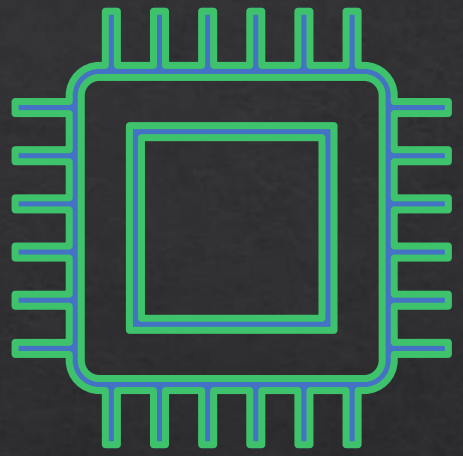


SIMD and Auto-Vectorization

By

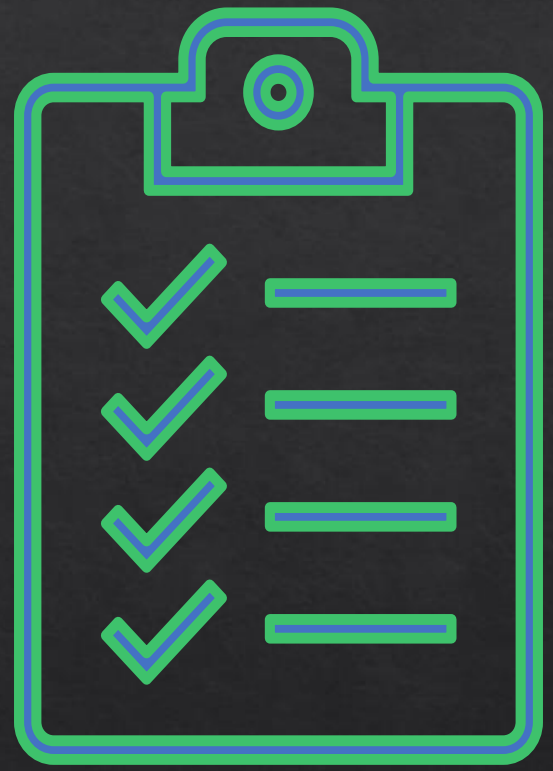
Mahendra Garodi

The Goal

- **We will talk about**
  - What is SIMD
  - What are the ways to take advantage of this feature
  - What is mean by Auto-Vectorization
  - What options does compiler provides to do diagnosis on code
- **What we are not covering**
  - Specific SIMD architecture and implementation details
  - What techniques do compiler to use vectorize the code

Agenda

- Classes of Parallelism
- A classic SAXPY loop
- SIMD Architecture
- Taking Advantage of SIMD
- Auto Vectorization
- Performance Improvement



Classes of Parallelism (In Software)

Data Level Parallelism

Many data items that can be operated at same time

Task-Level Parallelism

Tasks of work can be created that can be operate independently and largely in parallel

```
// Data Level Parallelism
for(size_t i = 0 ; i < size; i++)
{
    C[i] = A[i] + B[i];
}

// Tasks level Parallelism
void doTask1()
{
  // Do something
}

void doTask2()
{
  // Do something
}

void doWork()
{
    std::thread task1(doTask1);
    std::thread task2(doTask2);
    // Do something more here
    task1.join();
    task2.join();
}
```

Classes of Parallelism (In Hardware)

- Instruction Level Parallelism
- Vector Architectures and Graphic Processor Units
- Thread-Level Parallelism
- Multiprocessor
- Multicomputer

A Classic SAXPY Loop

(Scalar Version)

https://godbolt.org/z/W1cshGoo5

```
void saxpy(int* X, int* Y,
           int a, int n)
{
    for(int i = 0; i < n; i++)
    {
        X[i] = a * X[i] + Y[i];
    }
}
```

```
saxpy(int*, int*, int, int):
        test    ecx, ecx
        jle     .L1
        mov     ecx, ecx
        mov     eax, 0
.L3:
        mov     r8d, edx
        imul    r8d, DWORD PTR [rdi+rax*4]
        add     r8d, DWORD PTR [rsi+rax*4]
        mov     DWORD PTR [rdi+rax*4], r8d
        add     rax, 1
        cmp     rax, rcx
        jne     .L3
.L1:
        ret
```

A Classic SAXPY Loop

(Scalar Version)

https://godbolt.org/z/W1cshGoo5

```
saxpy(int*, int*, int, int):
        test    ecx, ecx
        jle     .L1
        mov     ecx, ecx
        mov     eax, 0
.L3:
        mov     r8d, edx
        imul    r8d, DWORD PTR [rdi+rax*4]
        add     r8d, DWORD PTR [rsi+rax*4]
        mov     DWORD PTR [rdi+rax*4], r8d
        add     rax, 1
        cmp     rax, rcx
        jne     .L3
.L1:
        ret
```

X[0] = a * X[0] + Y[0]

X[1] = a * X[1] + Y[1]

X[2] = a * X[2] + Y[2]

X[3] = a * X[3] + Y[3]

X[4] = a * X[4] + Y[4]

X[5] = a * X[5] + Y[5]

X[6] = a * X[6] + Y[6]

X[7] = a * X[7] + Y[7]

.....

A Classic SAXPY Loop

(SIMD Version 1)

```
void saxpy(int* X, int* Y,
           int a, int n)
{
    for(int i = 0; i < n; i++)
    {
        X[i] = a * X[i] + Y[i];
    }
}
```

X[0] = a * X[0] + Y[0]

X[1] = a * X[1] + Y[1]

X[2] = a * X[2] + Y[2]

X[3] = a * X[3] + Y[3]

X[4] = a * X[4] + Y[4]

X[5] = a * X[5] + Y[5]

X[6] = a * X[6] + Y[6]

X[7] = a * X[7] + Y[7]

.....

A Classic SAXPY Loop

(SIMD Version 2)

https://godbolt.org/z/1ddscaEa6

```
void saxpy(int* X, int* Y,
           int a, int n)
{
    for(int i = 0; i < n; i++)
    {
        X[i] = a * X[i] + Y[i];
    }
}
```

X[0] = a * X[0] + Y[0]

X[1] = a * X[1] + Y[1]

X[2] = a * X[2] + Y[2]

X[3] = a * X[3] + Y[3]

X[4] = a * X[4] + Y[4]

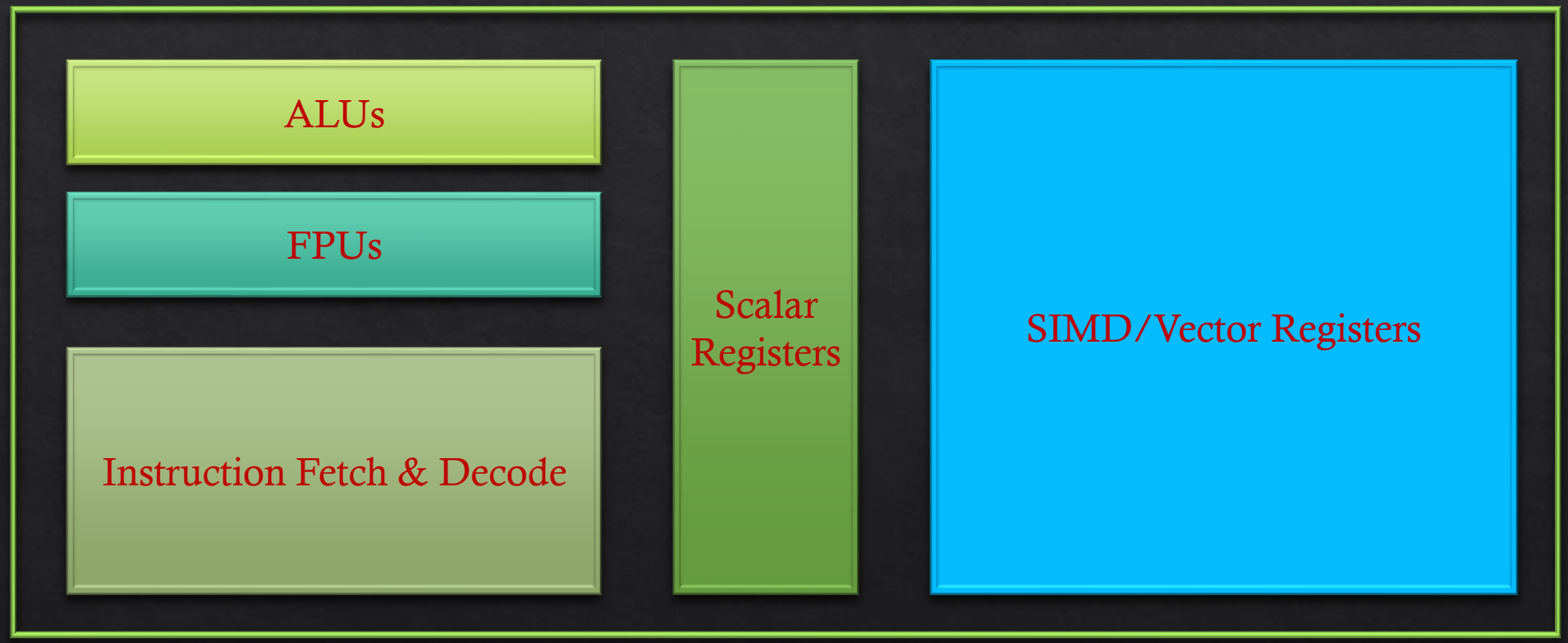
X[5] = a * X[5] + Y[5]

X[6] = a * X[6] + Y[6]

X[7] = a * X[7] + Y[7]

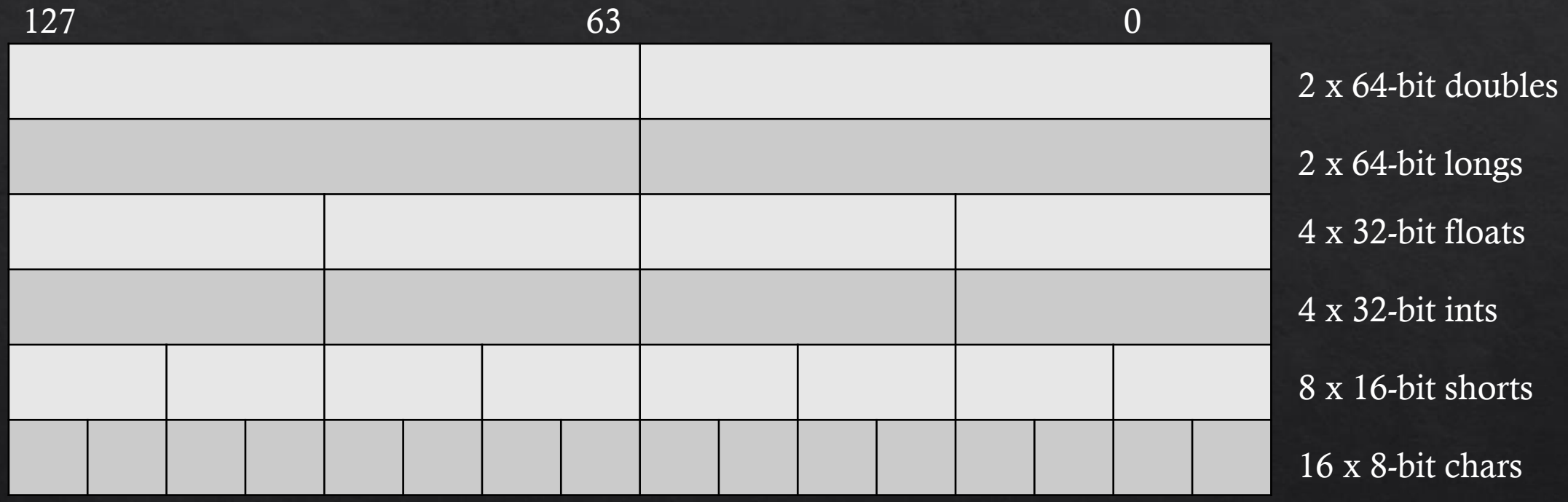
.....

CPU Core with SIMD Registers



Credits: CppCon 2016: Pablo Halpern “Introduction to Vector Parallelism"

SIMD Register Layout



Credits: CppCon 2016: Pablo Halpern “Introduction to Vector Parallelism"

Advantages of SIMD over Threads/Cores

- Energy efficient
- Zero scheduling overhead
- No synchronization required at the end of loop

Taking Advantage of SIMD

- Assembly
- SIMD Intrinsic functions
- Explicit Vectorization Libraries/Wrapper Classes
- Auto Vectorization

Assembly

- The most crude way of programming
- No transformation done by compiler
- **Advantages**
  - You will get what you request
  - Complete control
- **Disadvantages**
  - No source code level portability
  - Needs familiarity with instruction set

```
void doSomeWork(int* A, int* B)
{
    for(int i = 0; i < 102; i++)
    {
        A[i] = A[i] + B[i];
    }
}

// Equivalent Assembly
doSomeWork(int*, int*):
        xor     eax, eax
.L2:
        // For 100 elements
        movdqu  xmm0, XMMWORD PTR [rdi+rax]
        movdqu  xmm2, XMMWORD PTR [rsi+rax]
        paddd   xmm0, xmm2
        movups  XMMWORD PTR [rdi+rax], xmm0
        add     rax, 16
        cmp     rax, 400
        jne     .L2
        // For 2 elements
        movq    xmm0, QWORD PTR [rdi+400]
        movq    xmm1, QWORD PTR [rsi+400]
        paddd   xmm0, xmm1
        movq    QWORD PTR [rdi+400], xmm0
        ret
```

SIMD Intrinsics

- C or C++ pseudo-function calls that compiler replaces with the appropriate SIMD instructions.
- Let you use the data types and operations available in the SIMD implementation
- **Advantages**
  - Better than writing code in Assembly
- **Disadvantages**
  - Limited portability as intrinsics are specific to instruction set
  - SIMD operations must be stated explicitly by the programmer
  - Compiler has to perform some additional operations when managing registers

SIMD Intrinsics

- Example for adding elements of two array on SSE2
- https://godbolt.org/z/a3PenP6no

```
#include <emmintrin.h>
#include <immintrin.h>

constexpr int kSize = 102;

void doSomeWork(int* A, int* B)
{
    __m128i* first_src   = (__m128i*)A;
    __m128i* second_src  = (__m128i*)B;
    int loop_count = kSize/4;

    for ( int i = 0; i < loop_count; i++)
    {
        *first_src = _mm_add_epi32(*first_src, *second_src);
        ++first_src;
        ++second_src;
    }

    // Remainder Loop
    for (int i = loop_count * 4 ; i < kSize; i++)
    {
        A[i] = A[i] + B[i];
    }
}
```

Explicit Vectorization Libraries/Wrapper Classes

- Wrapper on the SIMD intrinsics
- **Advantages**
  - Perfect for platform portability
  - More readable code
- **Disadvantages**
  - Has to be more direct than auto vectorization
  - Requires understanding of SIMD computation model
  - Might need to manage remainder loop

std::experimental::simd

https://godbolt.org/z/Kz85fcf8Y

```cpp
#include <vector>
#include <algorithm>
#include <numeric>
#include <https://raw.githubusercontent.com/VcDevel
          /std-simd/compiler_explorer/simd.h>

using Vector = std::vector<float>;
float scalar_product(const Vector& a, const Vector& b)
{
    return std::inner_product(a.begin(), a.end(), b.begin(), 0);
}

namespace stdx = std::experimental;
using SimdFloat = stdx::native_simd<float>;
using VectorSimd = std::vector<SimdFloat>;

SimdFloat scalar_product_simd(const VectorSimd& a,
                              const VectorSimd& b)
{
    stdx::native_simd<float> result{0};
    return std::inner_product(a.begin(), a.end(), b.begin(),
                              static_cast<SimdFloat>(0));
}
```

XSIMD Example

https://godbolt.org/z/Goj8EK489

```cpp
#include <cstddef>
#include <vector>
#include <limits>
#include "xsimd/xsimd.hpp"

namespace xs = xsimd;
using vector_type = std::vector<double, XSIMD_DEFAULT_ALLOCATOR(double)>;

void mean(const vector_type& a, const vector_type& b, vector_type& res)
{
    std::size_t size = a.size();
    constexpr std::size_t simd_size = xsimd::simd_type<double>::size;
    std::size_t vec_size = size - size % simd_size;

    for(std::size_t i = 0; i < vec_size; i += simd_size)
    {
        auto ba = xs::load_aligned(&a[i]);
        auto bb = xs::load_aligned(&b[i]);
        auto bres = (ba + bb) / 2;
        bres.store_aligned(&res[i]);
    }
    for(std::size_t i = vec_size; i < size; ++i)
    {
        res[i] = (a[i] + b[i]) / 2;
    }
}
```

Explicit Vectorization Libraries/Wrapper Classes

https://github.com/VcDevel/std-simd *

https://github.com/xtensor-stack/xsimd *

https://github.com/agenium-scale/nsimd *

https://github.com/google/dimsum

https://github.com/p12tic/libsimdpp

https://github.com/aff3ct/MIPP

* *Available on Compiler Explorer*

Auto-Vectorization

- Compiler analyzes the code and emits SIMD instructions
- **Advantages**
  - Write code normally and rely on the compiler
  - Source code level portability
- **Disadvantages**
  - Compilers cannot vectorize everything
  - Doesn’t guarantee performance stability

Auto-Vectorization

```
void mul(int* A, int* B)
{
    for(int i = 0; i < 2048; i++)
    {
        A[i] = A[i] * B[i];
    }
}
```

https://godbolt.org/z/qaaosbfr6

```
mul(int*, int*):
        // Check if any overlap
        lea     rdx, [rsi+4]
        mov     rax, rdi
        sub     rax, rdx
        cmp     rax, 24
        mov     eax, 0
        jbe     .L2
.L3:
        // If no overlap then use SIMD
        vmovdqu ymm1, YMMWORD PTR [rdi+rax]
        vpmulld ymm0, ymm1, YMMWORD PTR [rsi+rax]
        vmovdqu YMMWORD PTR [rdi+rax], ymm0
        add     rax, 32
        cmp     rax, 8192
        jne     .L3
        vzeroupper
        ret
.L2:
        // If overlap then go with SISD
        mov     edx, DWORD PTR [rdi+rax]
        imul    edx, DWORD PTR [rsi+rax]
        mov     DWORD PTR [rdi+rax], edx
        add     rax, 4
        cmp     rax, 8192
        jne     .L2
        ret
```

SLP Vectorization

- SLP – Superword Level Parallelism
- Combine similar independent instructions into vector instructions.
- Can Vectorize
  - Memory accesses
  - Arithmetic operations
  - Comparison operations

```
void foo(int a1, int a2, int b1,
         int b2, int *A)
{
  A[0] = a1*(a1 + b1);
  A[1] = a2*(a2 + b2);
  A[2] = a1*(a1 + b1);
  A[3] = a2*(a2 + b2);
}

foo(int, int, int, int, int*):
        lea     eax, [rdi+rdx]
        imul    eax, edi
        lea     edx, [rsi+rcx]
        imul    edx, esi
        vmovd   xmm1, eax
        vpinsrd xmm0, xmm1, edx, 1
        vpunpcklqdq     xmm0, xmm0, xmm0
        vmovdqu XMMWORD PTR [r8], xmm0
        ret
```

Auto-Vectorization (GCC)

- By default, enabled at -O3
- Can be explicitly enabled by '-**ftree-vectorize**'
- -fopt-info-vec-all : All information
- -fopt-info-vec-missed : Information about missed optimization
- -fopt-info-vec-optimized : Information about optimized loops
- -march : Explicitly specify the architecture
  - native is good option

Auto-Vectorization (Clang/LLVM)

- By default, enabled at -O2
- -Rpass-analysis=loop-vectorize, -Rpass-analysis=slp-vectorize
- -Rpass=loop-vectorize, Rpass=slp-vectorize
- -Rpass-missed=loop-vectorize, -Rpass-missed=slp-vectorize
- -march : Explicitly specify the architecture
  - native is good option

Auto-Vectorization (Clang/LLVM)

- Force Vectorization Parameters
  - -force-vector-width
  - -force-vector-interleave
  - #pragma clang loop vectorize(enable/disable) interleave(enable/disable)
  - #pragma clang loop vectorize_width(<positive_number>) interleave_count(<positive_number>)

What Types of Loops Can be Vectorized

- Loops with unknown trip count
- Runtime Checks of Pointers
- Reductions
- Inductions
- If Conversion
- Pointer Induction Variables
- Reverse Iterators
- Scatter / Gather
- Vectorization of Mixed Types
- Global Structures Alias Analysis
- Vectorization of function calls
- Partial unrolling during vectorization
- Epilogue Vectorization

Credits: https://llvm.org/docs/Vectorizers.html

Things that hurts…

- Possible aliasing – Might array X and Y overlap
- Unknown alignment – Aligned accesses to vector length is faster
- Opaque function calls – Must typically be called serially
- Significant control divergence, especially using ***break*** or ***continue***
- Gathers and Scatters
  - Non-Contiguous load and store – Might not be profitable to vectorize

Credits: CppCon 2016: Pablo Halpern “Introduction to Vector Parallelism"

How to Analyze the Code?

Thanks !