CS60203: Design Optimization of Computing Systems *August 24, 2024*

Advanced SIMD Optimisations

Instructor: Mainack Mondal *Scribe-By: <Tanishq Prasad >*

1 Recap: Motivation behind SIMD

- **Enhanced Performance:** Boosts performance by exploiting data parallelism, executing multiple operations simultaneously, reducing the need for loops and branching. [1]
- **Compact Machine Code:** Reduces the number of instructions required to perform a task, thereby reducing the size of the machine code. [1]
- **Optimized Memory Access:** Optimizes memory access by processing contiguous data elements, reducing latency and boosting overall performance. [1]

2 Memory Alignment

2.1 Why do we need memory alignment?

1. Faster memory access and cache efficiency
2. Easier bounds checking and debugging
3. Easier low-level programming and SIMD optimization (easier vectorization)

2.2 Memory Alignment in C/C++

2.2.1 Aligned Memory Allocation

Pointers are aligned on a 16-byte boundary as a mandate, leading to faster load and stores due to the reasons mentioned above. It can cause segmentation faults if misaligned data is loaded. Memory alignment in most systems is typically done in powers of 2. This is because memory alignment is closely tied to how processors access memory, and powers of 2 are naturally aligned with the binary nature of digital systems.

To achieve this in C/C++, we can use compiler directives such as:-

1. GCC specific [FSF15, 6.38]
   __attribute__ ((aligned (ALIGN)))
2. C++11 standard [ISO11, 6.2.8, 7.22.3]
   aligned_alloc(size_t alignment, size_t size)
   alignas(expression) and alignas(type_id) [2]

We can use the **alignof** operator to get the alignment in bytes of the specified type as a value of type size_t.

2.2.2 Unaligned Memory Allocation

It can work with any pointers but has some computational overhead as multiple reads might be required and additional code to be written to extract the data.

To achieve this in C/C++, we can use compiler directives such as:-

1. GCC specific [FSF15, 6.38]
   __attribute__ ((packed))

2.2.3 Memory Alignment Examples

1. struct V { short s[3]; } __attribute__ ((aligned(8)));
   size of V = 6 bytes + 2 bytes(padding)
2. char c[3] __attribute__ ((aligned(8)));
   size of c = 3 bytes + 5 bytes(padding)
3. struct Z { short s[10]; } __attribute__ ((aligned(8)));
   size of Z = 20 bytes + 4 bytes(padding)
4. struct A { char a; int b; } __attribute__ ((packed));
   size of A = 1 byte + 4 bytes(no padding because packed)
5. struct alignas(32) X { int x; };[2]
   size of X = 4 bytes + 28 bytes(padding)

2.3 Pointer Aliasing

Pointer aliasing refers to the situation where the same memory location can be accessed using different pointer names. The strict aliasing rule in C/C++ means that pointers are assumed not to alias if they point to fundamentally different types. Aliasing introduces strong constraints on program execution order. It can also result in code overhead. [3]

Example:

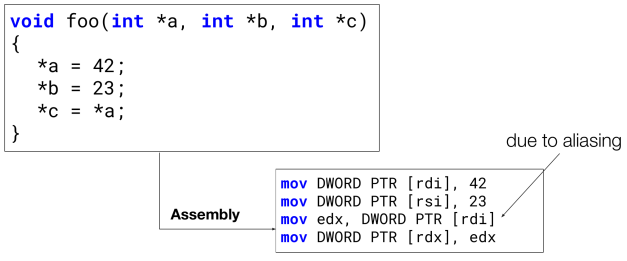

Figure 1: Code overhead due to pointer aliasing

By adding the **restrict** keyword into this code example, the compiler can optimize the resulting assembly language to increase the parallelization of the hardware operations. The following

example shows that using the restrict keyword to prevent aliasing uses fewer clock cycles to complete the same operation.

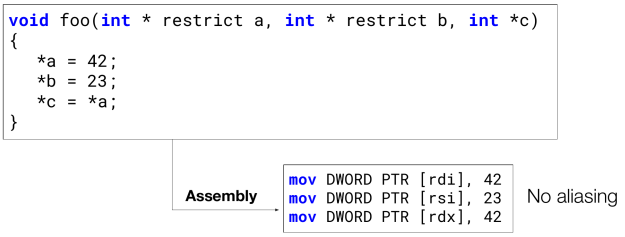

Figure 2: Preventing pointer aliasing via restrict

Caveats:

1. **restrict** needs to be used carefully
2. Programmer is responsible for proper usage
3. Mishandling can lead to wrong programs

3 GCC Auto Vectorization

3.1 Auto vectorization related Flags

1. -O -ftree-vectorize
   Activates auto vectorization
2. -O3
   Turns all optimisations including small string optimisations
3. -fopt-info-vec, -fopt-info-vec-missed
   Prints information about the optimisations applied and also about the missed ones [4]
4. -march=native
   Use instructions supported by the local CPU
5. -falign-functions=32, -falign-loops=32
   Aligns the address of functions and loops to be a multiple of 32 bytes

3.2 Explanation with an Example

```
# define SIZE (1 << 16)
void simpleLoop ( double * a, double * b)
{
    for ( int i = 0; i < SIZE ; i++)
    {
        a[i] += b[i];
    }
}
```

Figure 3: Simple code compiled with the flags -O, -ftree-vectorize, -fopt-info-vec, -fopt-info-vec-missed, -march=native

The compilation leads to **vectorization** and **versioning**, a technique used to generate multiple versions of the same function, each optimized for a specific set of input parameters. This is done to avoid the overhead of checking the input parameters at runtime.
The two versions of the asm output:-

```
.L3:
   vmovupd   (%rdi,%rax), %ymm1
   vaddpd    (%rsi,%rax), %ymm1, %ymm0
   vmovupd   %ymm0, (%rdi,%rax)
   addq      $32, %rax
   cmpq      $524288, %rax
   jne    .L3
   ret
```

Figure 4: Loop vectorized using 32 byte vectors (taken as vectors)

```
.L2:
   vmovsd    (%rdi,%rax), %xmm0
   vaddsd    (%rsi,%rax), %xmm0, %xmm0
   vmovsd    %xmm0, (%rdi,%rax)
   addq      $8, %rax
   cmpq      $524288, %rax
   jne    .L2
   ret
```

Figure 5: Loop versioned because of possible pointer aliasing (taken as scalars)

We can use **restrict** to prevent versioning.

```
# define SIZE (1 << 16)
void simpleLoop ( double * restrict a, double * restrict b)
{
    for ( int i = 0; i < SIZE ; i++)
    {
        a[i] += b[i];
    }
}
```

Figure 6: Loop vectorized using 32 byte vectors (no versioning)

```
.L2:
   vmovupd   (%rdi,%rax), %ymm1
   vaddpd    (%rsi,%rax), %ymm1, %ymm0
   vmovupd   %ymm0, (%rdi,%rax)
   addq      $32, %rax
   cmpq      $524288, %rax
   jne    .L2
   ret
```

Figure 7: asm output of the above code

Further analysis: The operations used in the assembly are **vmovupd** and **vaddpd** are for unaligned arguments for the load and store instructions.
We can align a and b to 32 bytes using the following paradigm:-

```
#define SIZE (1 << 16)
#define GCC_ALN(var, alignment) \
__builtin_assume_aligned(var, alignment)

void optimized_Loop(double *restrict a, double *restrict b)
{
    a = (double *)GCC_ALN(a, 32);
    b = (double *)GCC_ALN(b, 32);
    for (int i = 0; i < SIZE; i++)
    {
        a[i] += b[i];
    }
}
```

Figure 8: Using GCC_ALN to align to 32 bytes

```
.L2:
    vmovapd     (%rdi,%rax), %ymm1
    vaddpd      (%rsi,%rax), %ymm1, %ymm0
    vmovapd     %ymm0, (%rdi,%rax)
    addq        $32, %rax
    cmpq        $524288, %rax
    jne     .L2
    ret
```

Figure 9: asm output of the above code (using aligned operations now)

Similarly, we can use **alignas(32)** to achieve the same assembly code as previous.

3.3 Auto vectorizaion requirements and limitations

1. The loop should consist primarily of straight-line code. There should be no jumps or branches such as switch statements, but masked assignments are allowed, including if-then-else constructs that can be interpreted as masked assignments. [5]

2. The loop should be countable, i.e. the number of iterations should be known before the loop starts to execute, though it need not be known at compile time. Consequently, there should be no data-dependent exit conditions, with the exception of very simple search loops. [5]

3. There should be no backward loop-carried dependencies. For example, the loop must not require statement 2 of iteration 1 to be executed before statement 1 of iteration 2 for correct results. This allows consecutive iterations of the original loop to be executed simultaneously in a single iteration of the unrolled, vectorized loop. [5]

4. There should be no special operators and no function or subroutine calls, unless these are inlined, either manually or automatically by the compiler, or they are SIMD (vectorized) functions. Intrinsic math functions such as sin(), log(), fmax(), etc. are allowed, since the compiler runtime library contains SIMD (vectorized) versions of these functions. See the comments section for a more extensive list. [5]

5. If a loop is part of a loop nest, it should normally be the inner loop.

4 Fused-Multiply-Add (FMA)

It means to combine multiplication and addition by computing (x*y+z) as a single operation. It was introduced by IBM in their POWER architecture in 1997.

4.1 FMA Family

1. FMA(x, y, c): Fused Multiply-Add → x * y + c
2. FMS(x, y, c): Fused Multiply-Subtract → x * y - c
3. FNMA(x, y, c): Fused Negative Multiply-Add → - x * y + c
4. FNMS(x, y, c): Fused Negative Multiply-Subtract → - x * y + c

4.2 Why FMA?

If we multiply and add in separate steps, the multiplication step is first computed. The result is rounded to double precision followed by the addition step, and then that result is again rounded to double precision, which leads to lesser accuracy. It also leads to adding extra dependency between the consecutive multiply and add instructions, creating pipeline stalls. In FMA, all this is done in a single step and computed without rounding after multiplication.
In FMA, all this is done in a single step and computed without rounding after multiplication.

4.3 Advantages of FMA

1. Higher Accuracy - Only one rounding step as opposed to two.
2. Higher performance - Lower latency and better pipelining.
   AMD Bulldozer: FP ADD/MUL/FMA latency = 5

References

[1] CelerData website. What is Single Instruction, Multiple Data (SIMD)? https://celerdata.com/glossary/single-instruction-multiple-data-simd, Dec 2024.

[2] cppreference. alignas specifier. https://en.cppreference.com/w/cpp/language/alignas, Aug 2024.

[3] AMD Technical Information Portal. Pointer aliasing. https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding/Pointer-Aliasing, Jun 2024.

[4] malat. Understanding gcc 4.9.2 auto-vectorization output. https://stackoverflow.com/questions/30305830/understanding-gcc-4-9-2-auto-vectorization-output, Feb 2019.

[5] intel. Vectorizable loop requirements. https://www.intel.com/content/www/us/en/developer/articles/technical/requirements-for-vectorizable-loops.html, Apr 2019.