Code Generation for Data Processing
Lecture 6: Vectorization

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2023/24
Parallel Data Processing

- Sequential execution has inherently limited performance
  - Clock rate, data path lengths, speed of light, ...
- Parallelism is the key to substantial and scalable perf. improvements
- Modern systems have many levels of parallelism:
  - Multiple nodes/systems, connected via network
  - Different compute units (CPU, GPU, etc.), connected via PCIe
  - Multiple CPU sockets, connected via QPI (Intel) or HyperTransport (AMD)
  - Multiple CPU cores
  - Multiple threads per core
  - Instruction-level parallelism (superscalar out-of-order execution)
  - Data parallelism (SIMD)
Single Instruction, Multiple Data (SIMD)

- Idea: perform same operations on multiple data in parallel
- First computer with SIMD operations: MIT Lincoln Labs TX-2, 1957
- Wider use in HPC in 1970s with vector processors (Cray et al.)
  - Ultimately replaced by much more scalable distributed machines
- SIMD-extensions for multimedia processing from 1990s onwards
  - Often include very special instructions for image/video/audio processing
- Shift towards HPC and data processing around 2010
- Extensions for machine learning/AI in late 2010s

---

SIMD: Idea

- Multiple data elements are stored in *vectors*
  - Size of data may differ, vector size is typically constant
  - Single elements in vector referred to as *lane*
- (Vertical) Operations apply the same operation to all lanes

<table>
<thead>
<tr>
<th></th>
<th>lane 3</th>
<th>lane 2</th>
<th>lane 1</th>
<th>lane 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>src 1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>src 2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>result</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

- Horizontal operations work on neighbored elements
SIMD ISAs: Design

- Vectors are often implemented as fixed-size wide registers
  - Examples: ARM NEON 32×128-bit, Power QPX 32×256-bit
  - Data types and element count is defined by instruction
- Some ISAs have dynamic vector sizes: ARM VFP, ARM SVE, RISC-V V
  - Problematic for compilers: variable spill size, less constant folding
- Data types vary, e.g. i8/i16/i32/i64/f16/bf16/f32/f64/f128
  - Sometimes only conversion, sometime with saturating arithmetic
- Masking allows to suppress operations for certain lanes
  - Dedicated mask registers (AVX-512, SVE, RVV) allow for hardware masking
  - Can also apply for memory operations, optionally suppressing faults
  - Otherwise: software masking with another vector register
SIMD: Use Cases

- **Dense linear algebra**: vector/matrix operations
  - Implementations: Intel MKL, OpenBLAS, ATLAS, ...

- **Sparse linear algebra**
  - Needs gather/scatter instructions

- **Image and video processing**, **manipulation**, **encoding**

- **String operations**
  - Implemented, e.g., in glibc, simdjson

- **Cryptography**
SIMD ISAs: Usage Considerations

- Very easy to implement in hardware
  - Simple replication of functional units and larger vector registers
  - Too large vectors, however, also cause problems (AVX-512)

- Offer significant speedups for certain applications
  - With 4x parallelism, speed-ups of \(~3x\) are achievable

- Caveat: non-trivial to program
  - Optimized routines provided by libraries
  - Compilers try to auto-vectorize, but often need guidance
SIMD Programming: (Inline) Assembly

- Idea: SIMD is too complicated, let programmer handle this
- Programmer specifies exact code (instrs, control flow, and registers)
- Inline assembly allows for integration into existing code
  - Specification of register constraints and clobbers needed

- “Popular” for optimized libraries
  - Allows for best performance
    - Very tedious to write, manual register allocation, non-portable
    - No optimization across boundaries
SIMD Programming: Intrinsics

- Idea: deriving a SIMD schema is complicated, delegate to programmer
- Intrinsic functions correspond to hardware instructions
  - __m128i _mm_add_epi32 (__m128i a, __m128i b)
- Programmer explicitly specifies vector data processing instructions
  compiler supplements registers, control flow, and scalar processing

+ Allows for very good performance, still exposes all operations
+ Compiler can to some degree optimize intrinsics
  - GCC does not; Clang/LLVM does – intrinsics often lowered to LLVM-IR vectors
  - Tedious to write, non-portable
float sdot(size_t n, const float x[n], const float y[n]) {
    size_t i = 0;
    __m128 sum = _mm_set_ps1(0);
    for (i = 0; i < (n & ~3ul); i += 4) {
        __m128 xl = _mm_loadu_ps(&x[i]);
        __m128 yl = _mm_loadu_ps(&y[i]);
        sum = _mm_add_ps(sum, _mm_mul_ps(xl, yl));
    }
    // ... take care of tail (i..<n) ...
}

Intrinsics for Unknown Vector Size

- Size not known at compile-time, but can be queried at runtime
  - SVE: instruction `incd` adds number of vector lanes to register
- In C: behave like an incomplete type, except for parameters/returns
- Flexible code often slower than with assumed constant vector size

Consequences:
- Cannot put such types in structures, arrays, `sizeof`
- Stack spilling implies variably-sized stack

Instructions to set mask depending on bounds: `whilelt`, ...
- No loop peeling for tail required
SIMD Programming: Target-independent Vector Extensions

- Idea: vectorization still complicated, but compiler can choose instrs.
  - Programmer still specifies exact operations, but in target-independent way
  - Often mixable with target-specific intrinsics
- Compiler maps operations to actual target instructions
- If no matching target instruction exists, use replacement code
  - Inherent danger: might be less efficient than scalar code
- Often relies on explicit vector size
GCC Vector Extensions

```c
#include <stdint.h>

typedef uint32_t uint32x4_t
   __attribute__((vector_size(16)));

uint32x4_t
addvec(uint32x4_t a, uint32x4_t b) {
   return a + b;
}

uint32x4_t
modvec(uint32x4_t a, uint32x4_t b) {
   return a % b;
}
```

```asm
addvec:
paddd xmm0, xmm1
   ret

modvec:
movd ecx, xmm1
movd eax, xmm0
xor edx, edx
pextrd edi, xmm1, 1
div ecx
pextrd eax, xmm0, 1
pextrd ecx, xmm1, 2
mov esi, edx
xor edx, edx
div edi
pextrd eax, xmm0, 2
mov r8d, edx
xor edx, edx
div ecx
pextrd ecx, xmm1, 3
pextrd eax, xmm0, 3
movd xmm0, esi
pinsrd xmm0, r8d, 1
mov edi, edx
xor edx, edx
div ecx
movd xmm1, edi
pinsrd xmm1, edx, 1
punpcklqdq xmm0, xmm1
ret
```
LLVM-IR: Vectors

- \(<N \times ty>\) – fixed-size vector type, e.g. \(<4 \times i32>\)
  - Valid element type: integer, floating-point, pointers
  - Memory layout: densely packed (i.e., \(<8 \times i2> \approx i16>\)
- \(<\text{vscale} \times N \times ty>\) – scalable vector, e.g. \(<\text{vscale} \times 4 \times i32>\)
  - Vector with a multiple of \(N\) elements
  - Intrinsic @llvm.vscale.i32() – get runtime value of \text{vscale}

- Most arithmetic operations can also operate on vectors
- \text{insertelement}/\text{extractelement}: modify single element
  - Example: \(\%4 = \text{insertelement} <4 \times \text{float}> \%0, \text{float} \%1, \text{i32} \%2\)
  - Index can be non-constant value
LLVM-IR: shufflevector

- Instruction to reorder values and resize vectors
- `shufflevector <n x ty> %x, <n x ty> %y, <m x i32> %mask`
  - `%x, %y` – values to shuffle, must have same size
  - `%mask` – element indices for result (0..<n refer to %x, n..<2n to %y)
  - Result is of type `<m x ty>`

```
shufflevector <4 x i32> %x, <4 x i32> %y, <3 x i32> <i32 1, i32 7, i32 7>
```

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>4</td>
<td>8</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>67</td>
<td>10</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

Diagram:

```
    23  4  8  25
    ▼   ▼   ▼
     4  53  53
     ▼   ▼   ▼
      1    7    7
```
LLVM-IR: Lowering Intrinsics

- Intrinsics translated to native LLVM-IR if possible
  - Allows optimizations
  - Intent of programmer might get lost

```c
#include <immintrin.h>
__m128 func(__m128 a, __m128 b) {
  __m128 rev = _mm_shuffle_epi32(a + b, 0x1b);
  return _mm_round_ps(rev, _MM_FROUND_TO_NEG_INF);
}
```

```c
define <4 x float> @func(<4 x float> %0, <4 x float> %1) {
  %3 = fadd <4 x float> %0, %1
  %4 = shufflevector <4 x float> %3, <4 x float> poison, <4 x i32> <i32 3, i32 2, i32 1, i32 0>
  %5 = tail call <4 x float> @llvm.x86.sse41.round.ps(<4 x float> %4, i32 1)
  ret <4 x float> %5
}
declare <4 x float> @llvm.x86.sse41.round.ps(<4 x float>, i32 immarg)
```
SIMD Programming: Single Program, Multiple Data (SPMD)

- So far: manual vectorization
- Observation: same code is executed on multiple elements
- Idea: tell compiler to vectorize handling of single element
  - Splice code for element into separate function
  - Tell compiler to generate vectorized version of this function
  - Function called in vector-parallel loop

- Needs annotation of variables
  - Varying: variables that differ between lanes
  - Uniform: variables that are guaranteed to be the same
    (basically: scalar values that are broadcasted if necessary)
#pragma omp declare simd
int foo(int x, int y) {
    return x + y;
}

Compiler generates version that operates on vector

foo:
    add edi, esi
    mov eax, edi
    ret

_ZGVxN4vv_foo:
    paddd xmm0, xmm1
    ret
#pragma omp declare simd uniform(y)
int foo(int x, int y) {
    return x + y;
}

- Uniform: always same value

foo:
    add edi, esi
    mov eax, edi
    ret

_ZGVxN4vu_foo:
    movd xmm1, eax
    pshufd xmm2, xmm1, 0
    paddd xmm0, xmm2
    ret
SPMD: Example (OpenMP) – if/else

```c
#pragma omp declare simd
int foo(int x, int y) {
    int res;
    if (x > y) res = x;
    else res = y - x;
    return res;
}
```

▶ Diverging control flow: all paths are executed

```asm
foo:
    mov eax, esi
    sub eax, edi
    cmp edi, esi
    cmovg eax, edi
    ret

_ZGVxN4vv_foo:
    movdqa xmm2, xmm0
    pcmpgtd xmm0, xmm1
    psubd xmm1, xmm2
    pblendvb xmm1, xmm2, xmm0
    movdqa xmm0, xmm1
    ret
```
SPMD to SIMD: Handling if/else

- Control flow solely depending on uniforms: nothing different
- Otherwise: control flow may diverge
  - Different lanes may choose different execution paths
  - But: CPU has only one control flow, so all paths must execute
- Condition becomes mask, mask determines result
- After insertion of masks, linearize control flow
  - Relevant control flow now encoded in data through masks
- Problem: side-effects prevent vectorization
SPMD to SIMD: Handling Loops

- Uniform loops: nothing different
- Otherwise: need to retain loop structure
  - “active” mask added to all loop iterations
  - Loop only terminates once all lanes terminate (active is zero)
  - Lanes that terminated early need their values retained
- Approach also works for nested loops/conditions
- Irreducible loops need special handling\(^\text{21}\)

SPMD Implementations on CPUs

- OpenMP SIMD functions
  - Need to be combined with `#pragma omp simd` loops

- Intel ispc\(^{22}\) (Implicit SPMD Program Compiler)
  - Extension of C with keywords `uniform`, `varying`
  - Still active and interesting history\(^{23}\)

- OpenCL on CPU
  - Very similar programming model
  - But: higher complexity for communicating with rest of application

---


\(^{23}\) https://pharr.org/matt/blog/2018/04/30/ispc-all
Semi-explicit vectorization
Programmer chooses level of vectorization
  E.g., inner vs. outer loop
Compiler does actual work

+ Allows simple formulation of complex control flow
  Compilers often fail at handling complex control flow well
    Loops are particularly problematic
SIMD Programming: Auto-vectorization

- Idea: programmer is too incompetent/busy, let compiler do vectorization

- Inherently difficult and problematic, after decades of research
  - Recognizing and matching lots of patterns
  - Instruction selection becomes more difficult
  - Compiler lacks domain knowledge about permissible transformations

- Executive summary of the state of the art:
  - Auto-vectorization works well for very simple cases
  - For “medium complexity”, code is often suboptimal
  - In many cases, auto-vectorization fails on unmodified code
Auto-vectorization Strategies

- Loop Vectorization
  - Try to transform loop body into vectors with $n$ lanes
  - Often needs tail loop for remainder that doesn’t fill a vector
  - Extremely common

- Superword-level Parallelism (SLP)
  - Vectorize constructs outside of loops
  - Detect neighbored stores, try to fold operations into vectors
Loop Vectorization: Strategy

► Only consider innermost loop (at first)

1. Check legality: is vectorization possible at all?
   ► Only vectorizable data types and operations used
   ► No loop-carried dependencies, overlapping memory regions, etc.

2. Check profitability: is vectorization beneficial?
   ► Consider: runtime checks, gather/scatter, masked operations, etc.
   ► Needs information about target architecture

3. Perform transformation
Outer Loop Vectorization

- Vectorizing the innermost loop not always beneficial
  - Example 1: inner loop has only few iterations
  - Example 2: inner loop has loop-carried dependencies
- Thus: need to consider outer loops as well
  - Also: vectorization on multiple levels might be beneficial
- Very limited support in compilers, if any
Auto-vectorization is Hard

- Biggest problem: data dependencies
  - Resolving loop-carried dependencies is difficult
- Memory aliasing
  - Overlapping arrays, or – worse – loop counter
- Which loop level to vectorize? Multiple?
- Loop body *might* impact loop count
- Function calls, e.g. for math functions
- Strided memory access (e.g., only every n-th element)
- Choosing vectorization level (outer loop *might* be better)

- Is vectorization profitable *at all*?
- Often black box to programmer, preventing fine-grained tuning
Vectorization – Summary

- SIMD is an easy way to improve performance numbers of CPUs
- Most general-purpose ISAs have one or more SIMD extensions
- Recent trend: variably-length vectors
- Inline Assembly: easiest for compiler, but extremely tedious
- Intrinsics: best trade-off towards performance and usability
- Target-independent operations: slightly increase portability
- SPMD: strategy dominant for GPU programming
- Auto-vectorization: very hard, unsuited for complex code
Vectorization – Questions

- Why do modern CPUs provide SIMD extensions?
- Why come variable-length SIMD extensions with higher runtime costs?
- How are SIMD intrinsics lowered to LLVM-IR?
- What is the downside of target-independent vector operations?
- How can if/else/for constructs be vectorized?
- What is the difference between a uniform and a varying variable?
- Why is auto-vectorization often sub-par to manual optimization?