Code Generation for Data Processing

Lecture 13: Vectorization

Alexis Engelke

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

Winter 2022/23
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\(^5^6\)

- 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>src 1</th>
<th>lane 3</th>
<th>lane 2</th>
<th>lane 1</th>
<th>lane 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></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
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
Fault Suppression

- Variable-length vectors are problematic for buffers of unknown size
  - Example: NUL-terminated C strings

- Classical approach: ensure alignment to prevent page faults
  - These types of out-of-bounds reads are guaranteed to be non-harmful
  - Downside: needs loop peeling code for start/end

- More recent approach: make hardware suppress exceptions
  - Option 1: specify that masked out lanes do not produce faults
  - Option 2: stop loading after first fault, store as mask in register
  - Downside 1: increased complexity in hardware, may use microcode
  - Downside 2: permits speculative vectorization at cost of more instructions
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

#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;
}
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 add(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
SPMD: Example (OpenMP)

```c
#pragma omp declare simd uniform(y)
int add(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

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

_ZGVxN4vv_foo:
    movdqa xmm2, xmm0
    pcmplgt xmm0, xmm1
    psubd xmm1, xmm2
    pblendvb xmm1, xmm2, xmm0
    movdqa xmm0, xmm1
    ret
```
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
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\(^\text{57}\)

SPMD Implementations on CPUs

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

- Intel ispc\(^{58}\) (Implicit SPMD Program Compiler)
  - Extension of C with keywords `uniform`, `varying`
  - Still active and interesting history\(^{59}\)

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

---


\(^{59}\) [https://pharr.org/matt/blog/2018/04/30/ispc-all](https://pharr.org/matt/blog/2018/04/30/ispc-all)
SIMD Programming: SPMD on CPUs

▸ 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 is Hard

- Biggest problem: data dependencies
  - Resolving loop-carried dependencies is difficult
- Memory aliasing
  - Overlapping arrays, or – worse – loop counter
- 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
Auto-vectorization Strategies

- **Inner Loop Vectorization**: unroll innermost loop \( n \) times
  - Try to compact loop body into vectors with \( n \) lanes

- **Outer Loop Vectorization**: unroll outer loop \( n \) times
  - Try to compact loop body into vectors with \( n \) lanes
  - Generally does not support diverging control flow in loop body

- **Superword-level Parallelism (SLP)**: packing series of scalar stores
  - Detect neighbored stores, try to fold operations into vectors
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