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

- 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
  Pointer Aliasing
  (Intel) SIMD Extensions

Empiric Analysis of GCC’s autovectorization
  GCC Compiler & Compiler Flags
  Autovectorization Examples

Autovectorization Requirements and Limitations

Conclusion

References
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
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));
Overview

- Refers to memory addressed by different names
- Example
  \[ \text{char } b; \text{ char } *a = \&b; \]
- Needs to be considered by the compiler
- Can result in code overhead (next slide)
void foo(int *a, int *b, int *c) {
    *a = 42;
    *b = 23;
    *c = *a;
}  

Figure: Pointer Aliasing, C Code

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

Figure: Pointer Aliasing, Resulting Assembly Code
restrict Keyword [ISO07, §6.7.3.1]

- C99 keyword to mark pointers as not being aliases
void foo(int * restrict a, int * restrict b, int* c) {
    *a = 42;
    *b = 23;
    *c = *a;
}

Figure: Resolving Pointer Aliasing, C Code

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

Figure: Resolving Pointer Aliasing, Resulting Assembly
Remarks

- `restrict` needs to be used carefully
- Programmer is responsible for proper usage
- Mishandling can lead to wrong programs
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
AVX-512 (ZMM0-ZMM31)
AVX (YMM0-YMM15)
SSE (XMM0-XMM15)
x86-64 Vector Operations - Overview [Lom11]

- Example Instructions
  - Move: \((V)\text{MOV[A/U]}P[D/S]\)
  - Comparing: \((V)\text{CMP[P/S]}[D/S]\)
  - Arithmetic Operations: \((V)[\text{ADD/SUB/MUL/DIV}[P/S][D/S]\)

- 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: \text{vmovapd ymm0, YMMWORD PTR [rdi+rax]}
Empiric Analysis of GCC’s auto-vectorization
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 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
Version 1: Simple Loop

```c
#define SIZE (1L << 16)

void simpleLoop(double * a, double * b) {
    for (int i = 0; i < SIZE; i++)
        a[i] += b[i];
}
```
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
Version 2: Improved Loop

```c
#define SIZE  (1L << 16)
void improvedLoop(double * restrict a, double * restrict b)
{
    for (int i = 0; i < SIZE; i++)
    {
        a[i] += b[i];
    }
}
```
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
# define SIZE (1L << 16)
# define GCC_ALN(var, alignment)
    __builtin_assume_aligned(var, alignment)

void optimizedLoop(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];
    }
}
GCC Autovectorization VII

Remark

- __builtin_assume_aligned: Caller has to assure the memory is aligned → 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

```c
struct data {
    alignas(32) double vec[SIZE];
};
void optimizedLoop(struct data * restrict a,
                   struct data * restrict b)
{
    for (int i = 0; i < SIZE; i++)
        a->vec[i] += b->vec[i];
}
```

- GCC creates exactly the same output
- Advantage: Can be compiled with other compilers
- But: Other compilers may need additional directives/keywords
<table>
<thead>
<tr>
<th>Loop</th>
<th>Number of cycles (in ∅) ¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Loop</td>
<td>106.442</td>
</tr>
<tr>
<td>Improved Loop</td>
<td>105.883</td>
</tr>
<tr>
<td>Optimized Loop</td>
<td>99.719</td>
</tr>
<tr>
<td>Optimized Loop C11</td>
<td>99.540</td>
</tr>
<tr>
<td>Non-vectorized Loop</td>
<td>444.142</td>
</tr>
</tbody>
</table>

Table: Average runtime of the example loops

¹TSC using rdtscp instruction
Auto-vectorization - Not profitable loops

Non profitable loop

```c
void nonProfitableLoop(double * a, double * b)
{
    for (int i = 0; i < 8; i++)
    {
        a[i] += b[i];
    }
}
```

GCC output with -fopt-info-vec-missed

`nonProfitableLoop.c:3:5: note: not vectorized: vectorization not profitable.`
Autovectorization Requirements and Limitations
Autovectorization 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 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.  
http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

[FSF15] FREE SOFTWARE FOUNDATION, Inc.:  
Using the GNU Compiler Collection (GCC).  
https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc/
References II

Geneva, CH, 2007. – Standard

Geneva, CH, 2011. – Standard
[Lom11] LOMONT, Chris:

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