data:



- Thread 0 writes `T0`
- Invalidates copy on Core 1
- Thread 1 writes `T1`
- Invalidates copy on Core 0
- …

False Sharing

Encountering synchronization costs without synchronization/shared data:

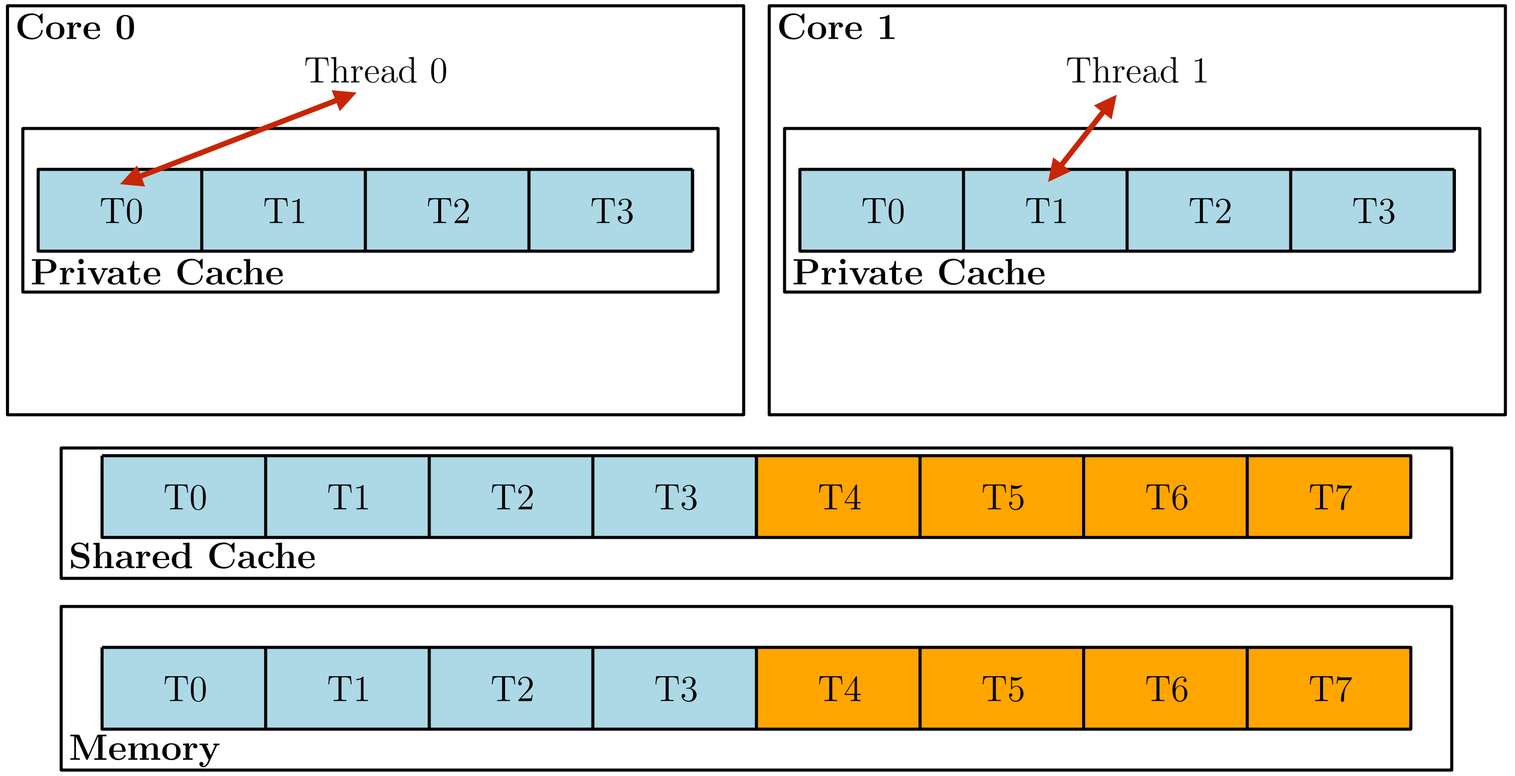


- Thread 0 writes `T0`
- Invalidates copy on Core 1
- Thread 1 writes `T1`
- Invalidates copy on Core 0
- …

- Synchronization overhead

False Sharing

Encountering synchronization costs without synchronization/shared data:

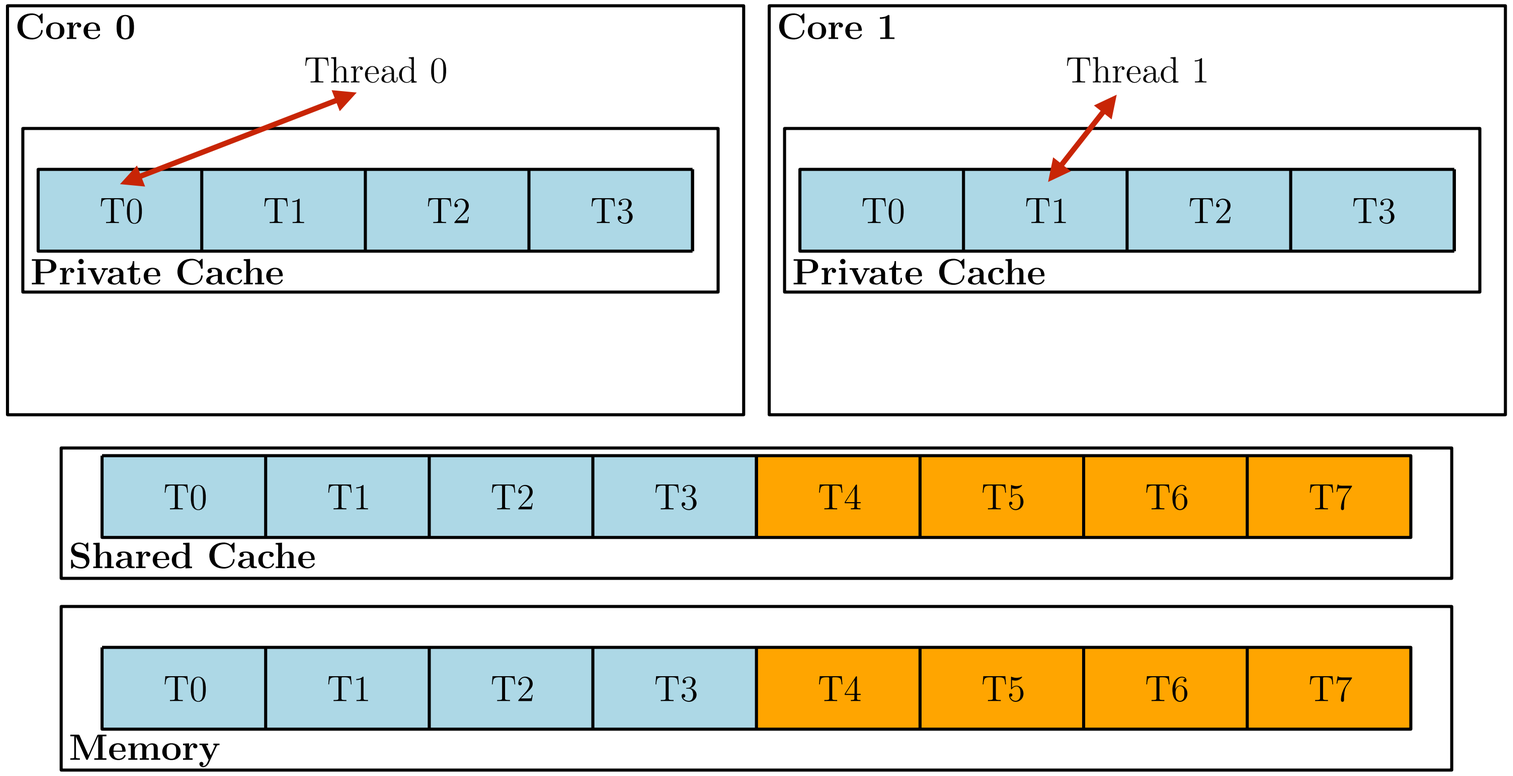


- Thread 0 writes `T0`
- Invalidates copy on Core 1
- Thread 1 writes `T1`
- Invalidates copy on Core 0
- …

- Synchronization overhead
- Longer access latencies

False Sharing

Encountering synchronization costs without synchronization/shared data:

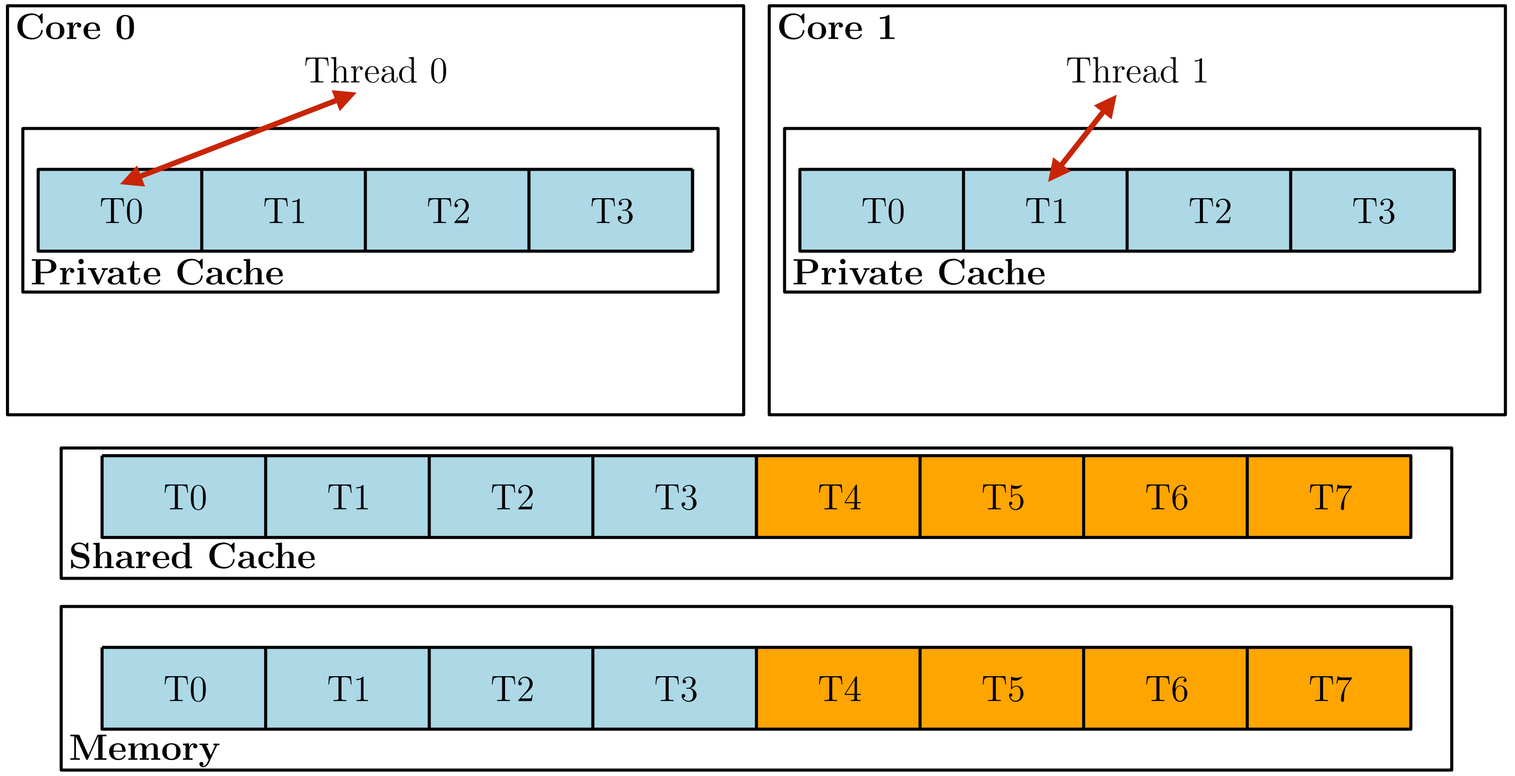


- Thread 0 writes `T0`
- Invalidates copy on Core 1
- Thread 1 writes `T1`
- Invalidates copy on Core 0
- …

- Synchronization overhead
- Longer access latencies
- Mis-speculation overhead

False Sharing

Encountering synchronization costs without synchronization/shared data:



- Thread 0 writes `T0`
- Invalidates copy on Core 1
- Thread 1 writes `T1`
- Invalidates copy on Core 0
- …

- Synchronization overhead
- Longer access latencies
- Mis-speculation overhead
- **Correct results!**

False Sharing

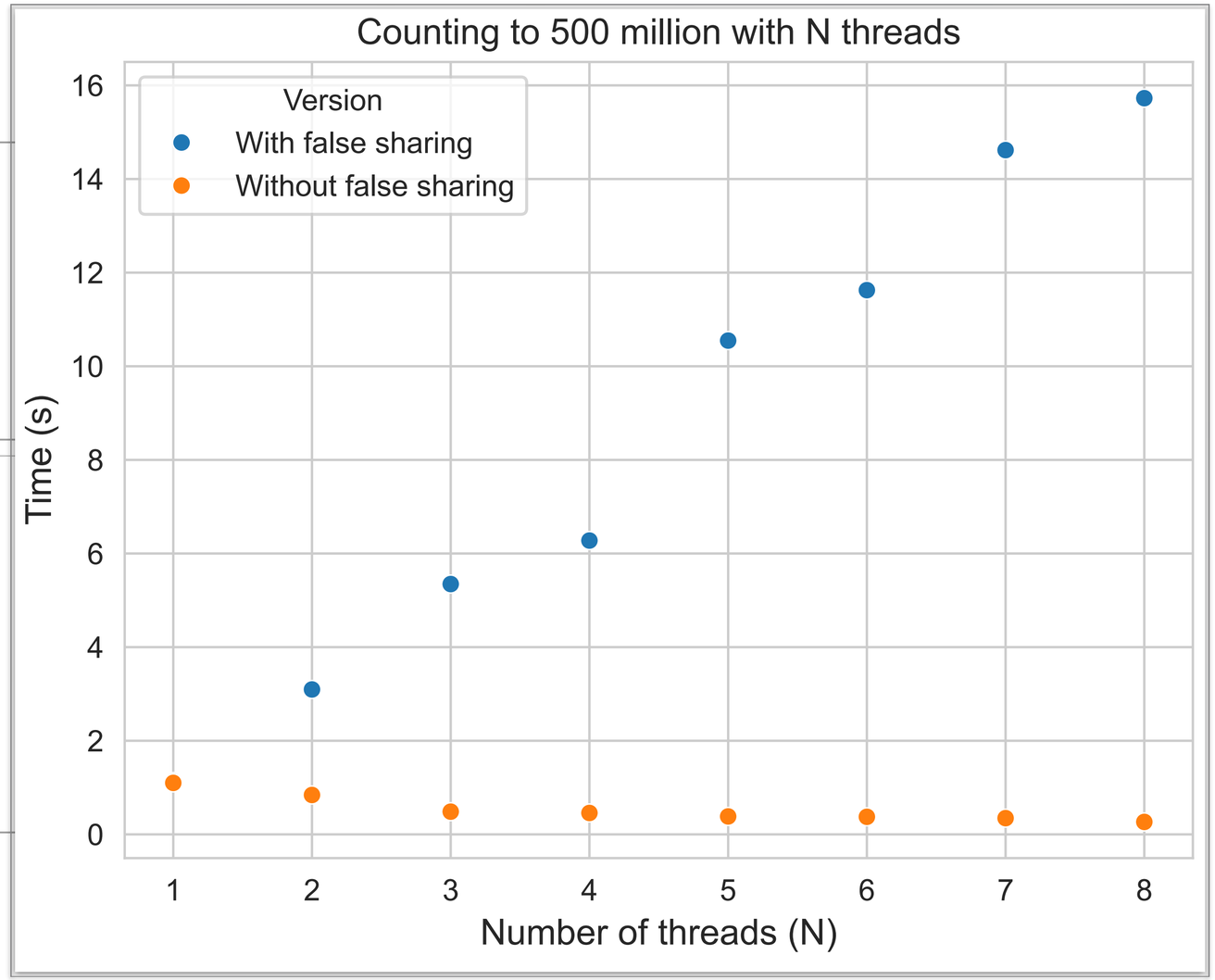
```cpp
struct ThreadOutputs {
    std::int32_t outputs[N];
};
void count_to(std::int32_t K) {
    // create N threads; thread i counts up
    // outputs[i] until, in total, K is reached
}
```

```cpp
struct ThreadOutputs {
    struct Output { alignas(64) std::int32_t x; };
    // each x is on its own cache line
    Output outputs[N];
};
void count_to(std::int32_t K) {
    // create N threads; thread i counts up
    // outputs[i].x until, in total, K is reached
}
```

False Sharing

```cpp
struct ThreadOutputs {
    std::int32_t outputs[N];
};
void count_to(std::int32_t K) {
    // create N threads; thread i counts up
    // outputs[i] until, in total, K is reached
}
```

```cpp
struct ThreadOutputs {
    struct Output { alignas(64) std::int32_t x; };
    // each x is on its own cache line
    Output outputs[N];
};
void count_to(std::int32_t K) {
    // create N threads; thread i counts up
    // outputs[i].x until, in total, K is reached
}
```



Measuring Multithreaded Performance

Measuring Multithreaded Performance

Classical measures:

- Instructions per Cycle (IPC)

Measuring Multithreaded Performance

Classical measures:

- Instructions per Cycle (IPC)
- CPU Utilization

Measuring Multithreaded Performance

Classical measures:

- Instructions per Cycle (IPC)
- CPU Utilization

Issues in multi-threaded scenarios:

- Spinning (busy-waiting) shows as high utilization

Measuring Multithreaded Performance

Classical measures:

- Instructions per Cycle (IPC)
- CPU Utilization

Issues in multi-threaded scenarios:

- Spinning (busy-waiting) shows as high utilization
- Often part of taking a lock (before an expensive context switch)

Measuring Multithreaded Performance

Classical measures:

- Instructions per Cycle (IPC)
- CPU Utilization

Issues in multi-threaded scenarios:

- Spinning (busy-waiting) shows as high utilization
- Often part of taking a lock (before an expensive context switch)

Replacement: Effective time

Measuring Multithreaded Performance

Classical measures:

- Instructions per Cycle (IPC)
- CPU Utilization

Issues in multi-threaded scenarios:

- Spinning (busy-waiting) shows as high utilization
- Often part of taking a lock (before an expensive context switch)

Replacement: Effective time

- Is a much dirtier, more heuristic measure; depends on interpreting profiling data

Measuring Multithreaded Performance

Classical measures:

- Instructions per Cycle (IPC)
- CPU Utilization

Issues in multi-threaded scenarios:

- Spinning (busy-waiting) shows as high utilization
- Often part of taking a lock (before an expensive context switch)

Replacement: Effective time

- Is a much dirtier, more heuristic measure; depends on interpreting profiling data

Other measures: synchronization or preemption wait time, spin time

Measuring Multithreaded Performance

Classical measures:

- Instructions per Cycle (IPC)
- CPU Utilization

Issues in multi-threaded scenarios:

- Spinning (busy-waiting) shows as high utilization
- Often part of taking a lock (before an expensive context switch)

Replacement: Effective time

- Is a much dirtier, more heuristic measure; depends on interpreting profiling data

Other measures: synchronization or preemption wait time, spin time

Goals: find expensive locks, detect balancing issues between producers/consumers, …

Low Level vs. High Level Parallelism

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Usually better: High-level parallelism.

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Usually better: High-level parallelism.

Examples:

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Usually better: High-level parallelism.

Examples:

- Schedule the subroutines/heuristics of a solver on the available threads

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Usually better: High-level parallelism.

Examples:

- Schedule the subroutines/heuristics of a solver on the available threads
- Run multiple copies of a heuristic with different seeds, take the best result

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Usually better: High-level parallelism.

Examples:

- Schedule the subroutines/heuristics of a solver on the available threads
- Run multiple copies of a heuristic with different seeds, take the best result
- Run different algorithms (or different parameters), stop when first algorithm is done

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Usually better: High-level parallelism.

Examples:

- Schedule the subroutines/heuristics of a solver on the available threads
- Run multiple copies of a heuristic with different seeds, take the best result
- Run different algorithms (or different parameters), stop when first algorithm is done

Potential problems:

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Usually better: High-level parallelism.

Examples:

- Schedule the subroutines/heuristics of a solver on the available threads
- Run multiple copies of a heuristic with different seeds, take the best result
- Run different algorithms (or different parameters), stop when first algorithm is done

Potential problems:

- Memory requirements depend on number of threads

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Usually better: High-level parallelism.

Examples:

- Schedule the subroutines/heuristics of a solver on the available threads
- Run multiple copies of a heuristic with different seeds, take the best result
- Run different algorithms (or different parameters), stop when first algorithm is done

Potential problems:

- Memory requirements depend on number of threads
- Can be bad for large inputs

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ algorithms, Intel TBB, …: support low-level parallelism.

Usually better: High-level parallelism.

Examples:

- Schedule the subroutines/heuristics of a solver on the available threads
- Run multiple copies of a heuristic with different seeds, take the best result
- Run different algorithms (or different parameters), stop when first algorithm is done

Potential problems:

- Memory requirements depend on number of threads
- Can be bad for large inputs
- May require a ‘split’ thread pool (number of threads per task/subsolver/…)

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ alg -level p

Usually

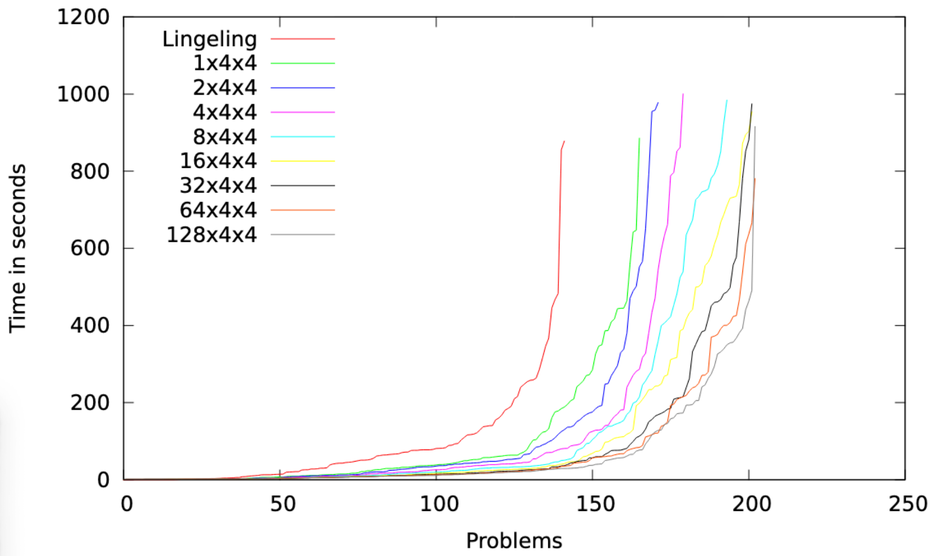
Exampl

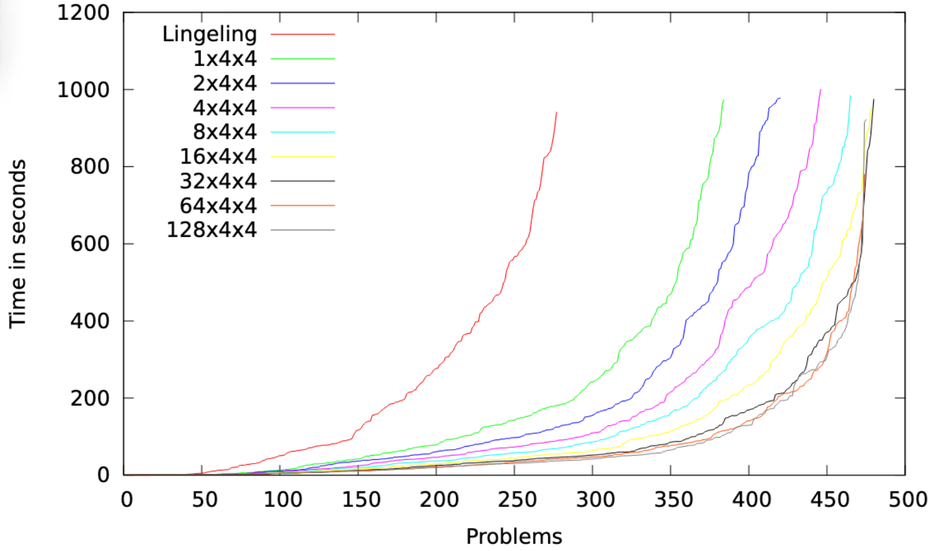
HordeSat: A Massively Parallel Portfolio SAT Solver

Tomáš Balyo, Peter Sanders, Carsten Sinz *

Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Experiments made using benchmarks from the application tracks of the 2011 and 2014 Sat Competitions [3] show that HordeSat can outperform state-of-the-art parallel SAT solvers on multiprocessor machines and is scalable on computer clusters with thousands of processors. Indeed, we even observe superlinear average speedup for difficult instances.





Potential problems:

- Memory requirements depend on number of thre
- Can be bad for large inputs
- May require a ‘split’ thread pool (number of threa

Low Level vs. High Level Parallelism

Parallel sum, sort, …: Low-level parallelism.

C++ alg -level p

Usually

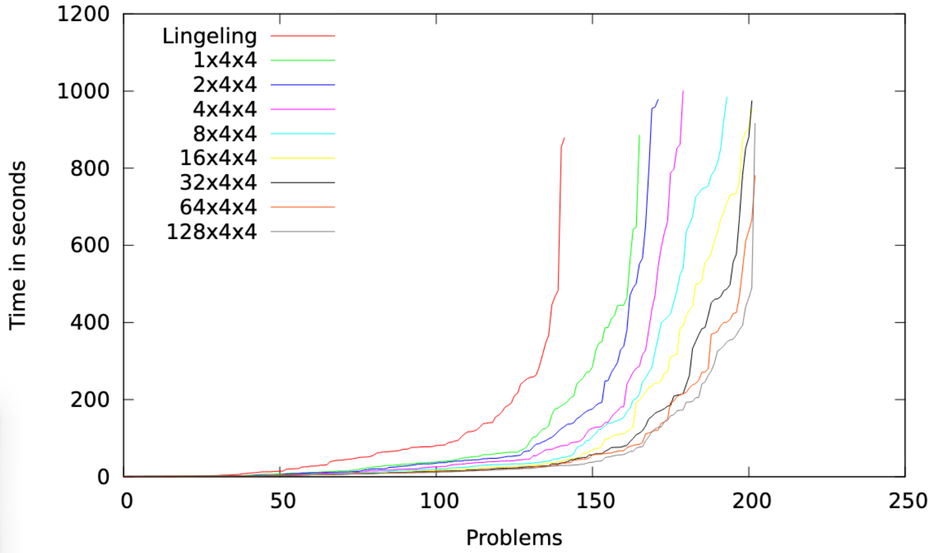
Exampl

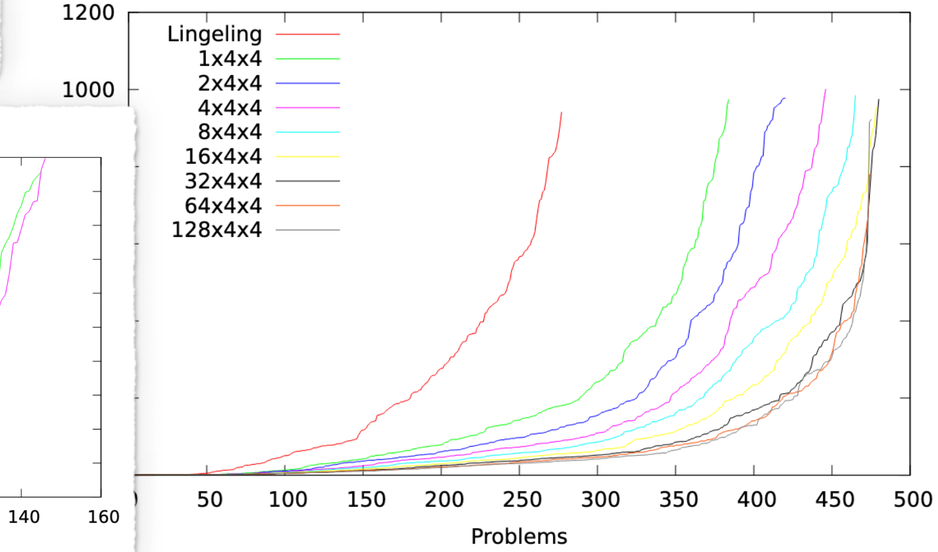
HordeSat: A Massively Parallel Portfolio SAT Solver

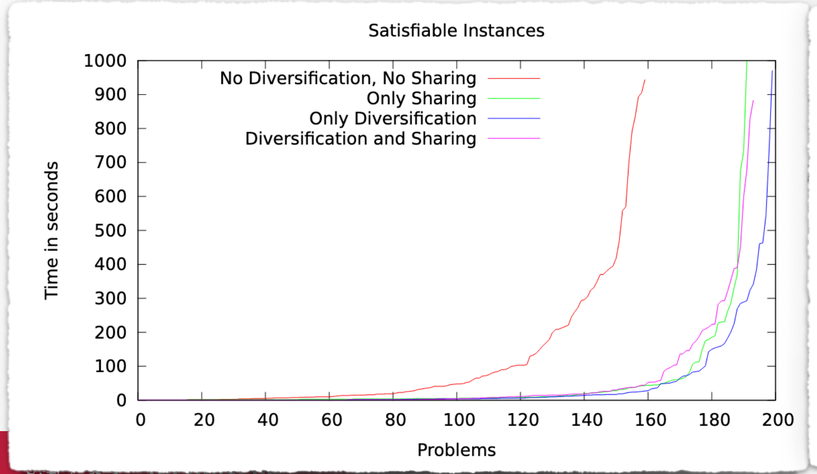
Tomáš Balyo, Peter Sanders, Carsten Sinz *

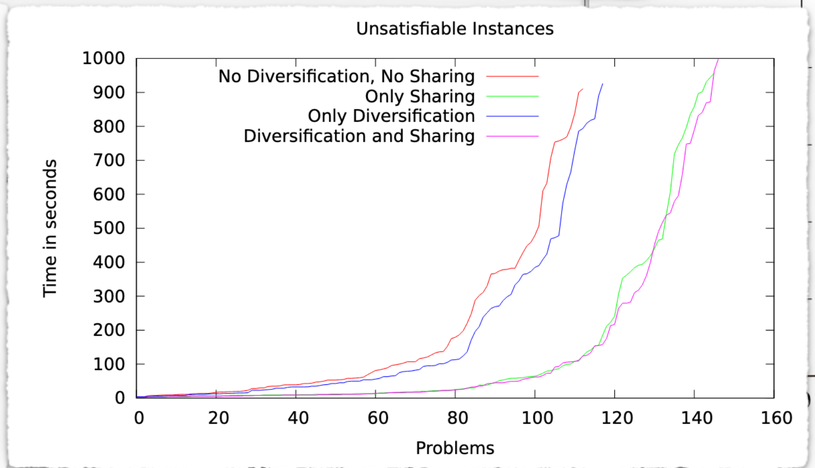
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Experiments made using benchmarks from the application tracks of the 2011 and 2014 Sat Competitions [3] show that HordeSat can outperform state-of-the-art parallel SAT solvers on multiprocessor machines and is scalable on computer clusters with thousands of processors. Indeed, we even observe superlinear average speedup for difficult instances.









Reading

Optional further reading for a deeper dive:

- Bakhvalov, Denis. 2024.
  Performance Analysis and Tuning on Modern CPUs. 2nd (in progress).
- Drepper, Ulrich. 2007. What Every Programmer Should Know about Memory.
- Fog, Agner. 2024. Optimizing Software in C++:
  An Optimization Guide for Windows, Linux, and Mac Platforms.
- Hennessy, John L., and David A. Patterson. 2019. Computer Architecture: A Quantitative Approach. 6th ed. Morgan Kaufmann.