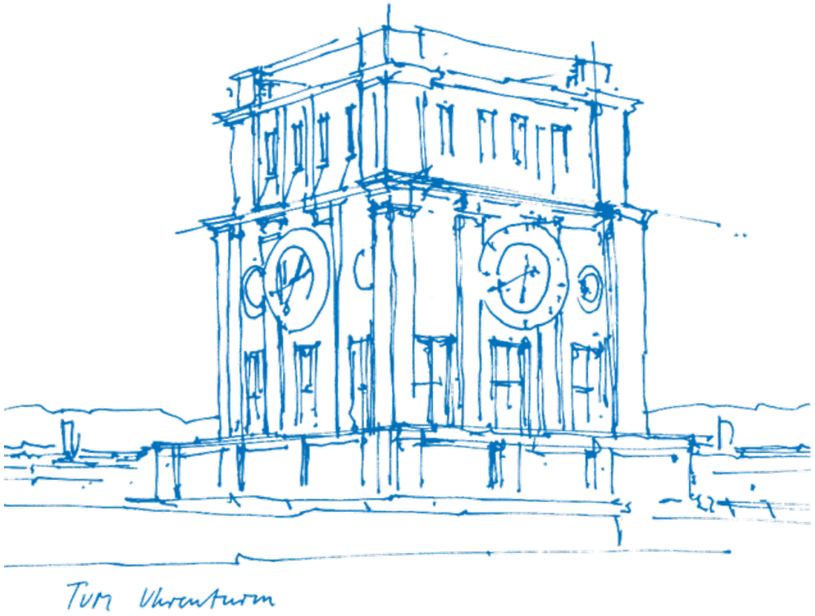
Parallel Programming Tutorial - SIMD

Vincent Bode, M.Sc.
Chair for Computer Architecture and Parallel Systems (CAPS)
Technical University Munich

May 26, 2020



Public Service Announcements

- The current assignment is extended to Friday night. Here is some supporting material:
  - The exercise support wiki on GitLab.
  - Help from the Q&A session (link to recording here or on moodle).
- New RocketChat channel for discussing high-performance solutions: #parprog-need-for-speed.
- There was some feedback asking us to include some more examples in the exercises.
- Additional (unofficial) resource: parprog comprehensive question bank here (source: GitHub).

Introduction to SIMD

Execution Models

What mental model do you have for the execution of this code on a single core?

```
// a and b are float arrays
for( int i = 0; i < 128; i++ ) {
    c[i] = a[i] * b[i];
}
```

Sequential Execution

Iteration	0
Operand 1	
Operation	
Operand 2	
Result	

Sequential Execution

Iteration	0
Operand 1	
Operation	*
Operand 2	
Result	

Sequential Execution

Iteration	0
Operand 1	$\boxed{0}$
Operation	*
Operand 2	
Result	

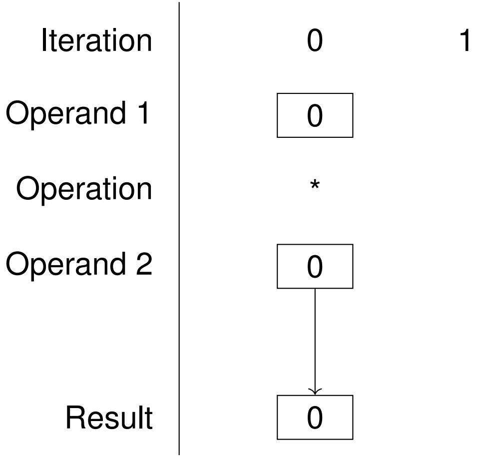
Sequential Execution

Iteration	0
Operand 1	0
Operation	*
Operand 2	0
Result	

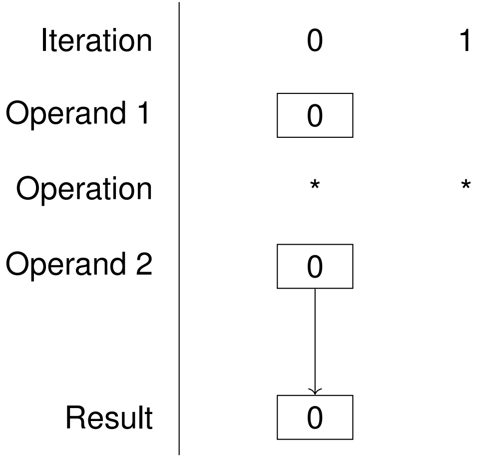
Sequential Execution

Iteration	0
Operand 1	0
Operation	*
Operand 2	0
Result	0

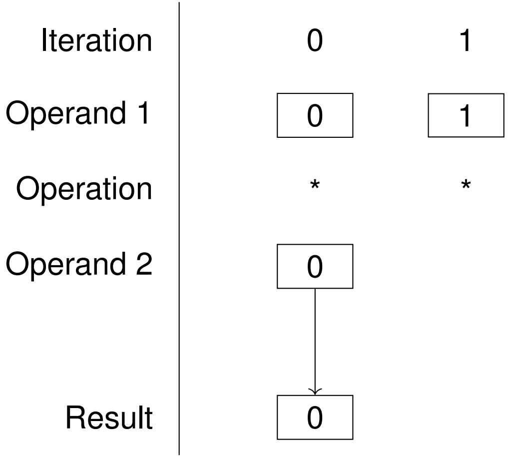
Sequential Execution



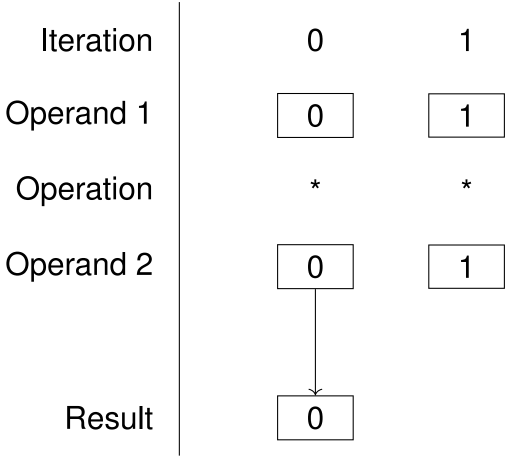
Sequential Execution



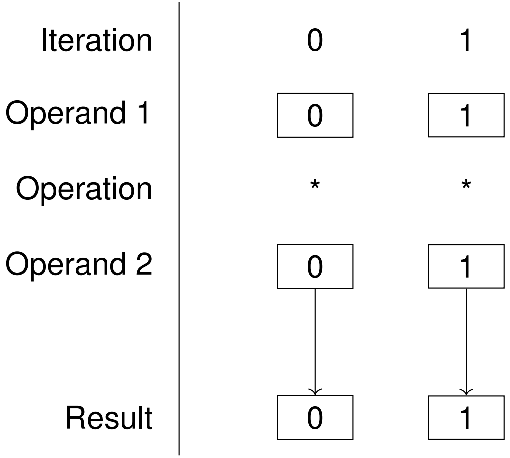
Sequential Execution



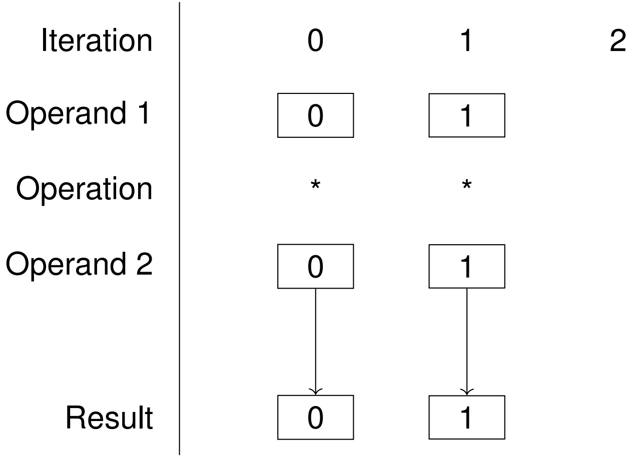
Sequential Execution



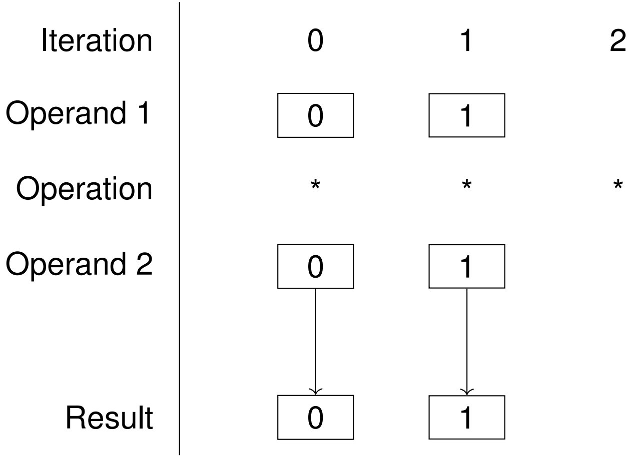
Sequential Execution



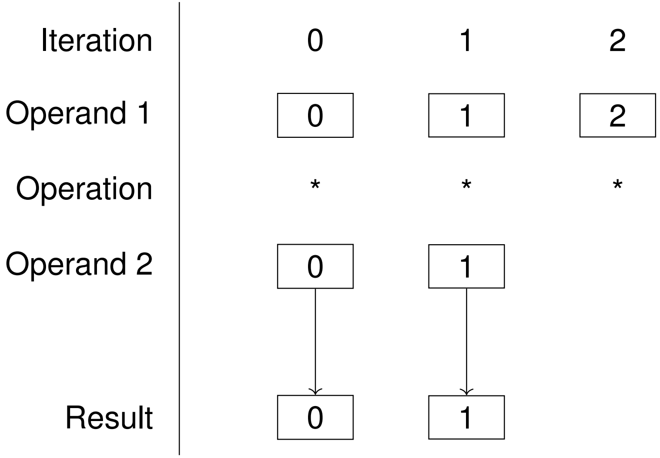
Sequential Execution



Sequential Execution



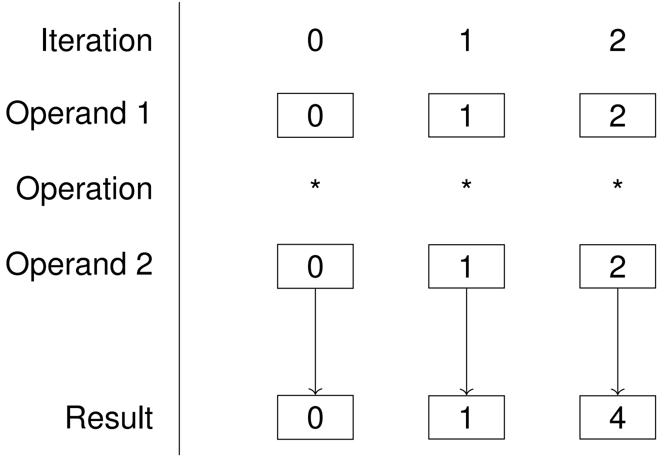
Sequential Execution



Sequential Execution

Iteration	0	1	2
Operand 1	0	1	2
Operation	*	*	*
Operand 2	0	1	2
Result	0	1	

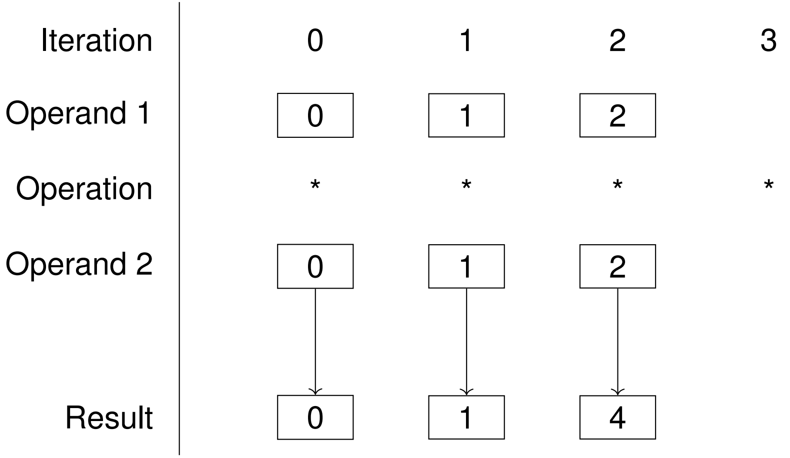
Sequential Execution



Sequential Execution

Iteration	0	1	2	3
Operand 1	0	1	2	
Operation	*	*	*	
Operand 2	0	1	2	
Result	0	1	4	

Sequential Execution



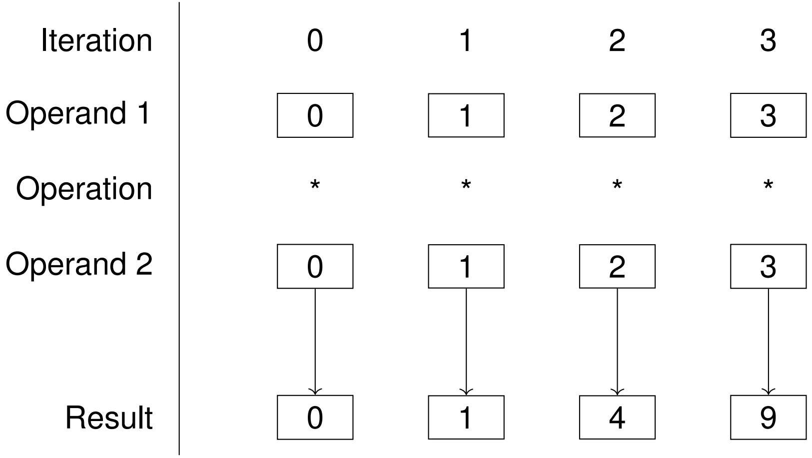
Sequential Execution

Iteration	0	1	2	3
Operand 1	0	1	2	3
Operation	*	*	*	*
Operand 2	0	1	2	
Result	0	1	4	

Sequential Execution

Iteration	0	1	2	3
Operand 1	0	1	2	3
Operation	*	*	*	*
Operand 2	0	1	2	3
Result	0	1	4	

Sequential Execution



Sequential Execution

Iteration	0	1	2	3
Operand 1	0	1	2	3
Operation	*	*	*	*
Operand 2	0	1	2	3
Result	0	1	4	9

Single Core Vectorized

Iteration	0-4
Operand 1	
Operation	
Operand 2	
Result	

Single Core Vectorized

Iteration	0-4
Operand 1	
Operation	*
Operand 2	
Result	

Single Core Vectorized

Iteration	0-4			
Operand 1	0	1	2	3
Operation	*			
Operand 2				
Result				

Single Core Vectorized

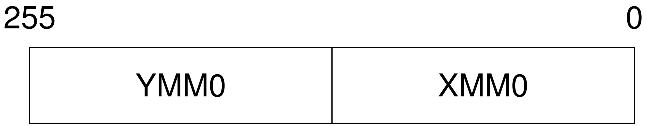
Iteration	0-4			
Operand 1	0	1	2	3
Operation	*			
Operand 2	0	1	2	3
Result				

Single Core Vectorized

Iteration	0-4			
Operand 1	0	1	2	3
Operation	*			
Operand 2	0	1	2	3
Result	0	1	4	9

SIMD

- This is SIMD: Single Instruction Mulitple Data.
- To support this, we need vector registers and Instruction Set support.
- Intel has had multiple extensions to its SIMD instructions.
- For example, SSE operates with 128 bit registers. AVX with 256 bit registers:



Intrinsics

Compilers can automatically vectorize your code in some cases. However, we will focus on Intel Intrinsics, a set of C style functions that can help you vectorize your code without writing assembly.

Intrinsics define datatypes and operations on these datatypes. For AVX, in GCC, these are defined in the header <immintrin.h >.

Intrinsics

AVX Datatypes:

- __m256 can hold eight 32 bit floating point numbers (float)
- __m256d can hold four 64 bit floating point numbers (double)
- __m256i can hold eight, 32 bit integer (int) OR four, 64 bit integers

AVX function examples:

- Functions for loading / storing data e.g. _mm256_loadu_ps
- Arithmetic functions e.g. _mm_add_ps add packed floating point numbers
- Byte manipulation functions like _mm_movelh_ps.

Intrinsics

Some terminology in the Intrinsics Guide:

- Packed (_ps) operations operate on the entire vector.
- Scalar operations (_sd) operate on the least significant data element (bits 0-31 for floats).
- Latency: Number of clock cycles from beginning of execution to end of execution.
  - Unit: cycles (lower is better)
- Throughput: Rate of execution of instructions.
  - Unit: Cycles per Instruction (CPI, lower is better) or Instructions per Cycle (IPC, higher is better)

These functions and datatypes are much closer to assembly than to normal C. You necessarily have to think about how to use these vector instructions efficiently to make full use of the hardware, which you also need to consider. It is recommended that you consult the Intel Intrinsics Guide.

Simple Example

Simple Example

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>

int main() {
    int size = 20000;
    int iterations = 100000;

    uint32_t* a = (uint32_t*)malloc(size * sizeof(uint32_t));
    for (int i = 0; i < size; ++i) a[i] = i;

    for (int iter = 0; iter < iterations; ++iter) {
        for (int i = 0; i < size; ++i) {
            a[i] = ((a[i] * a[i]) >> 1) ^ a[i];
        }
    }

    uint32_t sum = 0;
    for (int i = 0; i < size; ++i) sum += a[i];
    printf("%x\n", sum);
}
```

Simple Example

Zoom in on the interesting code:

```
a[i] = ((a[i] * a[i]) >> 1) ^ a[i];
```

Simple Example

C is much too hard to read, so look at the assembly

```c
a[i] = ((a[i] * a[i]) >> 1) ^ a[i];
```

```asm
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

You can explore the assembly yourself at https://godbolt.org/z/nh-KJ_

Simple Example

Rewrite the code to clearly show the individual operations.

```
uint32_t* a_ptr = a + i;
uint32_t a_i    = *a_ptr;
uint32_t mul    = a_i * a_i;
uint32_t srl    = mul >> 1;
uint32_t xor    = srl ^ a_i;
*a_ptr          = xor;
```

```
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

Simple Example

Do the actual vectorisation.

```
__m128i* a_ptr = a + i
__m128i a_i    = *a_ptr
__m128i mul    = a_i * a_i
__m128i srl    = mul » 1
__m128i xor    = srl ^ a_i
*a_ptr = xor
```

```
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

Simple Example

Do the actual vectorisation.

```
__m128i* a_ptr  = a + i ???
__m128i a_i     = *a_ptr
__m128i mul     = a_i * a_i
__m128i srl     = mul » 1
__m128i xor     = srl ^ a_i
*a_ptr = xor
```

```
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

Simple Example

Do the actual vectorisation.

```
__m128i* a_ptr = (__m128i*)(a + i);
__m128i a_i    = *a_ptr ???
__m128i mul    = a_i * a_i
__m128i srl    = mul >> 1
__m128i xor    = srl ^ a_i
*a_ptr = xor
```

```
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

Simple Example

Do the actual vectorisation.

```
__m128i* a_ptr = (__m128i*)(a + i);
__m128i a_i    = _mm_load_si128(a_ptr);
__m128i mul    = a_i * a_i ???
__m128i srl    = mul >> 1
__m128i xor    = srl ^ a_i
*a_ptr = xor
```

```
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

Simple Example

Do the actual vectorisation.

```
__m128i* a_ptr = (__m128i*)(a + i);
__m128i a_i    = _mm_load_si128(a_ptr);
__m128i mul    = _mm_mullo_epi32(a_i, a_i);
__m128i srl    = mul » 1 ???
__m128i xor    = srl ^ a_i
*a_ptr = xor
```

```
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

Simple Example

Do the actual vectorisation.

```
__m128i* a_ptr = (__m128i*)(a + i);
__m128i a_i    = _mm_load_si128(a_ptr);
__m128i mul    = _mm_mullo_epi32(a_i, a_i);
__m128i srl    = _mm_srli_epi32(mul, 1);
__m128i xor    = srl ^ a_i ???
*a_ptr = xor
```

```
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

Simple Example

Do the actual vectorisation.

```
__m128i* a_ptr = (__m128i*)(a + i);
__m128i a_i    = _mm_load_si128(a_ptr);
__m128i mul    = _mm_mullo_epi32(a_i, a_i);
__m128i srl    = _mm_srli_epi32(mul, 1);
__m128i xor    = _mm_xor_si128(srl, a_i);
*a_ptr = xor ???
```

```
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

Simple Example

Do the actual vectorisation.

```c
__m128i* a_ptr = (__m128i*)(a + i);
__m128i a_i    = _mm_load_si128(a_ptr);
__m128i mul    = _mm_mullo_epi32(a_i, a_i);
__m128i srl    = _mm_srli_epi32(mul, 1);
__m128i xor    = _mm_xor_si128(srl, a_i);
_mm_store_si128(a_ptr, xor);
```

```asm
mov edx, DWORD PTR [rcx]
mov esi, edx
imul esi, edx
shr esi
xor edx, esi
mov DWORD PTR [rcx-4], edx
```

Simple Example

Do the actual vectorisation.

```c
__m128i* a_ptr = (__m128i*)(a + i);
__m128i a_i    = _mm_load_si128(a_ptr);
__m128i mul    = _mm_mullo_epi32(a_i, a_i);
__m128i srl    = _mm_srli_epi32(mul, 1);
__m128i xor    = _mm_xor_si128(srl, a_i);
_mm_store_si128(a_ptr, xor);
```

```asm

vmovdqa xmm0, XMMWORD PTR [rdx]
vpmulld xmm1, xmm0, xmm0
vpsrld xmm1, xmm1, 1
vpxor xmm0, xmm0, xmm1
vmovaps XMMWORD PTR [rdx-16], xmm0
```

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>


int main() {
    int size = 20000;
    int iterations = 100000;

    uint32_t* a = (uint32_t*)malloc(size * sizeof(uint32_t))
    for (int i = 0; i < size; ++i) a[i] = i;

    for (int iter = 0; iter < iterations; ++iter) {
        for (int i = 0; i < size; ++i) {
            __m128i* a_ptr = (__m128i*)(a + i);
            __m128i a_i = _mm_load_si128(a_ptr);
            __m128i mul = _mm_mullo_epi32(a_i, a_i);
            __m128i srl = _mm_srli_epi32(mul, 1);
            __m128i xor = _mm_xor_si128(srl, a_i);
            _mm_store_si128(a_ptr, xor);
        }
    }

    uint32_t sum = 0;
    for (int i = 0; i < size; ++i) sum += a[i];
    printf("%x\n", sum);
}
```

Does it work?

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>


int main() {
    int size = 20000;
    int iterations = 100000;

    uint32_t* a = (uint32_t*)malloc(size * sizeof(uint32_t))
    for (int i = 0; i < size; ++i) a[i] = i;

    for (int iter = 0; iter < iterations; ++iter) {
        for (int i = 0; i < size; i += 4) {
            __m128i* a_ptr = (__m128i*)(a + i);
            __m128i a_i = _mm_load_si128(a_ptr);
            __m128i mul = _mm_mullo_epi32(a_i, a_i);
            __m128i srl = _mm_srli_epi32(mul, 1);
            __m128i xor = _mm_xor_si128(srl, a_i);
            _mm_store_si128(a_ptr, xor);
        }
    }

    uint32_t sum = 0;
    for (int i = 0; i < size; ++i) sum += a[i];
    printf("%x\n", sum);
}
```

Does it work?

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <immintrin.h>

int main() {
    int size = 20000;
    int iterations = 100000;

    uint32_t* a = (uint32_t*)malloc(size * sizeof(uint32_t))
    for (int i = 0; i < size; ++i) a[i] = i;

    for (int iter = 0; iter < iterations; ++iter) {
        for (int i = 0; i < size; i += 4) {
            __m128i* a_ptr = (__m128i*)(a + i);
            __m128i a_i = _mm_load_si128(a_ptr);
            __m128i mul = _mm_mullo_epi32(a_i, a_i);
            __m128i srl = _mm_srli_epi32(mul, 1);
            __m128i xor = _mm_xor_si128(srl, a_i);
            _mm_store_si128(a_ptr, xor);
        }
    }

    uint32_t sum = 0;
    for (int i = 0; i < size; ++i) sum += a[i];
    printf("%x\n", sum);
}
```

Does it work?

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <immintrin.h>

int main() {
    int size = 20000;
    int iterations = 100000;

    uint32_t* a = (uint32_t*)aligned_alloc(16, size * sizeof(uint32_t))
    for (int i = 0; i < size; ++i) a[i] = i;

    for (int iter = 0; iter < iterations; ++iter) {
        for (int i = 0; i < size; i += 4) {
            __m128i* a_ptr = (__m128i*)(a + i);
            __m128i a_i = _mm_load_si128(a_ptr);
            __m128i mul = _mm_mullo_epi32(a_i, a_i);
            __m128i srl = _mm_srli_epi32(mul, 1);
            __m128i xor = _mm_xor_si128(srl, a_i);
            _mm_store_si128(a_ptr, xor);
        }
    }

    uint32_t sum = 0;
    for (int i = 0; i < size; ++i) sum += a[i];
    printf("%x\n", sum);
}
```

Does it work?

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <immintrin.h>

int main() {
    int size = 20000;
    int iterations = 100000;

    uint32_t* a = (uint32_t*)aligned_alloc(16, size * sizeof(uint32_t))
    for (int i = 0; i < size; ++i) a[i] = i;

    for (int iter = 0; iter < iterations; ++iter) {
        for (int i = 0; i < size; i += 4) {
            __m128i* a_ptr = (__m128i*)(a + i);
            __m128i a_i = _mm_load_si128(a_ptr);
            __m128i mul = _mm_mullo_epi32(a_i, a_i);
            __m128i srl = _mm_srli_epi32(mul, 1);
            __m128i xor = _mm_xor_si128(srl, a_i);
            _mm_store_si128(a_ptr, xor);
        }
    }

    uint32_t sum = 0;
    for (int i = 0; i < size; ++i) sum += a[i];
    printf("%x\n", sum);
}
```

Yes!

Speedup?

Speedup?

$\sim$3.6

Assignment 5 – SIMD

Assignment 5 – SIMD

- Transposed matrix multiplication $C \Leftarrow \alpha AB^T + \beta C$
- That means we always calculate the dot product of the i-th row of A and the j-th row of B
  - Why not normal multiplication?
- Your task is to *manually* vectorise the code using Intel intrinsics
- Required speedup is 2

The server only supports AVX!
(No AVX2, no AVX512.)

The server only supports AVX!

(No AVX2, no AVX512.)

The server only supports AVX!

(No AVX2, no AVX512.)

Assignment 5, Hints

- Only use instructions up until AVX (this includes all of the SSE extensions)
  - If you try to use anything that is not supported, the compiler should generate an extremely unhelpful error message
  - In case you manage to convince the compiler to compile your code anyway, the executable will crash on the server (but probably run fine locally)
  - The slides of Micheal Klemm's talk include some AVX512 instructions, take care
- Use the Intel Intrinsics Guide to find out which instructions to use
- Note that the input is not a even multiple of the vector size, so you have to process the remainder
- Inspect the assembly to find out what the compiler is actually doing, either the old-fashioned way (`gcc -S ... > out.s` and inspect the file) or via Compiler Explorer
- The server's CPU is an Intel Xeon E5-2680 v0 2.70 GHz (Sandy Bridge)
- Also remember the Q&A sessions, Mo. 14:00.

Content Questions

This is a placeholder.

<insert your question here>

What we covered today

- Using intrinsics for SIMD
- Assignment 05