Advanced SIMD Optimizations

Mainack Mondal
Sandip Chakraborty

CS 60203
Autumn 2024

Outline

- SSE Memory Operations
  - Kinds of Memory operations
  - Memory Alignment
  - Pointer Aliasing

- GCC Auto Vectorization

- Fused-Multiply-Add (FMA)
  - What is FMA? Why FMA?
  - Family of FMA
  - Advantages

SSE Memory Operations

(Slides partially taken from Marat Dukhan and Andreas Schmitz)

SSE Memory Operations

We have already seen two kinds of load/store instructions

- Aligned (eg: `_mm_load_ps`)
  - Mandate that pointer is aligned on 16-byte boundary
  - Pros: Faster loads/stores (why?)
  - Cons: Can cause segmentation faults if misaligned data is loaded

- Mis-Aligned (eg: `_mm_loadu_ps`)
  - Can work with any pointers
  - But, has computation overhead
    - Multiple reads necessary
    - Additional code to extract the data

Memory Alignment in C/C++

But … How do you align memory?

The answer is : Using specific ***compiler directives***

- Directives to control alignment behavior:
  - GCC specific [FSF15, 6.38]
    - `__attribute__ ((aligned (ALIGN)))`
    - `__attribute__ ((packed))`
  - C11 Standard [ISO 11, 6.2.8, 7.22.3]
    - `aligned_alloc(size_t alignment, size_t size)`
    - `_Alignas(expression)` and `_Alignas(type)`

Memory Alignment Examples

- `struct V{short s[3];} __attribute__ ((aligned(8));`
  - size of V = 6 bytes + 2 bytes (padding)

- `char c[2] __attribute__((aligned(8)));`
  - size of c = 2 bytes + 6 bytes (padding)

- `struct A{char a; int b;} __attribute__((packed));`
  - size of A = 1 byte + 4 bytes (no padding because packed)

Pointer Aliasing

- Refers to memory addressed by different names
- Example: `char b; char *a = &b;`
- Needs to be considered by the compiler
- Can result in code overhead (will see in examples)

Pointer Aliasing Examples

```c
void foo(int *a, int *b, int *c)
{
   *a = 42;
   *b = 23;
   *c = *a;
}
```

Assembly

```
mov DWORD PTR [rdi], 42
mov DWORD PTR [rsi], 23
mov edx, DWORD PTR [rdi]
mov DWORD PTR [rdx], edx
```

due to aliasing

Pointer Aliasing Examples

How do you stop aliasing? Use `restrict` keyword (C99 standard)

```c
void foo(int * restrict a, int * restrict b, int *c)
{
    *a = 42;
    *b = 23;
    *c = *a;
}
```

Assembly

```
mov DWORD PTR [rdi], 42
mov DWORD PTR [rsi], 23
mov DWORD PTR [rdx], 42
```

No aliasing

Pointer Aliasing - Remarks

- `restrict` needs to be used carefully
- Programmer is responsible for proper usage
- Mishandling can lead to wrong programs

Outline

- SSE Memory Operations
  - Kinds of Memory operations
  - Memory Alignment
  - Pointer Aliasing
- **GCC Auto Vectorization**
- Fused-Multiply-Add (FMA)
  - What is FMA? Why FMA?
  - Family of FMA
  - Advantages

GCC Auto Vectorization

Auto-vectorization related Flags

- `-O -ftree-vectorize`
  - Activates auto-vectorization

- `-O3`
  - Optimizations including auto vectorization

- `-fopt-info-vec, -fopt-info-vec-missed`
  - List (not) vectorized loops + additional information

- `-march=native`
  - Use instructions supported by the local CPU

- `-falign-functions=32, -falign-loops=32`
  - Aligns the address of functions / loops to be a multiple of 32 bytes

GCC Auto-Vectorization Examples

Simple Loop

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

Compiler Output:
optimized: loop vectorized using 32 byte vectors
optimized: loop versioned for vectorization because of possible aliasing

compile this with:

```
-O -ftree-vectorize, -fopt-info-vec,
-fopt-info-vec-missed, -march=native
```

GCC Auto-Vectorization Examples (contd.)

Simple Loop (asm output)

2 versions

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

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

Let’s make it better

GCC Auto-Vectorization Examples (Contd.)

Improved Loop

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

Compiler Output:
optimized: loop vectorized using 32 byte vectors

GCC Auto-Vectorization Examples (contd.)

Improved Loop (asm output)

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

No versions

Scope for further improvement ?

- Yes, it uses unaligned load/store

Let’s make it even better

GCC Auto-Vectorization Examples (Contd.)

Optimised Loop

```
#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];
    }
}
```

Compiler Output:
optimized: loop vectorized using 32 byte vectors

GCC Auto-Vectorization Examples (contd.)

Optimized Loop (asm output)

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

aligned load/store

GCC Auto-Vectorization Examples (Contd.)

Optimised Loop (C11 Compatible)

```
#include <stdalign.h>
#define SIZE (1 << 16)

struct data {
    alignas(32) double vec[SIZE];
};

void optimized_Loop(struct data *restrict a, struct data *restrict b){
    for (int i = 0; i < SIZE; i++)
        a->vec[i] += b->vec[i];
}
```

Compiler Output:
optimized: loop vectorized using 32 byte vectors

Note: This produces the same assembly code as previous

Auto-vectorization Requirements and Limitations

- Countable loops
- No backward loop-carried dependencies
- No function calls : Except vectorizable math functions e.g. sin, sqrt,...
- Straight-line code (only one control flow: no switch)
- Loop to be vectorized must be innermost loop if nested

References: Intel: requirements for vectorizable loops

Outline

- SSE Memory Operations
  - Kinds of Memory operations
  - Memory Alignment
  - Pointer Aliasing

- GCC Auto Vectorization

- **Fused-Multiply-Add (FMA)**
  - What is FMA? Why FMA?
  - Family of FMA
  - Advantages

FMA: Fused-Multiply-Add

What is FMA?

- Basically combining multiplication and Addition
  - computes `x*y + z`
- Introduced by IBM in their POWER Architecture in 1990s

FMA Family

- `FMA(x, y, c)`- Fused Multiply-Add → `x * y + c`
- `FMS(x, y, c)` - Fused Multiply-Subtract → `x * y - c`
- `FNMA(x, y, c)` - Fused Negative Multiply-Add → `-x*y + c`
- `FNMS(x, y, c)` - Fused Negative Multiply-Subtract → `-x * y - c`

Why FMA?

Let's see a comparison between FMA and Multiply-then-add:

- Multiply + Add:
  - First, (x * y) is computed and rounded to double precision
  - Then, (x * y) + c is computed and also rounded to double
- Fused Multiply-Add:
  - Single operation
  - Computed without intermediate rounding after multiplication

Advantages of FMA

- **Higher accuracy (How?)**

- **Higher performance**
  - Performs addition and multiplication at the cost of single multiplication
    - *AMD Bulldozer*: FP ADD/MUL/FMA latency = 5

  - Allows high-performance implementation of other floating-point algorithms eg: division, sqrt etc.

Resources on FMA

For further information see

- "Handbook of Floating Point Arithmetic" (2009), Muller, Brisebarre, de Dinechin, Jeannerod, Lefèvre

- "IA64 and Elementary Functions" (2000), Peter Markstein

- Wikipedia article on FMA: Link

- FMA Instruction set: Link