GCC Autovectorization

A journey through compiler options, SIMD extensions and C standards

Andreas Schmitz

Seminar: Automation, Compilers, and Code-Generation

06.07.2016

Motivation

What is vectorization?

- Perform one operation on multiple elements of a vector
- Chunk-wise processing instead of element wise
- Can improve computing time

Motivation

- Utilize the CPU's vectorization features
- Produce fast and small binaries

Disclaimer

Disclaimer

- The following only concentrates on C11 and GCC 5.3
- Some of the shown code snippets / directives may also apply to C++, older C standards or other compilers

Agenda

Basics

Memory Alignment I

Overview

- Data is stored in memory **aligned** or **unaligned**
  - ➔ Aligned: Address is a multiple of the alignment
- Some architectures need data to be aligned
- Intel: unaligned data access possible. But: Computation Overhead
  - ➔ Multiple reads necessary
  - ➔ Additional code to extract the data
- Data(-structures) can be aligned by adding padding

Memory Alignment II

Dealing with Alignment

- Directives to control the alignment behavior
- **GCC** specific [FSF15, 6.38]
  - `__attribute__ ((aligned (ALIGN)))`
  - `__attribute__ ((packed))`
  - Used with: `struct` and `union` or simply arrays
- **C11** Standard [ISO11, 6.2.8,7.22.3]
  - `aligned_alloc(size_t alignment, size_t size);`
  - `_Alignas(expression)` and `_Alignas(type)`

Memory Alignment III

Examples

- `struct V{short s[3];} __attribute__ ((aligned(8));`
- `char c[2] __attribute__((aligned(8)));`
- `struct A{char a; int b;} __attribute__((packed));`

Pointer Aliasing I

Overview

- 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 (next slide)

Pointer Aliasing II

```
1  void foo(int *a, int *b, int* c) {
2        *a = 42;
3        *b = 23;
4        *c = *a;
5  }
```

Figure: Pointer Aliasing, C Code

```
1  mov DWORD PTR [rdi], 42
2  mov DWORD PTR [rsi], 23
3  mov eax, DWORD PTR [rdi]
4  mov DWORD PTR [rdx], eax
```

Figure: Pointer Aliasing, Resulting Assembly Code

Pointer Aliasing III

`restrict` Keyword [ISO07, §6.7.3.1]

- C99 keyword to mark pointers as not being aliases

Pointer Aliasing IV

```
1  void foo(int * restrict a, int *
       restrict b, int* c) {
2       *a = 42;
3       *b = 23;
4       *c = *a;
5  }
```

Figure: Resolving Pointer Aliasing, C Code

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

Figure: Resolving Pointer Aliasing, Resulting Assembly

Pointer Aliasing V

Remarks

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

(Intel) SIMD Extensions I

SIMD Extension Overview

- Intel: MMX, SSE, SSE2, ... ,AVX, AVX2, AVX-512
- ARM: NEON
- Have “Bookkeeping” and Initialization overhead
- SIMD Extensions usually differ in:
  - ➔ size/number of the registers
  - ➔ operations
  - ➔ data types
  - ➔ ...

→ **Typically require:** aligned data, no pointer aliasing

(Intel) SIMD Extensions II

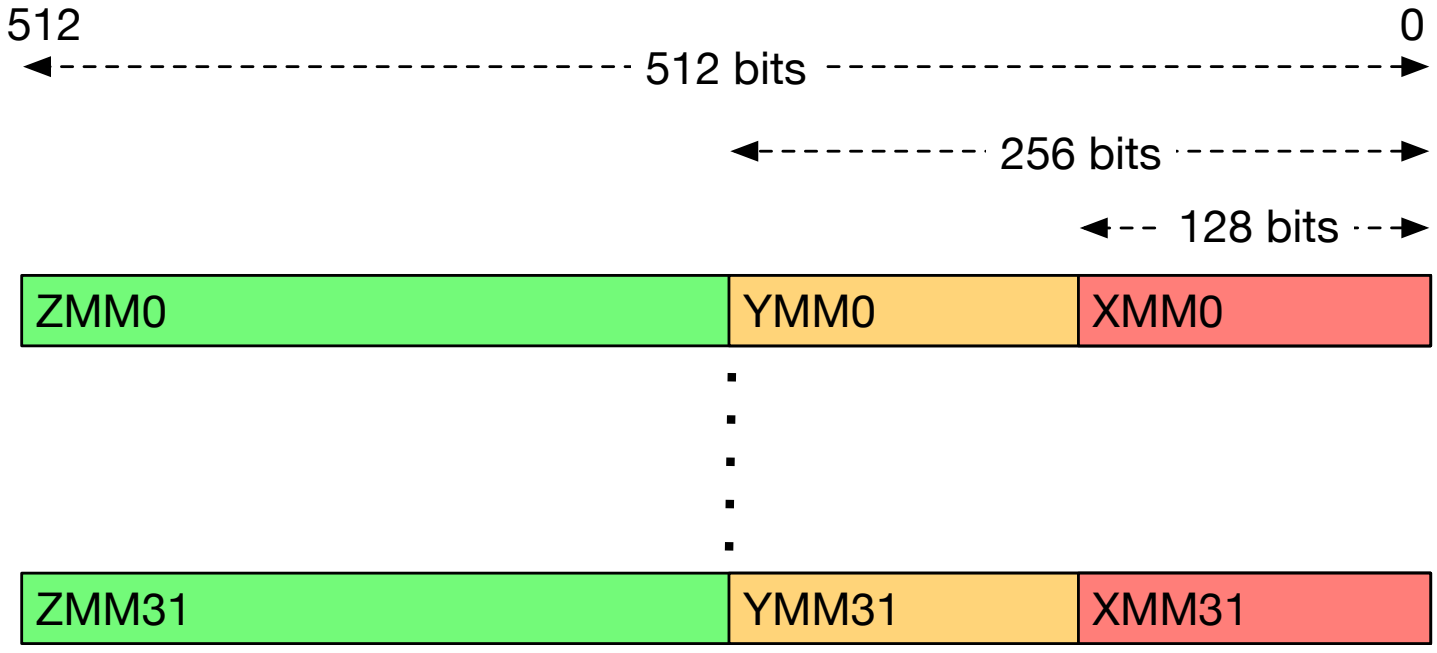

Figure: x86-64 Vector Registers

- AVX-512 (ZMM0-ZMM31)
- AVX (YMM0-YMM15)
- SSE (XMM0-XMM15)

(Intel) SIMD Extensions III

x86-64 Vector Operations - Overview [Lom11]

- Example Instructions
  - ➔ Move: `(V)MOV[A/U]P[D/S]`
  - ➔ Comparing: `(V)CMP[P/S][D/S]`
  - ➔ Arithmetic Operations: `(V)[ADD/SUB/MUL/DIV][P/S][D/S]`
- Instruction Decoding
  - ➔ `V` - AVX
  - ➔ `P,S` - packed, scalar
  - ➔ `A,U` - aligned, unaligned
  - ➔ `D,S` - double, single
  - ➔ `B, W, D, Q` - byte, word, doubleword, quadword integers
  - ➔ `[]` - required, `()` - optional
- Example: `vmovapd ymm0, YMMWORD PTR [rdi+rax]`

Empiric Analysis of GCC's autovectorization

GCC Compiler Flags

GCC Autovectorization Compiler Flags [FSF15]

- `-O -ftree-vectorize`
  - ➔ Activate autovectorization
- `-O3`
  - ➔ Optimizations including autovectorization,
- `-fopt-info-vec`, `-fopt-invo-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 Directives

GCC Vectorization `pragmas` [FSF15, 6.60.14]

- `#pragma GCC ivdep`
  - ➔ programmer asserts no loop-carried dependencies

GCC Autovectorization I

GCC Autovectorization Examples

1. Simple Loop
2. Improved Loop
3. Optimized Loop
4. C11 compatible solution
5. Non profitable loop

→ Compiled with the previously shown compiler flags

GCC Autovectorization II

Version 1: Simple Loop

```
1  #define SIZE    (1L << 16)
2  void simpleLoop(double * a, double * b)
3  {
4      for (int i = 0; i < SIZE; i++)
5      {
6          a[i] += b[i];
7      }
8  }
```

GCC Autovectorization III

GCC output: Version 1

```
simpleLoop.c:4:5: note: loop vectorized
simpleLoop.c:4:5: note: loop versioned for
      vectorization because of possible aliasing
simpleLoop.c:4:5: note: loop peeled for
      vectorization to enhance alignment
```

DEMO: Version 1

- Resulting assembly code

GCC Autovectorization IV

Version 2: Improved Loop

```
1  #define SIZE    (1L << 16)
2  void improvedLoop(double * restrict a, double *
       restrict b)
3  {
4      for (int i = 0; i < SIZE; i++)
5      {
6          a[i] += b[i];
7      }
8  }
```

GCC Autovectorization V

GCC output: Version 2

```
improvedLoop.c:4:5: note: loop vectorized
improvedLoop.c:4:5: note: loop peeled for
      vectorization to enhance alignment
```

DEMO: Version 2

- Resulting assembly code

GCC Autovectorization VI

Version 3: Optimized Loop

```
1  #define SIZE    (1L << 16)
2  #define GCC_ALN(var, alignment)
       __builtin_assume_aligned(var, alignment)
3  void optimizedLoop(double * restrict a, double *
       restrict b)
4  {
5      a = (double *) GCC_ALN(a, 32);
6      b = (double *) GCC_ALN(b, 32);
7      for (int i = 0; i < SIZE; i++)
8      {
9          a[i] += b[i];
10     }
11 }
```

GCC Autovectorization VII

Remark

- `__builtin_assume_aligned`: Caller has to assure the memory is aligned $\rightarrow$ segfault otherwise

GCC output: Version 3

```
optimizedLoop.c:7:5: note: loop vectorized
```

```
.L2:
    vmovapd ymm0, YMMWORD PTR [rdi+rax]
    vaddpd ymm0, ymm0, YMMWORD PTR [rsi+rax]
    vmovapd YMMWORD PTR [rdi+rax], ymm0
    add rax, 32
    cmp rax, 524288
    jne .L2
```

GCC Autovectorization VIII

C11 compatible solution

```
1  struct data{
2      alignas(32) double vec[SIZE];
3  };
4  void optimizedLoop(struct data * restrict a,
       struct data * restrict b)
5  {
6      for (int i = 0; i < SIZE; i++)
7          a->vec[i] += b->vec[i];
8  }
```

- GCC creates exactly the same output
- Advantage: Can be compiled with other compilers
- But: Other compilers may need additiona directives/keywords

GCC Autovectorization IX

Empiric Runtime Analysis

Loop	**Number of cycles (in ∅)** [1]
Simple Loop	106.442
Improved Loop	105.883
Optimized Loop	99.719
Optimized Loop C11	99.540
Non-vectorized Loop	444.142

Table: Average runtime of the example loops

[1] TSC using `rdtscp` instruction

Autovectorization - Not profitable loops

Non profitable loop

```
1  void nonProfitableLoop(double * a, double * b)
2  {
3      for (int i = 0; i < 8; i++)
4      {
5          a[i] += b[i];
6      }
7  }
```

GCC output with -fopt-info-vec-missed

```
nonProfitableLoop.c:3:5: note: not vectorized:
                         vectorization not profitable.
```

Autovectorization Requirements and Limitations

Autovectorizazion Requirements and Limitations

Requirements and Limitations [Cor12]

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

→ Intel Vectorization Guidelines [Sab12]

Conclusion

Conclusion I

Vector-aware coding

- Follow the Vectorization Guidelines
- Evaluate compiler reports/output
- Check the resulting assembly code
- Evaluate the performance / binary size

Conclusion II

What we haven't talked about

- Pipelining
- Cache Utilization

References

References I

[Cor12] Corden, Martyn:
Requirements for Vectorizable Loops.
(2012).
https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/

[Eva06] Evans, David:
x86 Assembly Guide.
(2006).
http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

[FSF15] Free Software Foundation, Inc.:
Using the GNU Compiler Collection (GCC).
(2015).
https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc/

References II

[ISO07] International Organization for Standardization: Programming Languages - C99.
Version: 2007.
http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf.
Geneva, CH, 2007. –
Standard

[ISO11] International Organization for Standardization: Programming Languages - C - Committee Draft.
Version: 2011.
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.
Geneva, CH, 2011. –
Standard

References III

[Lom11] Lomont, Chris:
Introduction to Intel® Advanced Vector Extensions.
(2011).
https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

[Lom12] Lomont, Chris:
Introduction to x64 Assembly.
(2012).
https://software.intel.com/en-us/articles/introduction-to-x64-assembly

References IV

[Pip12] Piper, Chuck:
An Introduction to Vectorization with the Intel® C++ Compiler.
(2012).
http://d3f8ykwhia686p.cloudfront.net/1live/intel/An_Introduction_to_Vectorization_with_Intel_Compiler_021712.pdf

[Sab12] Sabahi, Mark:
A Guide to Auto-vectorization with Intel® C++ Compilers.
(2012).
https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compile