GPUs and SIMD programming

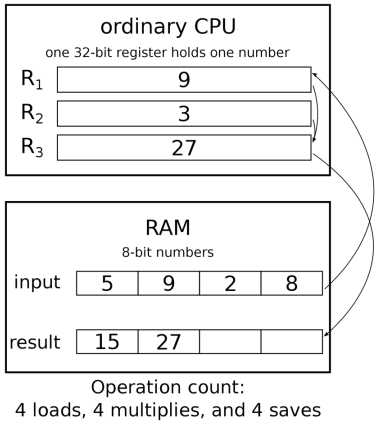
DAT470/DIT065 Computational techniques for large-scale data

2024-05-15

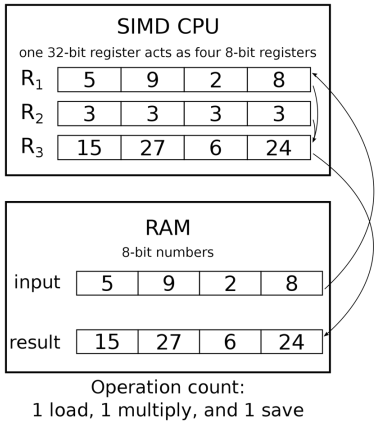
Matti Karppa

Vector extensions (SIMD)

- Ordinarily, CPU registers hold **scalars**, individual numbers
- If we want to perform **vector operations**, for example, compute the elementwise sum of two arrays of numbers, we need to perform one such addition per each element
- This can amount to potentially $5n$ operations: 2 loads, 1 addition, 1 store per element
- If we introduce longer vector registers, we can treat the registers as vectors of m elements
  - There is some variation in language, but we may say that we **pack** m elements into the register
  - Sometimes we refer to the elements of the register as **lanes**: we have m lanes that perform operations simultaneously
- Optimistically, this can reduce the number of operations required by a factor of m



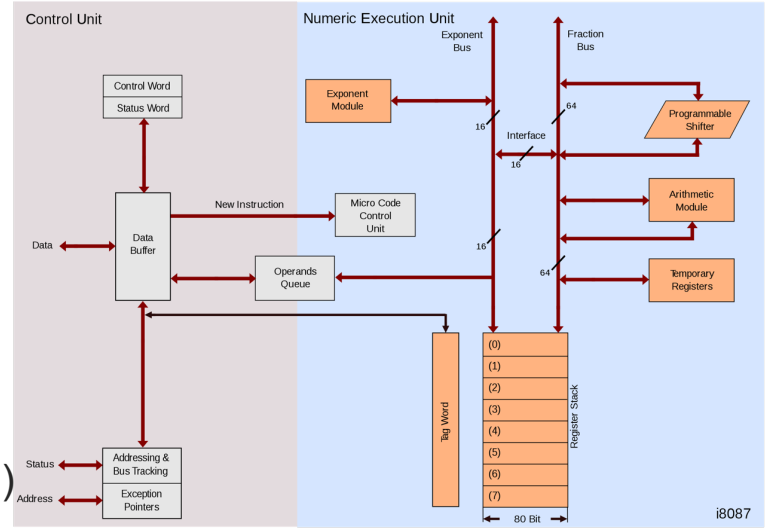
https://en.wikipedia.org/wiki/File:Non-SIMD_cpu_diagram1.svg



https://en.wikipedia.org/wiki/File:SIMD_cpu_diagram1.svg

x86 extensions: x87

- When the 16-bit Intel 8086 was released in 1978, it did not support floating-point arithmetic in hardware at all
- In 1980, Intel released the x87 coprocessor that introduced floating-point operations
- The processor has a **stack** of eight **80-bit** floating-point registers
  - Registers are commonly denoted ST(0), ST(1), …, ST(7)
  - Instructions do not directly specify which registers are accessed; instead, operands are pushed and popped on the stack and the topmost operands are manipulated
- The x87 predates IEEE 754: data is stored in Intel's proprietary **x86 Extended Precision Format** using 80 bits (15 bits for the exponent and 64 bits for the mantissa)
- For example, GCC makes this format available as the `long double` data type
- NumPy exposes this as `np.longdouble` and on Linux x86-64 as `np.float96` and `np.float128` (which are zero-padded to suitable word boundary)



https://commons.wikimedia.org/wiki/File:Intel_8087_arch.svg

x86 extensions: MMX

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

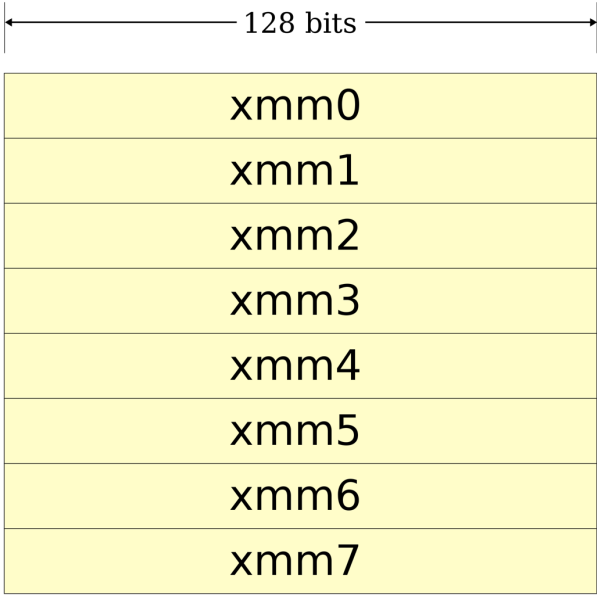
- The MMX (unofficially claimed to stand for MultiMedia eXtension) was the first SIMD extension to the x86, introduced in 1997, with the Pentium MMX
- MMX defines eight 64-bit registers MM0, MM1, …, MM7
  - Each register can hold one 64-bit word, or “pack” two 32-bit words or four 16-bit words or eight 8-bit bytes
  - Any basic arithmetic operation is then applied elementwise to all words packed in the register, that is, up to eight 8-bit operations are done with one instruction
- The MM-registers are aliases to the ST-registers of the x87, so no new register hardware is introduced
  - This means no new context is introduced (e.g., for context switching with multitasking operating systems), as the x87 registers are saved anyway
  - The MM registers are directly accessible, no need to use a stack
- The MMX only supports integer operations, no floating-point operations

x86 extensions: 3DNow!™

- AMD introduced its competitor to the MMX in 1998, with the AMD K6-2
- The important addition was packing single-precision floating-points into the MMX registers
- The instruction set was later extended with the Athlon family of CPUs
- However, the extension has since been deprecated and no longer supported by new CPUs

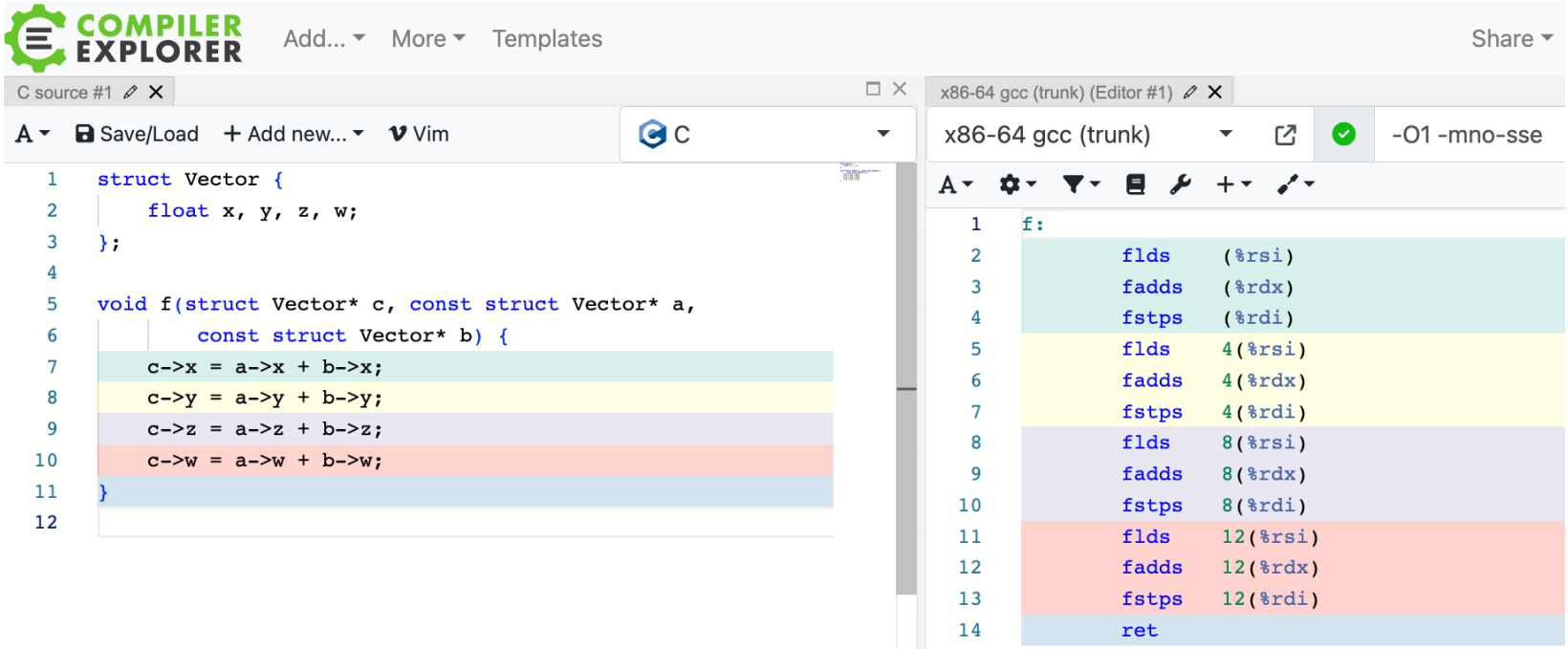
SSE

- Intel introduced the Streaming SIMD Extensions (SSE) in 1999 with the Pentium III
- SSE introduces eight completely new 128-bit vector registers XMM0, XMM1, …, XMM7
  - As with the MMX, the registers can hold multiple packed words
  - Initially, this was four 32-bit floating-points
  - From SSE2 (2000), also two 64-bit double-precision floating-points or integers, four 32-bit integers, eight 16-bit integers, or sixteen 8-bit bytes are allowed
- SSE introduces support for IEEE 754 arithmetic on the XMM registers
  - So modern x86 computers have two distinct hardware for doing floating-point computations
- x86-64 (2003) increases the number of XMM registers to 16

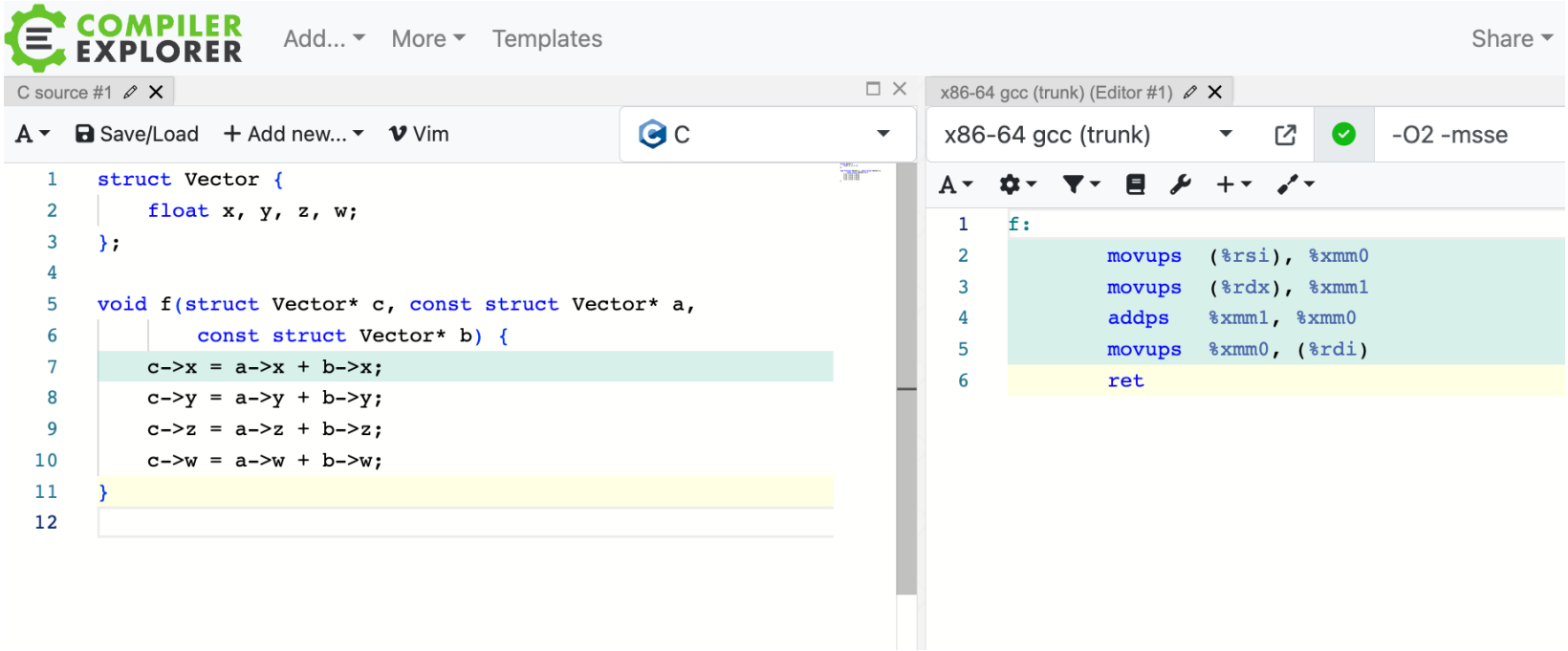


https://commons.wikimedia.org/wiki/File:XMM_registers.svg

Without SSE


COMPILER EXPLORER
Add...
More
Templates
Share
C source #1
Save/Load
Add new...
Vim
C
x86-64 gcc (trunk) (Editor #1)
x86-64 gcc (trunk)
-O1 -mno-sse
struct Vector {
float x, y, z, w;
};
void f(struct Vector* c, const struct Vector* a,
const struct Vector* b) {
c->x = a->x + b->x;
c->y = a->y + b->y;
c->z = a->z + b->z;
c->w = a->w + b->w;
}
f:
flds (%rsi)
fadds (%rdx)
fstps (%rdi)
flds 4(%rsi)
fadds 4(%rdx)
fstps 4(%rdi)
flds 8(%rsi)
fadds 8(%rdx)
fstps 8(%rdi)
flds 12(%rsi)
fadds 12(%rdx)
fstps 12(%rdi)
ret

With SSE


COMPILER EXPLORER
Add...
More
Templates
Share
C source #1
Save/Load
Add new...
Vim
C
x86-64 gcc (trunk) (Editor #1)
x86-64 gcc (trunk)
-O2 -msse
struct Vector {
float x, y, z, w;
};
void f(struct Vector* c, const struct Vector* a,
const struct Vector* b) {
c->x = a->x + b->x;
c->y = a->y + b->y;
c->z = a->z + b->z;
c->w = a->w + b->w;
}
f:
movups (%rsi), %xmm0
movups (%rdx), %xmm1
addps %xmm1, %xmm0
movups %xmm0, (%rdi)
ret

Some particularities of SSE

- For best performance, memory accesses with SSE instructions should be **aligned**
  - "Alignment" means in this case that the memory address should be divisible by 16, that is, aligned at the start of a contiguous 128-bit segment
- SSE allows also for unaligned accesses, but these are slower
- In addition to packed operations, SSE also allows "scalar" operations, that is, if we want to, e.g., perform ordinary floating-point operations on individual words
- In addition to arithmetic operations, SSE supports bitwise operations, comparison operations and **shuffling** operations
  - Shuffling means that we move the packed words around within the register
- SSE also contains cache management instructions, such as instructing the CPU to **prefetch** memory content into the cache
  - If we know a memory address will be accessed soon, it can be fetched into cache while other computations are being done to hide latency
- Later versions of SSE include special instructions for things like (just examples):
  - Popcount (how many 1's are there in a word)
  - Dot product
  - Fused Multiply Add (FMA): compute operation $result = a \cdot b + c$ such that the result is one of the operands a, b, c using one instruction; very useful for linear algebra

AVX

- Advanced Vector Extensions (AVX) were introduced by Intel in 2011
- AVX introduces 16 new 256-bit registers YMM0, YMM1, …, YMM15
  - The registers alias XMM registers, so the SSE registers are simply the lower 128-bits of these registers
  - Each register can hold eight 32-bit floating-points or four 64-bit floating-points
  - Memory alignment becomes 32 bytes (but unaligned accesses are also available, at a cost)
- Introduces lots of new instructions that make for easier vectorization of code, such as
  - Broadcasting an operand to all vector positions in a vector
  - Masked moves: only copy certain elements in a vector into destination and leave everything else untouched

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

AVX2

- Advanced Vector Extensions 2 (AVX2) were introduced by Intel in 2013
- AVX2 introduces new instructions and extends old instructions to be used more flexibly, for example:
  - Integer instructions for full YMM registers
  - Gather support: non-contiguous memory locations can be loaded into vector registers

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

AVX-512

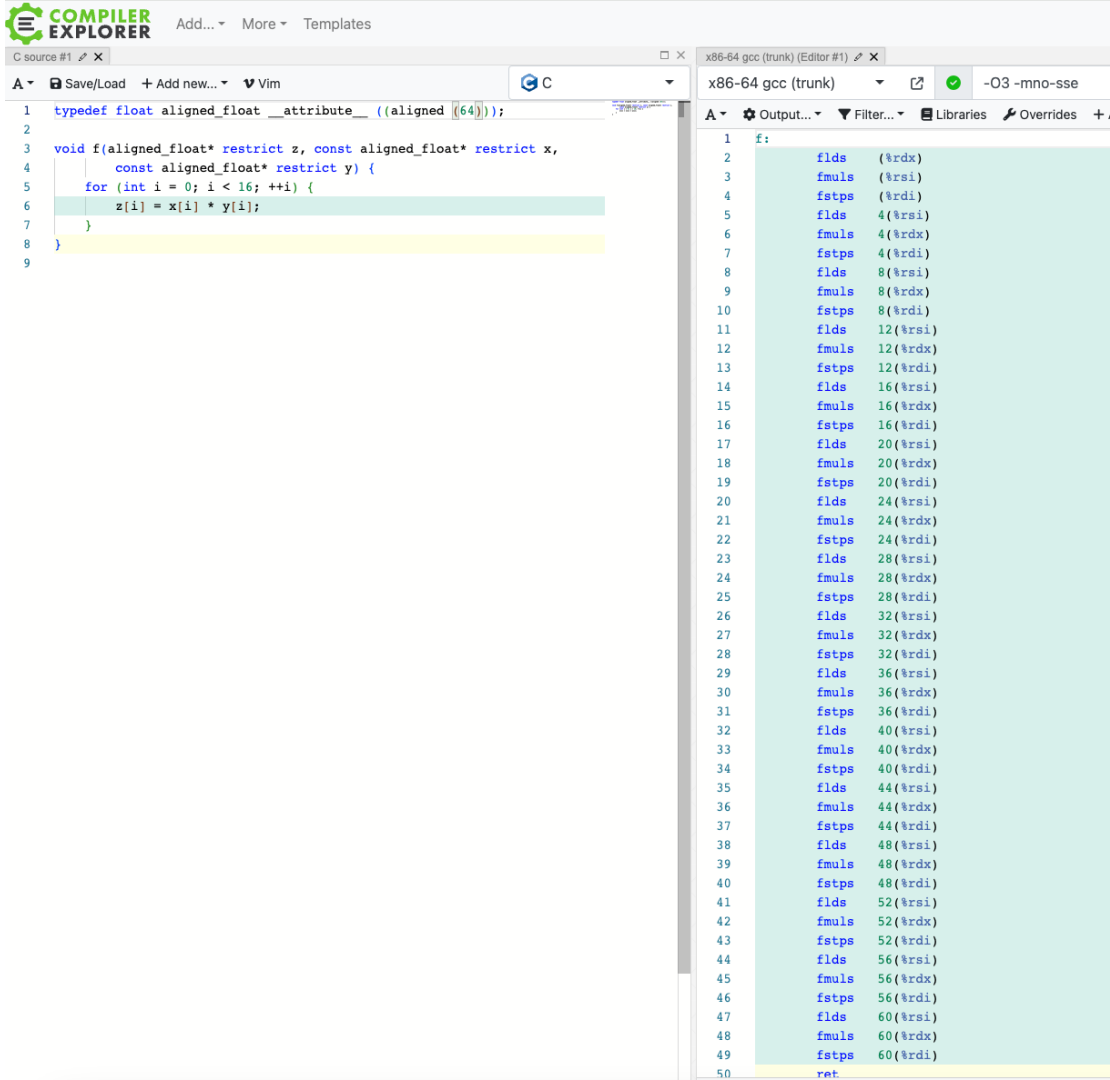
- AVX-512 was introduced by Intel in 2016
- Introduces 32 new 512-bit vector registers ZMM0, ZMM1, …, ZMM31
  - The old SSE and AVX registers are aliased as the low-bits of ZMM0, .., ZMM15
  - The low bits of ZMM16, ..., ZMM31 can be accessed analogously as XMM16, …, XMM31 and YMM16, …, YMM31
- AVX-512 is not a monolithic extension, but a group of independent extensions and not all features are implemented by all CPUs (notably consumer CPUs often do not have AVX-512 support at all)
- AVX-512 brings the CPU closer to an actual vector architecture, with the introduction of the **opmask register** that can be used as a binary mask to control which of the vector elements to write into a destination
- AVX-512 introduces scatter instructions to complement the pre-existing gather instructions to scatter data from registers into non-contiguous locations in memory

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

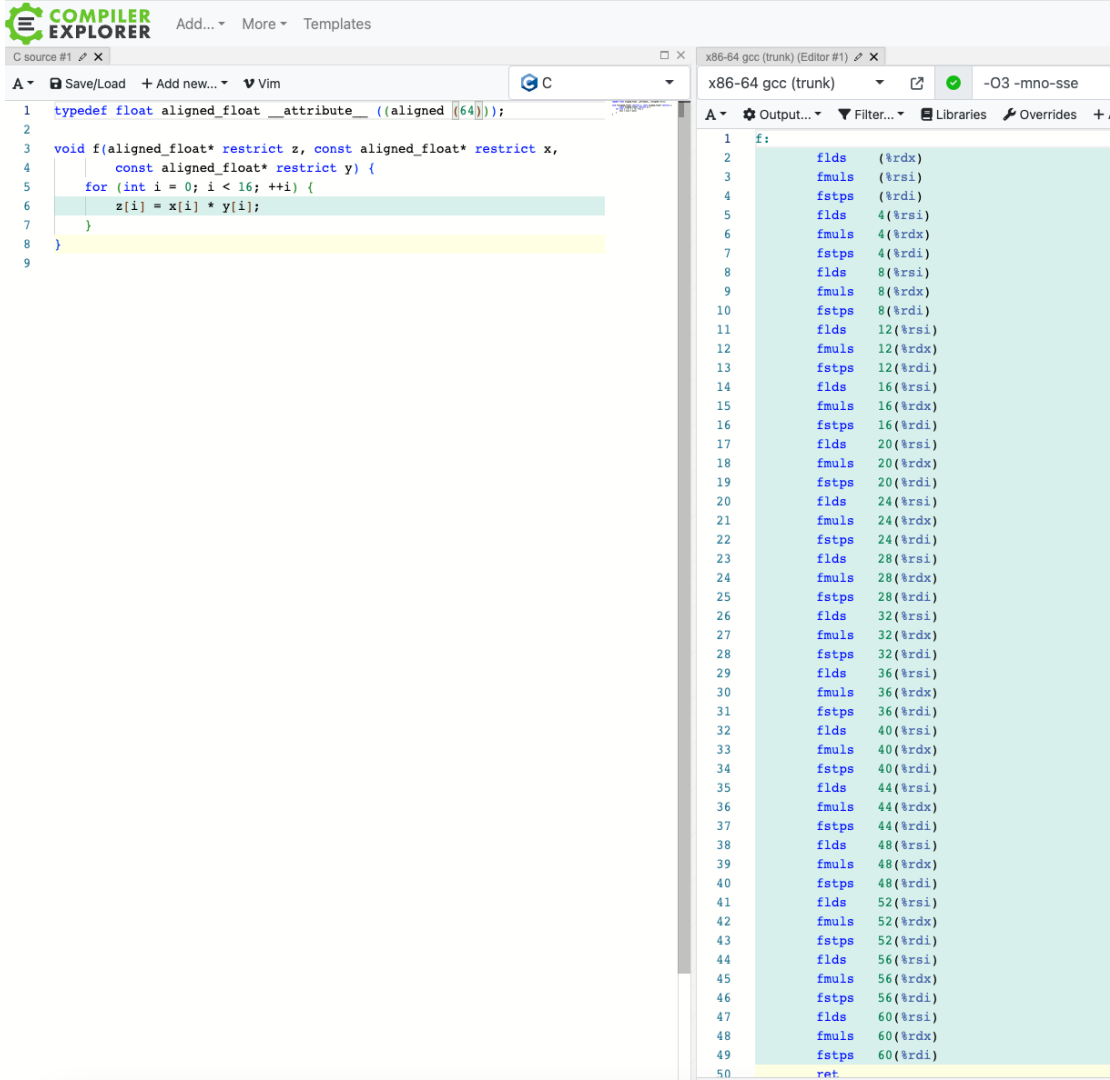
Example

- Given $x, y \in \mathbb{R}^{16}$, compute $z \leftarrow x \circ y$, that is, $z_i \leftarrow x_i y_i$ for $i \in \{1,2,\ldots,16\}$
- x, y, and z are stored as arrays of 32-bit single-precision floating-points of length 16
- That is, 512 bits in total
- We have annotated the type to be aligned at 64-byte boundary
- The keyword restrict tells allows us to assume that the arrays do not **alias** each other: the memory they point to are distinct
- Note: none of these assumptions can be checked by the compiler, we are telling the compiler “trust me, bro, I know what I’m doing”; if the data we pass to the function is not correctly aligned or the arguments do in fact alias one another, **anything can happen**



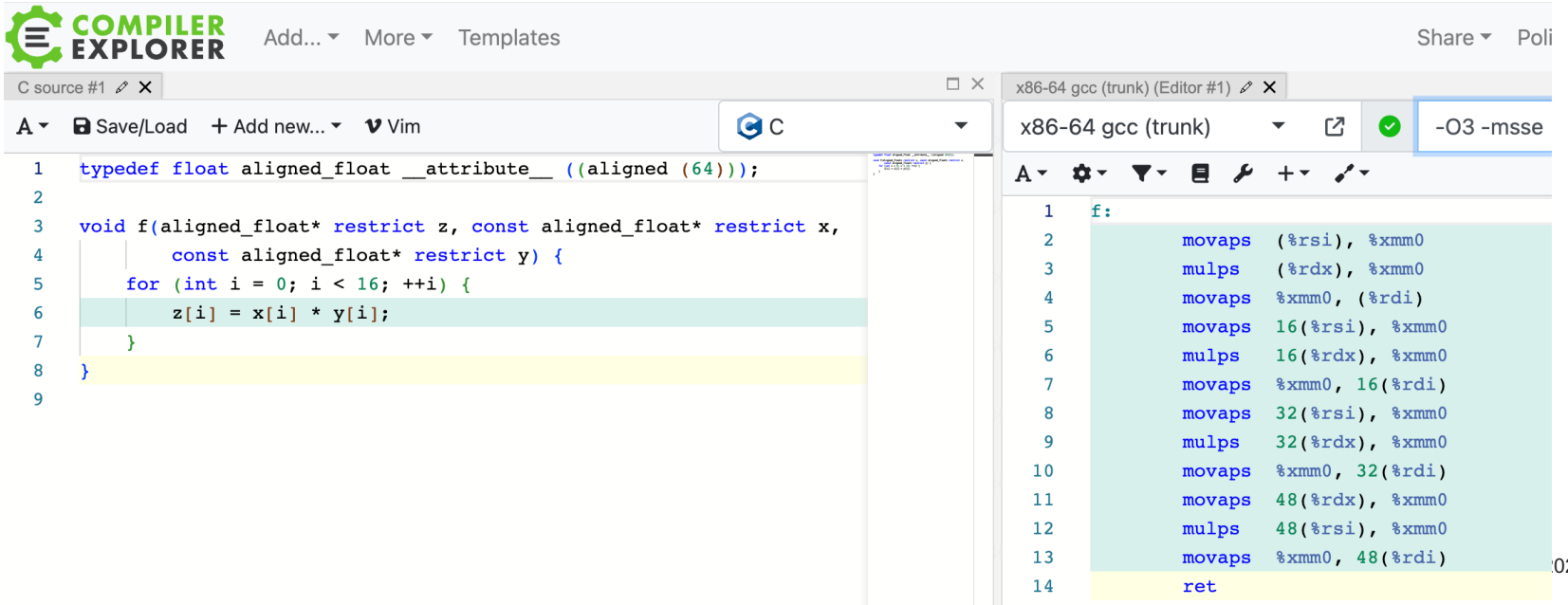
Example

- Given $x, y \in \mathbb{R}^{16}$, compute $z \leftarrow x \circ y$, that is, $z_i \leftarrow x_i y_i$ for $i \in \{1,2,\ldots,16\}$
- x, y, and z are stored as arrays of 32-bit single-precision floating-points of length 16
- That is, 512 bits in total
- We have annotated the type to be aligned at 64-byte boundary
- The keyword restrict tells allows us to assume that the arrays do not **alias** each other: the memory they point to are distinct
- Note: none of these assumptions can be checked by the compiler, we are telling the compiler “trust me, bro, I know what I’m doing”; if the data we pass to the function is not correctly aligned or the arguments do in fact alias one another, **anything can happen**



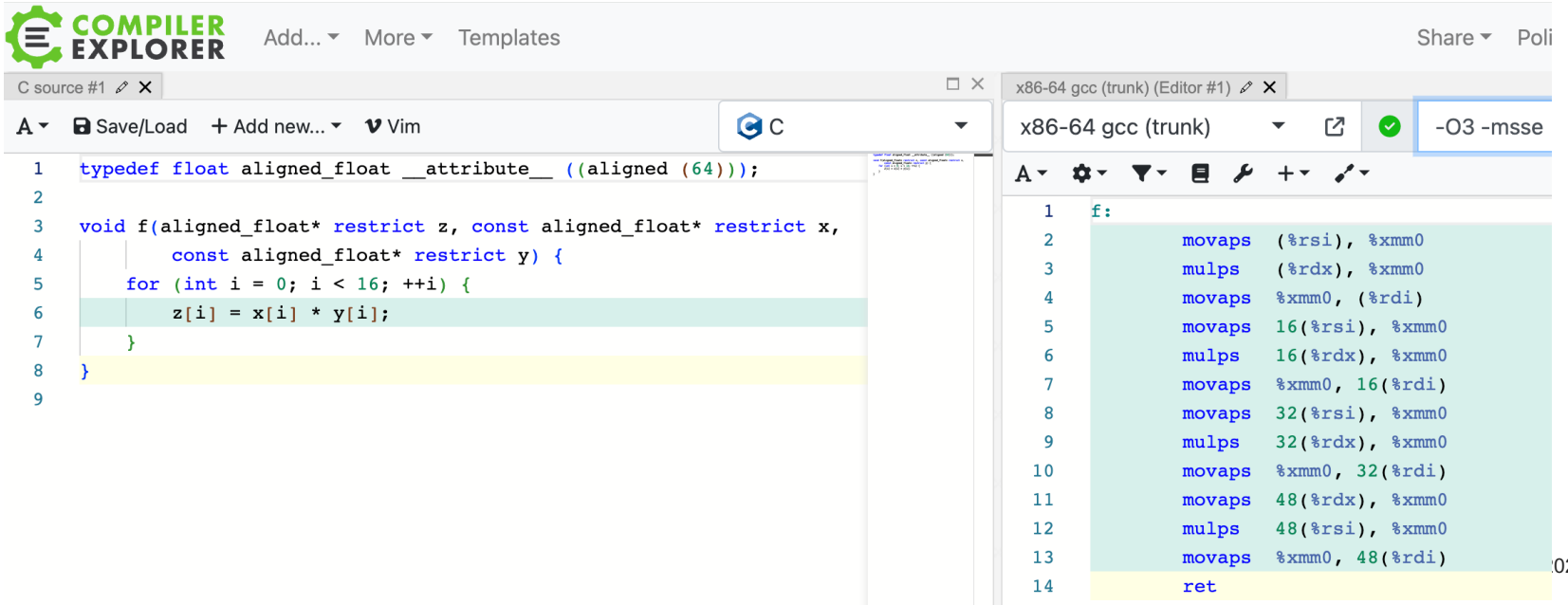
Example

- With SSE enabled, instead of unrolling the loop one element at a time, we can process 4 elements at a time (128 bits)
- Note that we can use the more efficient movaps instructions as the data is assumed aligned (unaligned data would cause the program to crash with these instructions)



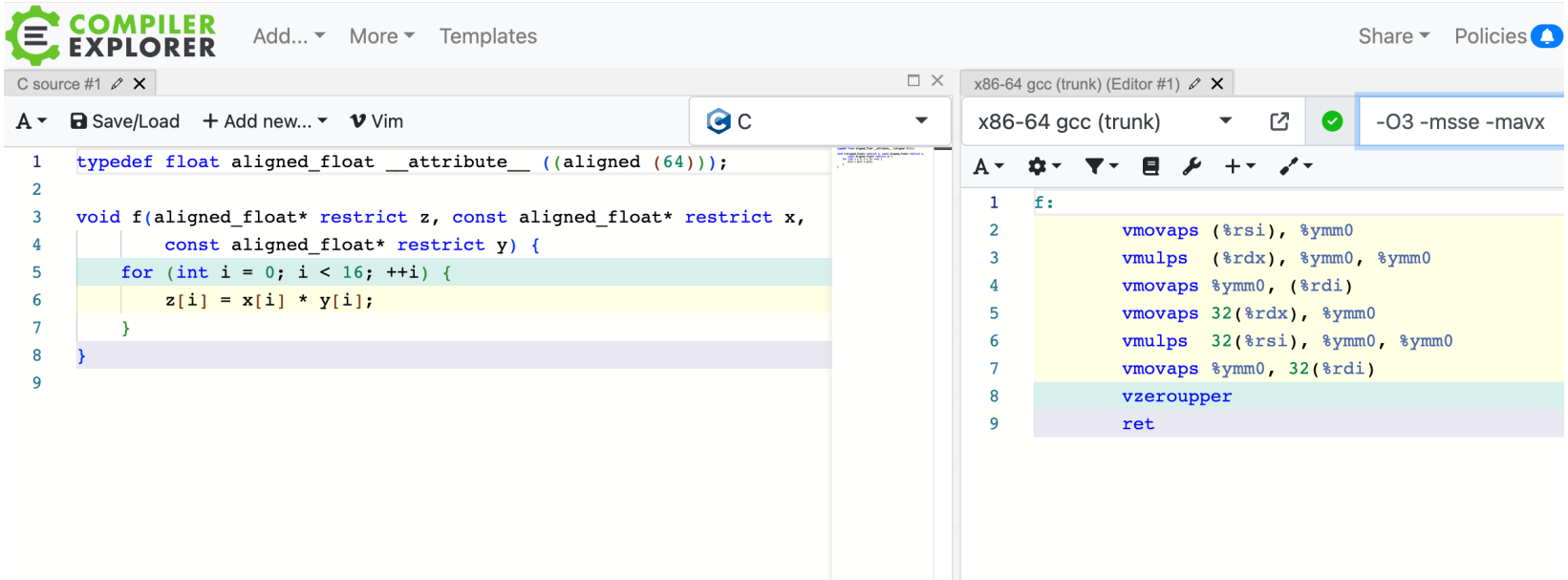
Example

- With SSE enabled, instead of unrolling the loop one element at a time, we can process 4 elements at a time (128 bits)
- Note that we can use the more efficient movaps instructions as the data is assumed aligned (unaligned data would cause the program to crash with these instructions)



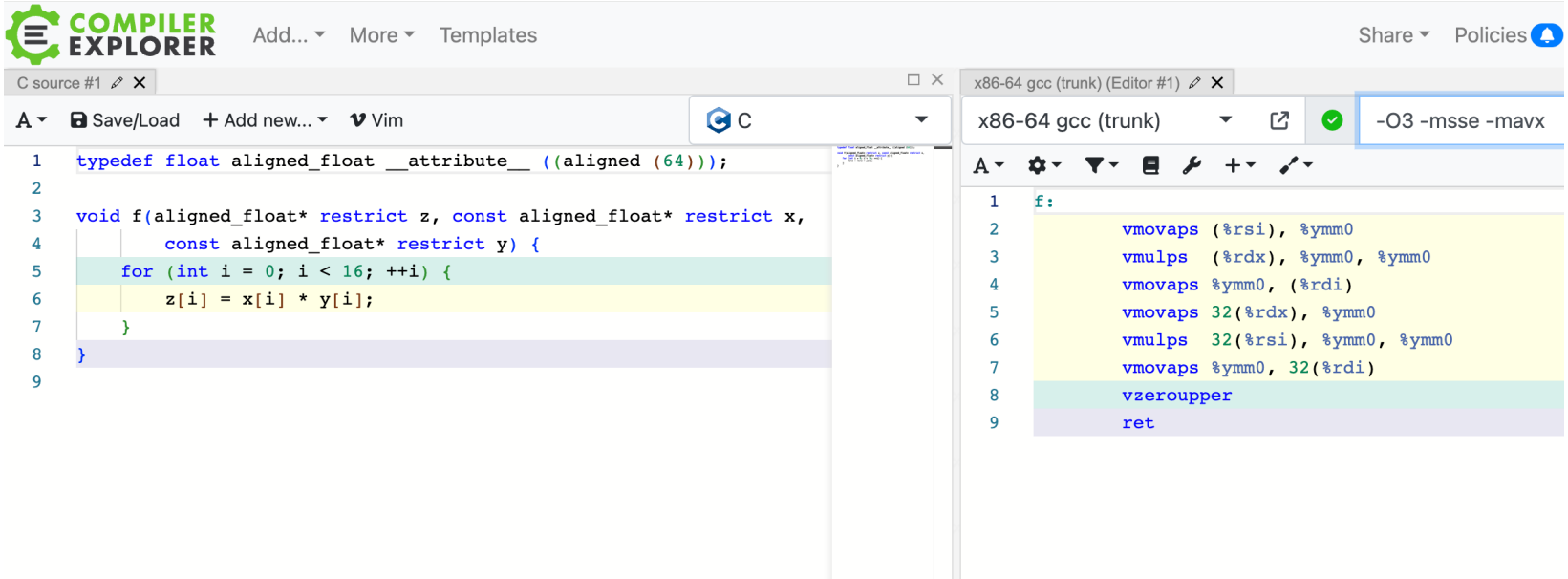
Example

- With AVX enabled, we can process 8 elements at a time (256 bits)



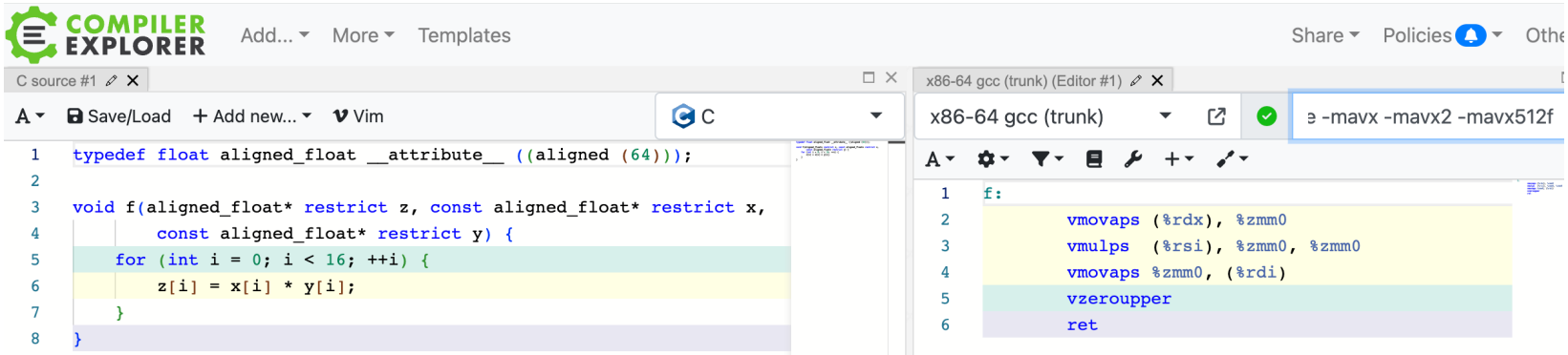
Example

- With AVX enabled, we can process 8 elements at a time (256 bits)



Example

- With AVX-512 enabled, we can process the entire array as one vector



Example

- Compute dot product of two long vectors of unsigned 8-bit integers

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

```
typedef uint8_t __attribute__((aligned(64))) uint8_t_aligned;
static uint8_t dot_product(int n,
                           uint8_t_aligned* __restrict x,
                           uint8_t_aligned* __restrict y) {
    uint8_t z = 0;
    for (int i = 0; i < n; ++i) {
        z += x[i] * y[i];
    }
    return z;
}
```

Example

- Compute dot product of two long vectors of unsigned 8-bit integers

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

```
typedef uint8_t __attribute__((aligned(64))) uint8_t_aligned;
static uint8_t dot_product(int n,
                           uint8_t_aligned* __restrict x,
                           uint8_t_aligned* __restrict y) {
    uint8_t z = 0;
    for (int i = 0; i < n; ++i) {
        z += x[i] * y[i];
    }
    return z;
}
```

Example

```
std::mt19937 rng(seed);
std::uniform_int_distribution<uint8_t> dist(0,255);

uint8_t_aligned* x = static_cast<uint8_t_aligned*>(aligned_alloc(64, n));
uint8_t_aligned* y = static_cast<uint8_t_aligned*>(aligned_alloc(64, n));

for (int i = 0; i < n; ++i)
  x[i] = dist(rng);
for (int i = 0; i < n; ++i)
  y[i] = dist(rng);

auto start = std::chrono::steady_clock::now();
uint8_t value = dot_product(n, x, y);
auto end = std::chrono::steady_clock::now();
auto diff = std::chrono::nanoseconds(end - start);

std::chrono::nanoseconds::rep time = diff.count();

printf("result: %" PRIx8 "\n", value);
printf("time elapsed: %ld ns\n", time);

free(x);
free(y);
```

Example

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

```
karppa@bayes:~$ g++ dot_product.cpp -o dot_product -O3
-mno-sse && ./dot_product 1073741824
result: e6
time elapsed: 698271600 ns
karppa@bayes:~$
```

Example

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

```
karppa@bayes:~$ g++ dot_product.cpp -o dot_product -O3 -msse
-msse2 -msse3 -msse4.2 && ./dot_product 1073741824
result: e6
time elapsed: 234855552 ns
```

Example

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

```
karppa@bayes:~$ g++ dot_product.cpp -o dot_product -O3 -msse -msse2
-msse3 -msse4.2 -mavx -mavx2 && ./dot_product 1073741824
result: e6
time elapsed: 225847231 ns
```

Example

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

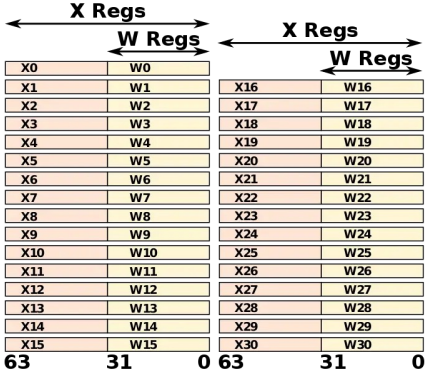
```
karppa@bayes:~$ g++ dot_product.cpp -o dot_product -O3 -msse -msse2
-msse3 -msse4.2 -mavx -mavx2 -mavx512f -mavx512bw && ./dot_product 1
073741824
result: e6
time elapsed: 199457182 ns
```

Example

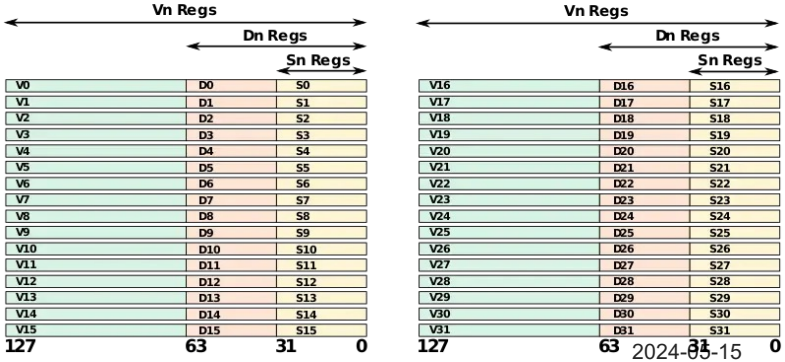
```
karppa@bayes:~$ g++ dot_product.cpp -o dot_product -O3 -msse -msse2
-msse3 -msse4.2 -mavx -mavx2 -mavx512f -mavx512bw -march=native &&
/dot_product 1073741824
result: e6
time elapsed: 198374051 ns
```

Aarch64

- The 64-bit ARM architecture has 31 64-bit general-purpose registers X0, .., X30
  - These registers hold integers
- The architecture implements **Advanced SIMD** extension, also known as **Neon**
  - There are 32 128-bit floating-point registers labeled V0, V1, .., V31
  - The register Vi is accessible also as Di, Si, Hi, or Bi, respectively, when one only wants to access the lower 64, 32, 16, or 8 bits, respectively
  - This enables treating the content, e.g., as a double- or single-precision floating-point numbers, or as 8–64-bit integers
  - Computations are done using IEEE 754 arithmetic
- Elements in the vector registers can be accessed as if elements of an array
- Indicating the proper size of elements allows one to perform elementwise vector operations
  - For example,
    ADD V0.2D, V1.2D, V2.2D
    treats the content of the registers as 2×64-bit integers and performs their elementwise addition



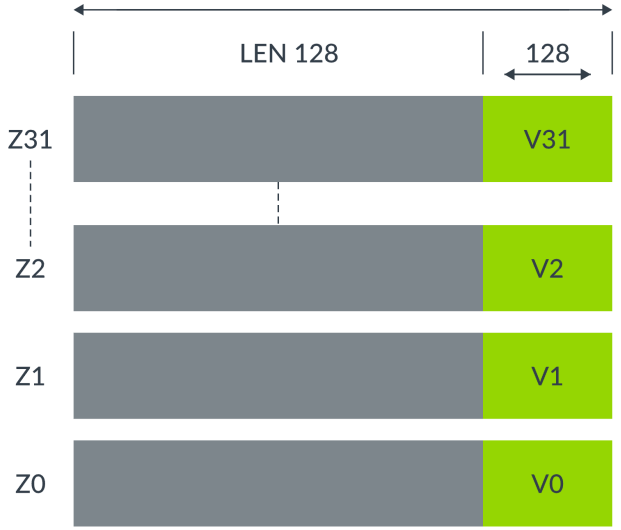
https://en.wikichip.org/wiki/File:arm_a64_registers.svg



https://en.wikichip.org/wiki/File:arm_a64_media_registers.svg

Scalable Vector Extension (SVE)

- SVE is a vector extension to the ARM architecture that differs from Intel's extensions by not prescribing a fixed length for the vector registers
  - CPU vendors can choose a suitable length between 128–2048 bits at 128-bit intervals
  - The same binary code can be executed regardless of the design choices by the CPU vendor
- SVE adds
  - 32 scalable vector registers Z0, …, Z31
  - 16 scalable predicate registers P0, ..., P15
- The lowest bits of the Z registers alias Neon V registers
- The Z registers can hold single and double-precision floating-points and 8–64-bit integers
- The predicate registers allow one to set bitmasks to enable or disable various vector operations on a per-element basis
- The architecture enables gather-scatter loads/stores from non-contiguous memory locations



Making use of vectorization in Python

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- The fundamental datatypes of Python are very much incompatible with vector extensions
  - Think, the `int`, the `list`, the `dictionary`, …
- This contributes to the inefficiency of Python code
- However, underlying C libraries are often aggressively optimized
- This holds true in particular with NumPy
- The elements of NumPy arrays are underlying C types, and as such, they are very efficiently vectorized
- If possible, you should try to phrase your code in terms of array operations
  - Avoid loops
  - Avoid accessing individual elements

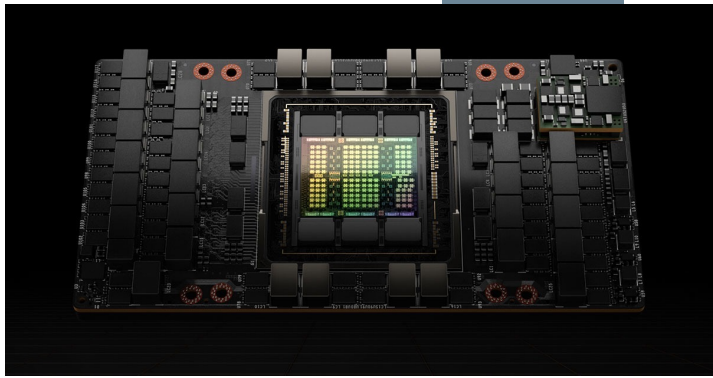
Graphics Processing Units (GPUs)

- GPUs have their background as **graphics accelerators**
  - Particularly **3D accelerators**
- Computer graphics involves computing large amounts of linear algebra operations simultaneously
- **General-Purpose computing on Graphics Processing Units (GPGPU)** refers to using the hardware for general-purpose computing (think: machine learning, blockchains, any kind of applications that are well suited for SIMD processing)
- Essentially, GPUs consist of thousands of small and relatively inefficient floating-point units that execute in parallel
  - Power is in the numbers
- GPUs excel in vectorized computations
- GPUs have very large register files

NVIDIA Tesla H100 GPU

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- 144 Streaming Multiprocessors (SM)
- Each SM has 128 FP32 CUDA cores
  - Total of 18,432 CUDA cores
- Also, each SM has 4 **tensor cores**
  - Total of 576 tensor cores
  - A tensor cores are special chips for Fused Multiply Add (FMA) operations
- 60 MiB of L2 cache
- 80 GiB of VRAM
- (Some variations to specs may occur depending on the form factor etc.)



https://www.nvidia.com/content/dam/en-zz/Solutions/gtcs22/data-center/h100/h100-og.jpg

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

Programming GPUs: CUDA

- CUDA (Compute Unified Device Architecture) is Nvidia's proprietary programming language for GPGPU programming
- Main unit of parallelism is the **thread**
- Unlike CPU threads, one thread more accurately corresponds to a lane in CPU computing
- Threads execute in a **warp** of 32 threads that all execute the same instructions simultaneously
  - Each thread can freely access distinct memory accesses (unlike vector instructions)
  - Threads may **shuffle** data between one another
- This means that loops and conditionals are slow in GPU programming: conditionals are typically handled by masking some threads by a predicate, so they still execute from both branches, but one branch is only marked as a NOP
- Typical idiom is that instead of a loop, we spawn a horde of threads, one thread per each element, and they execute in an unspecified order
  - If the code admits efficient parallelization, no synchronization is necessary, and this is super efficient
- In Nvidia-speak, the CPU-part of the computer is known as the **host** and the GPU as the **device**

BLAS

- Basic Linear Algebra Subroutines (BLAS) is a specification for low-level routines for performing most common linear algebra operations
  - Vector addition
  - Dot product
  - Matrix product
  - And so on…
- Originally specified in terms of FORTRAN 77, BLAS is the standard interface for low-level linear algebra operations
- The reference FORTRAN implementation is available at https://www.netlib.org/blas/ but is slow
- Multiple vendors have optimized implementations
  - Intel Math Kernel Library (MKL)
  - AMD Optimizing CPU Libraries (AOCL)
  - Arm Performance Libraries
  - NVIDIA cuBLAS
- Programs and libraries that make use of BLAS include NumPy, MATLAB, Julia, Mathematica, R, etc. etc.

Intel MKL

- The Intel Math Kernel Library (MKL) contains highly optimized implementations of BLAS, LAPACK, various other vector algebra operations, and basic primitives such as the FFT
- The code is aggressively optimized for Intel CPUs
  - Extensive use of vector extensions
  - Also, some use of multithreading
  - The code may behave suboptimally with x86 CPUs from other vendors
- The library comes with bindings for C, C++, and Fortran
- As the interface for, e.g., BLAS is highly standardized, the library can be easily plugged in as the backend for various software
  - For example, on x86-64, Anaconda comes with Intel MKL as the backend for NumPy, so you might have been using the MKL without knowing it

cuBLAS

- cuBLAS is Nvidia’s CUDA implementation of BLAS, targeting Nvidia GPUs
- As it implements the BLAS API, it is (relatively) straightforward to translate pre-existing CPU code into GPU-accelerated code
- Nvidia also provides other useful basic primitive libraries, such as cuFFT

AXPY and GEMM

- BLAS routines have names that are at most 6 letters long
  - This is a legacy requirement as FORTRAN 77 allows at most 6-letter identifiers
- The first character identifies datatype of vectors
  - `s` for single-precision float, `c` for single-precision complex, `d` for double-precision float, `z` for double-precision complex
- Sometimes there are multiple variants for special cases, indicated with a suffix character, such as `c` for conjugated vector
- We look at two examples:
  - `GEMM`, for General Matrix Multiply, is the main routine for computing matrix multiplication, and it computes $C \leftarrow \alpha AB + \beta C$ for scalar arguments α, β
  - Flags can be used to indicate that one or both of the input matrices needs to be transposed
  - `DGEMM` would be the variant that computes the product of double-precision matrices
  - `AXPY` computes $y \leftarrow \alpha x + y$ and is used to accumulate vectors with scalar argument α
  - `SAXPY` would be the variant that performs single-precision vector addition
- Next, we will implement some (simplified) variants of `SAXPY` and compare those

```
static void saxpy_simple(int n, float a, const float* x, float* y) {
  for (int i = 0; i < n; ++i)
    y[i] += a*x[i];
}
```

```
static void saxpy_avx512f(int n, float a, const float* x, float* y) {
  __m128 a4 = _mm_load_ss(&a);
  __m512 aa = _mm512_broadcastss_ps(a4);
  for (int i = 0; i < n; i += 16) {
    __m512 yy = _mm512_load_ps(y);
    __m512 xx = _mm512_load_ps(x);
    xx = _mm512_mul_ps(xx, aa);
    yy = _mm512_add_ps(yy, xx);
    _mm512_store_ps(y, yy);
    x += 16;
    y += 16;
  }
}
```

```
static void saxpy_avx512f_omp(int n, float a, const float* x, float* y) {
  __m128 a4 = _mm_load_ss(&a);
  __m512 aa = _mm512_broadcastss_ps(a4);
  #pragma omp parallel for
  for (int i = 0; i < n; i += 16) {
    float* yp = y + i;
    const float* xp = x + i;
    __m512 yy = _mm512_load_ps(yp);
    __m512 xx = _mm512_load_ps(xp);
    xx = _mm512_mul_ps(xx, aa);
    yy = _mm512_add_ps(yy, xx);
    _mm512_store_ps(yp, yy);
  }
}
```

```
float* x = static_cast<float*>(aligned_alloc(64, n*sizeof(float)));
float* y1 = static_cast<float*>(aligned_alloc(64, n*sizeof(float)));
float* y2 = static_cast<float*>(aligned_alloc(64, n*sizeof(float)));
float* y3 = static_cast<float*>(aligned_alloc(64, n*sizeof(float)));

std::mt19937 rng(seed);
std::uniform_real_distribution<float> dist;
float a = dist(rng);

for (int i = 0; i < n; ++i)
  y1[i] = dist(rng);
for (int i = 0; i < n; ++i)
  x[i] = dist(rng);
memcpy(y2, y1, sizeof(float)*n);
memcpy(y3, y1, sizeof(float)*n);
auto end = std::chrono::steady_clock::now();
std::chrono::duration<double> diff = end - start;
std::cout << "data generation took " << diff.count() << " s" << std::endl;

start = std::chrono::steady_clock::now();
saxpy_simple(n, a, x, y1);
end = std::chrono::steady_clock::now();
diff = end - start;
std::cout << "saxpy_simple took " << diff.count() << " s" << std::endl;

start = std::chrono::steady_clock::now();
saxpy_avx512f(n, a, x, y2);
end = std::chrono::steady_clock::now();
diff = end - start;
std::cout << "saxpy_avx512f took " << diff.count() << " s" << std::endl;

start = std::chrono::steady_clock::now();
saxpy_avx512f_omp(n, a, x, y3);
end = std::chrono::steady_clock::now();
diff = end - start;
std::cout << "saxpy_avx512f_omp took " << diff.count() << " s" << std::endl;

assert(array_equal(n,y1,y2));
assert(array_equal(n,y1,y3));
assert(array_equal(n,y2,y3));

free(x);
free(y1);
free(y2);
free(y3);
return EXIT_SUCCESS;
```

```
karppa@bayes:~$ g++ -O3 -march=native saxpy.cpp -o saxpy -fopenmp
karppa@bayes:~$ ./saxpy 1073741824
data generation took 18.4776 s
saxpy_simple took 4.23894 s
saxpy_avx512f took 0.73342 s
saxpy_avx512f_omp took 0.185025 s
```

```
__global__
static void saxpy_cuda_kernel(float a, const float* x, float* y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  y[i] += a*x[i];
}
```

```
start = std::chrono::steady_clock::now();
float* x_dev = nullptr;
float* y_dev = nullptr;
cudaMalloc(&x_dev, sizeof(float)*n);
cudaMalloc(&y_dev, sizeof(float)*n);
end = std::chrono::steady_clock::now();
diff = end - start;
std::cerr << "Allocating device memory took " << diff.count() << " s" << std::endl;
start = std::chrono::steady_clock::now();
cudaMemcpy(x_dev, x, sizeof(float)*n, cudaMemcpyHostToDevice);
cudaMemcpy(y_dev, y2, sizeof(float)*n, cudaMemcpyHostToDevice);
end = std::chrono::steady_clock::now();
diff = end - start;
std::cerr << "Copying data from host to device took " << diff.count() << " s" << std::endl;

start = std::chrono::steady_clock::now();
saxpy_cuda_kernel<<<num_blocks,num_threads>>>(a, x_dev, y_dev);
cudaDeviceSynchronize();
end = std::chrono::steady_clock::now();
diff = end - start;
std::cerr << "saxpy_cuda_kernel took " << diff.count() << " s" << std::endl;

start = std::chrono::steady_clock::now();
cudaMemcpy(y2, y_dev, sizeof(float)*n, cudaMemcpyDeviceToHost);
end = std::chrono::steady_clock::now();
diff = end - start;
std::cerr << "Copying data from device to host took " << diff.count() << " s" << std::endl;
```

```
cublasInit();
start = std::chrono::steady_clock::now();
cublasSetVector(n, sizeof(float), x, 1, x_dev, 1);
cublasSetVector(n, sizeof(float), y3, 1, y_dev, 1);
end = std::chrono::steady_clock::now();
diff = end - start;
std::cerr << "Copying cublas data from host to device took " << diff.count() << " s" << std::endl;

start = std::chrono::steady_clock::now();
cublasSaxpy(n, a, x_dev, 1, y_dev, 1);
cudaDeviceSynchronize();
end = std::chrono::steady_clock::now();
diff = end - start;
std::cerr << "cublasSaxpy took " << diff.count() << " s" << std::endl;

start = std::chrono::steady_clock::now();
cublasGetVector(n, sizeof(float), y_dev, 1, y3, 1);
end = std::chrono::steady_clock::now();
diff = end - start;
std::cerr << "Copying cublas data from device to host took " << diff.count() << " s" << std::endl;

cublasShutdown();
```

```
(cuda) karppa@bayes:~$ nvcc -O3 -lcublas saxpy.cu -o saxpy -arch sm_70
(cuda) karppa@bayes:~$ ./saxpy 1073741824
using 256 threads and 4194304 blocks
data generation took 44.2876 s
Allocating device memory took 1.10985 s
saxpy_simple took 5.39743 s
Copying data from host to device took 4.24819 s
saxpy_cuda_kernel took 0.015806 s
Copying data from device to host took 1.12808 s
Copying cublas data from host to device took 4.16786 s
cublasSaxpy took 0.016197 s
Copying cublas data from device to host took 1.18198 s
```

CuPy

- One way to make use of GPUs through Python is to use CuPy
- CuPy is essentially a GPU version of NumPy
- In most ordinary cases, one just needs to replace `numpy.ndarray` classes with `cupy.ndarray` classes; there is an analog of almost all functions in CuPy
- GPU arrays can be constructed (e.g., from NumPy arrays that are on host memory) with `Y = cp.asarray(X)`
  - By default, this is **non-blocking**, that is, it is done asynchronously in the background; adding argument `blocking = True` is necessary for measuring times
- Data can be downloaded from the GPU with `X = cp.asnumpy(Y)`
  - Again, this is asynchronous, so `blocking = True` may be useful
- As operations can be launched as asynchronous CUDA kernels, when performing measurements, explicit synchronization can be useful
  - This can be achieved with `cp.cuda.Stream.null.synchronize()`
- Otherwise we write just like normal NumPy code

SAXPY in CuPy

```
import numpy as np
import cupy as cp
import sys
import time

def saxpy_numpy(a, x, y):
    y += a*x

def saxpy_cupy(a, x, y):
    y += a*x
    cp.cuda.Stream.null.synchronize()
```

SAXPY in CuPy

```
rng = np.random.default_rng(1234)
n = int(sys.argv[1])
start = time.time()
a = rng.random(dtype = np.float32)
x = rng.random(size = n, dtype = np.float32)
y = rng.random(size = n, dtype = np.float32)
end = time.time()
print(f'Data generation took {end-start} s')

# the first access to the GPU is slower than later
_ = cp.asarray([0], blocking = True)

start = time.time()
x_gpu = cp.asarray(x, blocking = True)
y_gpu = cp.asarray(y, blocking = True)
end = time.time()
print(f'Uploading data to the GPU took {end-start} s')
```

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

SAXPY in CuPy

```
start = time.time()
saxpy_numpy(a, x, y)
end = time.time()
print(f'saxpy_numpy took {end-start} s')

start = time.time()
saxpy_cupy(a, x_gpu, y_gpu)
end = time.time()
print(f'saxpy_cupy took {end-start} s')

start = time.time()
y2 = cp.asnumpy(y_gpu, blocking = True)
end = time.time()
print(f'Downloading data from the GPU took {end-start} s')

assert np.array_equal(y,y2)
```

SAXPY in CuPy

```
(cupy) karppa@bayes:~$ python3 saxpy_cuda.py 1073741824
Data generation took 29.268832445144653 s
Uploading data to the GPU took 4.3902366161346436 s
saxpy_numpy took 2.389920234680176 s
saxpy_cupy took 0.0700535774230957 s
Downloading data from the GPU took 1.8804490566253662 s
```

Nvidia RAPIDS

- RAPIDS is a data science and machine learning oriented GPU library to enable rapid development of GPU-accelerated Python code that is largely compatible or at least similar to familiar CPU coce
- https://rapids.ai/
- Sublibraries include
  - GPU versions of Pandas dataframes with cuDF
  - Scikit-learn-like machine learning library cuML
  - Networkx-like library called cuGraph
  - Spatial analytics library cuSpatial
  - Approximate nearest neighbors and clustering with cuVS
  - Linear algebra and nearest neighbors etc. in RAFT
  - and others

Example: nearest neighbors using RAFT

- Linear scan for *k*-nearest neighbors can be done with `pylibraft.neighbors.brute_force.knn`
- See https://docs.rapids.ai/api/raft/nightly/pylibraft_api/neighbors/#brute-force
- Like sklearn, this returns a `(D,I)` tuple where the matrix `D` contains distances from the query point (rows) to the k-nearest neighbors (columns), and `I` the row index in the original dataset
- Note: the arrays returned is not a CuPy array (although CuPy arrays are compatible as arguments), so they need to be converted to CuPy arrays with `cp.asarray` (this does not create a copy so this is a cheap operation) if we want to use all CuPy features
- Also, the function only supports single-precision floating-point data as input
- A full example is not presented here, as it is the problem 5 in Assignment 7