SIMD Instruction Set Extensions for Multimedia

What is SIMD?

- Single instruction, multiple data (SIMD) is a type of parallel processing
- SIMD also known as short vector processors
- SIMD can be internal (part of the hardware design) and it can be directly accessible through an instruction set architecture (ISA)
- Perform the same operation on multiple data points simultaneously.
- Exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but each unit performs the exact same instruction at any given moment (just with different data).
- In contrast to vector architectures,SIMD extensions have three major omissions: no vector length register, no strided or gather/scatter data transfer instructions, and no mask registers.

Where is it used ?

- SIMD instructions are widely used to process 3D graphics
- The applications of SIMD include image processing, video and sound applications.
- SIMD is particularly applicable to common tasks such as adjusting the contrast in a digital image or adjusting the volume of digital audio.
- Also used in cryptography.
- Modern graphics cards with embedded SIMD , so the trend of general-purpose computing on GPUs may lead to wider use of SIMD in the future.

SIMD Instruction Set Extensions for Multimedia

- Started with the simple observation that many media applications operate on narrower data types than the 32-bit processors were optimized for.By partitioning the carry chains within, say, a 256-bit adder, a processor could perform simultaneous operations on short vectors of thirty-two 8-bit operands, sixteen 16-bit operands, eight 32-bit operands, or four 64-bit operands.
- For example, take image processing. A common operation found in programs such as Photoshop would be to reduce the amount of red in an image by half. Assuming a 32-bit traditional processor that is Single Instruction, Single Data (SISD) and a 24 bit image, the information for one pixel would be put into one 32-bit word for processing. Each pixel would have to be processed individually. In a 128-bit SIMD processor, four 32-bit pixels could be packed into one 128-bit word and all four pixels could be processed simultaneously. Theoretically, this translates to a four fold improvement in processing time.

Why are multimedia SIMD extensions so popular?

- They initially cost little to add to the standard arithmetic unit and they were easy to implement.
- They require scant extra processor state compared to vector architectures, which is always a concern for context switch times.
- You need a lot of memory bandwidth to support a vector architecture, which many computers don't have. SIMD doesnt need this.
- SIMD does not have to deal with problems in virtual memory when a single instruction can generate 32 memory accesses and any of which can cause a page fault.
- Another advantage of short, fixed length "vectors" of SIMD is that it is easy to introduce instructions that can help with new media standards, such as instructions that perform permutations or instructions that consume either fewer or more operands than vectors can produce.

SIMD in modern x86 architecture

- The MMX instructions added in 1996 repurposed the 64-bit floating-point registers, so the basic instructions could perform eight 8-bit operations or four 16-bit operations simultaneously.
- The Streaming SIMD Extensions (SSE) successor in 1999 added 16 separate registers (XMM registers) that were 128 bits wide, so now instructions could simultaneously perform sixteen 8-bit operations, eight 16-bit operations, or four 32-bit operations.
- Intel soon added double-precision SIMD floating-point data types via SSE2 in 2001, SSE3 in 2004, and SSE4 in 2007. Instructions with four single-precision floating-point operations or two parallel double-precision operations increased the peak floating-point performance of the x86 computers, as long as programmers placed the operands side by side.

SIMD in modern x86 architecture (continued)

- The Advanced Vector Extensions (AVX), added in 2010, doubled the width of the registers again to 256 bits (YMM registers) and thereby offered instructions that double the number of operations on all narrower data types
- AVX2 in 2013 added 30 new instructions
- AVX-512 in 2017 doubled the width again to 512 bits (ZMM registers), doubled the number of the registers again to 32

In general, the goal of these extensions has been to accelerate carefully written libraries rather than for the compiler to generate them. But recent x86 compilers are trying to generate such code, particularly for floating-point- intensive applications.

Programming Multimedia SIMD Architectures

- Given the ad hoc nature of the SIMD multimedia extensions, the easiest way to use these instructions has been through libraries or by writing in assembly language.
- Recent extensions have become more regular, giving compilers a more reason- able target. By borrowing techniques from vectorizing compilers, compilers are starting to produce SIMD instructions automatically.
- However, programmers must be sure to align all the data in memory to the width of the SIMD unit on which the code is run to prevent the compiler from generating scalar instructions for otherwise vectorizable code.

In the next slide, We can see the compiler automatically generating SIMD instructions. The example code is written in C++, and it is code for saxpy(single precision a*X plus Y)

SAXPY with compiler generated x86 sse instructions

```cpp
#include <array>
#include <iostream>
#include <numeric>

constexpr int N = 32; ▸ 3 refs
template <size_t N> ▸ 3 refs
void saxpy(std::array<float, N> &X, std::array<float, N> &Y, float a) { ▸ 1
  for (int i = 0; i < N; i++) { ▸ 5 refs
    Y[i] = a * X[i] + Y[i];
  }
}
int main() { ▸ 0 ref
  std::array<float, N> X; ▸ 3 refs
  std::array<float, N> Y; ▸ 4 refs
  // fills a and b with 1..N
  std::iota(X.begin(), X.end(), 1);
  std::iota(Y.begin(), Y.end(), 1);
  // will use simd intrinsics
  saxpy(X, Y, 10.0);
  for (int i = 0; i < N; i++) { ▸ 3 refs
    printf("%f ", Y[i]);
  }
  printf("\n");
}
```

SAXPY with compiler generated x86 sse instructions(continued)

```
11cb:   0f 28 05 3e 0e 00 00    movaps xmm0,XMMWORD PTR [rip+0×e3e]
11d2:   0f 29 84 24 80 00 00    movaps XMMWORD PTR [rsp+0×80],xmm0
11d9:   00
11da:   0f 28 0d 3f 0e 00 00    movaps xmm1,XMMWORD PTR [rip+0×e3f]
11e1:   0f 29 8c 24 90 00 00    movaps XMMWORD PTR [rsp+0×90],xmm1
11e8:   00
11e9:   0f 28 15 40 0e 00 00    movaps xmm2,XMMWORD PTR [rip+0×e40]
11f0:   0f 29 94 24 a0 00 00    movaps XMMWORD PTR [rsp+0×a0],xmm2
11f7:   00
11f8:   0f 28 1d 41 0e 00 00    movaps xmm3,XMMWORD PTR [rip+0×e41]
11ff:   0f 29 9c 24 b0 00 00    movaps XMMWORD PTR [rsp+0×b0],xmm3
1206:   00
1207:   0f 28 25 42 0e 00 00    movaps xmm4,XMMWORD PTR [rip+0×e42]
120e:   0f 29 a4 24 c0 00 00    movaps XMMWORD PTR [rsp+0×c0],xmm4
1215:   00
1216:   0f 28 2d 43 0e 00 00    movaps xmm5,XMMWORD PTR [rip+0×e43]
121d:   0f 29 ac 24 d0 00 00    movaps XMMWORD PTR [rsp+0×d0],xmm5
1224:   00
1225:   0f 28 35 44 0e 00 00    movaps xmm6,XMMWORD PTR [rip+0×e44]
```

clang generated x86 code with -O2 compilation flag,uses xmm registers

Gains By implicit SIMD Usage

```
1   #![feature(portable_simd)]
  1 #![allow(dead_code)]
  2 use core_simd::*;
  3 use std::time::Instant;
  4
  5 const N: usize = 10000000;
  6 const X: usize = 32;
  7 fn use_simd(f: Vec<Simd<i8, X>>) -> Vec<Simd<i8, X>> {
  8     f.into_iter().map(|x| x * 2).collect::<Vec<Simd<i8, X>>>()
  9 }
 10 fn no_simd(f: Vec<i8>) -> Vec<i8> {
 11     f.into_iter().map(|x| x * 2).collect::<Vec<i8>>()
 12 }
 13 fn timer<T, F: FnOnce() -> Vec<T>>(f: F) -> Vec<T> {
 14     let start = Instant::now();
 15     let g = f();
 16     println!("{:?}", start.elapsed());
 17     g
 18 }
```

Gains By implicit SIMD Usage(continued)

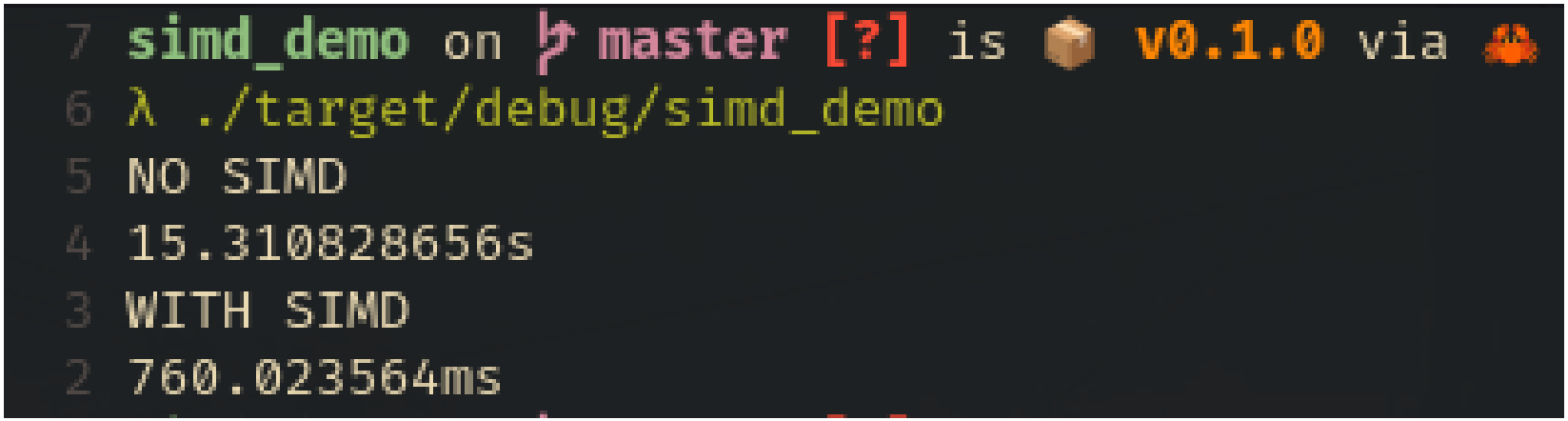
```rust
fn main() {
    println!("NO SIMD");
    let f = vec![2; X * N];
    let g = timer(move || no_simd(f));

    println!("WITH SIMD");
    let a = i8x32::from_array([2; X]);
    let f = vec![a; N];
    let h = timer(move || use_simd(f))
        .into_iter()
        .map(|x| x.as_array().to_vec())
        .into_iter()
        .flatten()
        .collect::<Vec<i8>>();

    assert_eq!(g, h);
}
```

Gains By implicit SIMD Usage(continued)

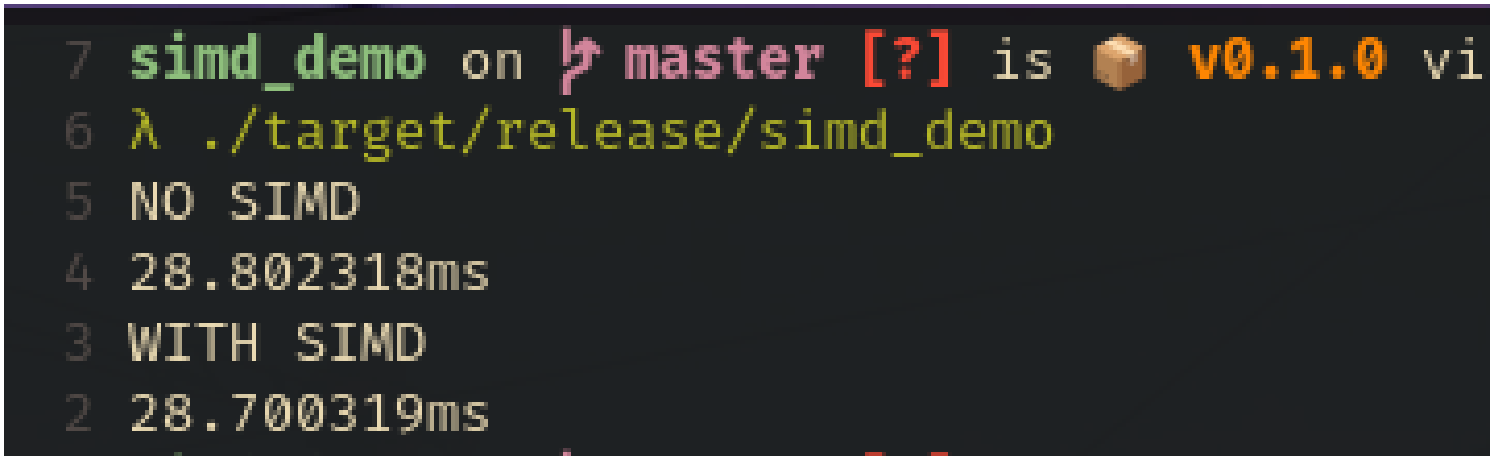
This was debug build of the code



The code with explicit simd instruction is almost 20 times as fast

Gains By implicit SIMD Usage(continued)

This was release build of the code(optimized build)



- Both take almost same exact time
- Compiler able to figure out that the `no_simd` function can be easily vectorized and able to optimize it by producing simd instruction.
- The `no_simd` and `use_simd` pretty much produce same code in optimized build.

End

THANK YOU