

Data-Parallel Execution using SIMD Instructions

Single Instruction Multiple Data

- data parallelism exposed by the instruction set
- CPU register holds multiple fixed-size values (e.g., 4 times 32-bit)
- instructions (e.g., addition) are performed on two such registers (e.g., executing 4 additions in one instruction)

4	2	1	7
---	---	---	---

+

3	1	6	2
---	---	---	---

=

7	3	7	9
---	---	---	---

AVX-512

- available on Skylake server CPUs
- 32 512-bit registers: ZMM0 to ZMM31
- can be interpreted as
 - ▶ 64 8-bit integers
 - ▶ 32 16-bit integers
 - ▶ 16 32-bit integers
 - ▶ 8 64-bit integers
 - ▶ 16 32-bit floats
 - ▶ 8 64-bit floats
- extensive and fairly orthogonal set of operations
- Skylake server CPUs have 2 AVX-512 processing units and can therefore process 128 bytes per cycle
- important subsets: AVX-512F (foundation), AVX-512BW (byte, word), AVX-512DQ (doubleword and quadword instructions), AVX-512CD (lzcnt and conflict detection)

Auto-Vectorization

- compilers can sometimes transform serial code into SIMD code
- this does not work for complicated code and is quite unpredictable (one compiler may work, another may not)

Intrinsics

- intrinsics provide an interface to SIMD instructions without writing assembly
- ZMM registers are represented as special data types:
 - ▶ `__m512i` (all integer types, width is specified by operations)
 - ▶ `__m512` (32-bit floats)
 - ▶ `__m512d` (64-bit floats)
- operations look like C functions, e.g., add 16 32-bit integers:
`__m512i _mm512_add_epi32(__m512i a, __m512i b)`
- compiler performs register allocation

Getting Data To/From Registers

- aligned load (memory location has to be 64-byte aligned):
`__m512i _mm512_load_si512 (void const* mem_addr)`
- unaligned load (slightly slower):
`__m512i _mm512_loadu_si512 (void const* mem_addr)`
- broadcast a single value (available for different widths):
`__m512i _mm512_set1_epi32 (int a)`
- there is no instruction for loading a 64-byte constant into a register (must happen through memory); however, there is a convenient (but slow) intrinsic for that:
`__m512i _mm512_set_epi32(int e15, ..., int e0)`
(arguments can also be specified in reverse: `*_setr_*`)
- store:
`void _mm512_store_epi32 (void* mem_addr, __m512i a)`

Arithmetic Operations

- addition/subtraction: add, sub
- multiplication (truncated): mullo (16, 32, or 64 bit input, output size same as input)
- saturated addition/subtraction: adds, subs (stays at extremum instead of wrapping, only 8 and 16 bits)
- absolute value: abs
- extrema: min/max
- multiplication (full precision): mul (only 32 bit input, produces 64 bit output)
- some of these are also available as unsigned variants (epu suffix)
- no integer division/modulo¹
- no overflow detection

¹division by power of 2 can be emulated using shift, division by constant can sometimes be implemented using multiplication/shifting/addition

Intrinsics Example

```

alignas(64) int in[1024];
void simpleMultiplication() {
    __m512i three = _mm512_set1_epi32(3);
    for (int i=0; i<1024; i+=16) {
        __m512i x = _mm512_load_si512(in + i);
        __m512i y = _mm512_mullo_epi32(x, three);
        _mm512_store_epi32(in + i, y); }}

```

```

xor     eax, eax
vmovups zmm0, ZMMWORD PTR CONST.0[rip]
.LOOP:
vpmulld zmm1, zmm0, ZMMWORD PTR [in+rax*4]
vmovups ZMMWORD PTR [in+rax*4], zmm1
add     rax, 16
cmp     rax, 1024
jl     .LOOP
ret
.CONST.0: .long 0x00000003,0x00000003,...

```


Logical and Bitwise Operations

- logical: and, andnot, or, xor
- rotate left (right) by same value: rol (ror)
- rotate left (right) by different values: rolv (rorv)
- shift² left (right) by same value: slli (srli)
- shift left (right) by different values: sllv (srlv)
- convert different sizes (zero/sign-extend, truncate): cvt
- count leading zeros: lzcnt

²8-bit shifts are missing

Comparisons

- compare 32-bit integers:
`__mmask16 _mm512_cmpOP_epi32_mask (__m512i a, __m512i b)`
- OP is one of (eq, ge, gt, le, lt, neq)
- comparisons may also take a mask as input, which is equivalent to performing AND on the masks
- assumes signed integers³
- result is a bitmap stored in a special “opmask” register (K1-K7) and is available as special data type (`__mmask8` to `__mmask64`)

³to compare unsigned integers, flip the most significant bit of inputs (using xor)

Operations on Masks

- operations on masks: `kand`, `knand`, `knot`, `kor`, `kxnor`, `kxor`
- `__mmask16 _kand (__mmask16 a, __mmask16 b)`
- masks are automatically converted to integers
- to count number of bit set to 1: `__builtin_popcount(mask)`

Zero Masking

- selectively ignore some of the SIMD lanes (using a bitmap)
- almost all operations support masking
- add elements, but set those not selected by mask to zero:

```
__m512i _mm512_maskz_add_epi32 (__mmask16 k, __m512i a, __m512i b)
```

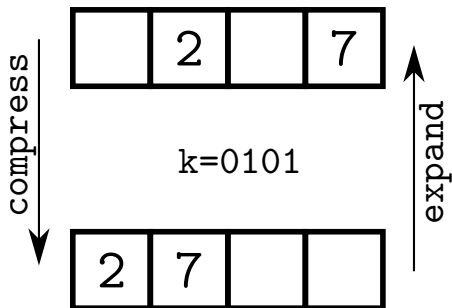
$$\begin{array}{r} \text{a} \quad \boxed{4} \quad \boxed{2} \quad \boxed{1} \quad \boxed{7} \\ \quad \quad + \quad k=0101 \\ \text{b} \quad \boxed{3} \quad \boxed{1} \quad \boxed{6} \quad \boxed{2} \\ \quad \quad = \\ \quad \quad \boxed{0} \quad \boxed{3} \quad \boxed{0} \quad \boxed{9} \end{array}$$

Masking Example

```
maskedArithmetic():
    xor     eax, eax
    vmovups zmm2, ZMMWORD PTR .CONST.0[rip]
    vmovups zmm1, ZMMWORD PTR .CONST.1[rip]
    vmovups zmm0, ZMMWORD PTR .CONST.2[rip]
.LOOP:   vmovups zmm3, ZMMWORD PTR [array+rax*4]
    vpcmpgtd k1, zmm2, zmm3
    vmovdqa32 zmm4{k1}{z}, zmm0
    vpmulld  zmm5, zmm3, zmm1
    vpaddd   zmm5{k1}, zmm3, zmm4
    vmovdqu32 ZMMWORD PTR [array+rax*4], zmm5
    add     rax, 16
    cmp     rax, 1024
    jb     .LOOP
    ret
.CONST.0: .long 0x00000009,0x00000009,...
.CONST.1: .long 0x00000007,0x00000007,...
.CONST.2: .long 0x00000003,0x00000003,...
```

Compress and Expand

- compress: `__m512i _mm512_maskz_compress_epi32(__mmask16 k, __m512i a)`
- expand: `__m512i _mm512_maskz_expand_epi32 (__mmask16 k, __m512i a)`
- also to memory: `compressstoreu`
- and from memory: `expandloadu`



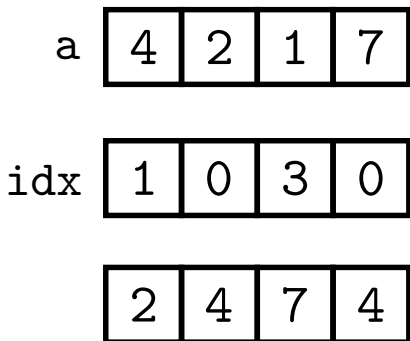
Selection Example

```
uint32_t scalar(int32_t* in, int32_t inCount, int32_t x, int32_t* out) {
    uint32_t outPos = 0;
    for (int32_t i=0; i<inCount; i++)
        if (in[i] < x)
            out[outPos++] = i;
    return outPos; }
```

```
uint32_t SIMD(int32_t* in, int32_t inCount, int32_t x, int32_t* out) {
    uint32_t outPos = 0;
    __m512i cmp = _mm512_set1_epi32(x); __m512i sixteen = _mm512_set1_epi32(16);
    __m512i indexes = _mm512_setr_epi32(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
    for (int32_t i=0; i<inCount; i+=16) {
        __m512i inV = _mm512_loadu_si512(in + i);
        __mmask16 mask = _mm512_cmplt_epi32_mask(inV, cmp);
        _mm512_mask_compressstoreu_epi32(out + outPos, mask, indexes);
        uint32_t count = __builtin_popcount(mask);
        indexes = _mm512_add_epi32(indexes, sixteen);
        outPos += count;
    }
    return outPos; }
```


Permute

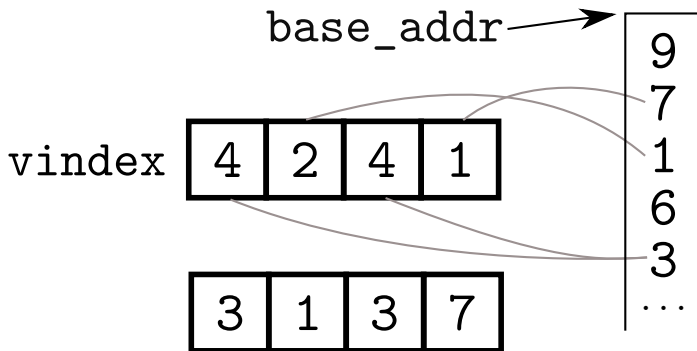
- `permute`⁴ (also called shuffle) a using the corresponding index in `idx`:
`__m512i _mm512_permutexvar_epi32 (__m512i idx, __m512i a)`
- a bit of a misnomer, is not just shuffle or permute but can replicate elements
- very powerful, can, e.g., be used to implement small, in-register lookup tables



⁴not available for 8-bit values

Gather (1)

- perform multiple loads and store results in register



Gather (2)

- load 16 32-bit integers using 32-bit indices:

```
__m512i _mm512_i32gather_epi32 (__m512i vindex, void const* base_addr,  
int scale)
```

- load 8 64-bit integers using 64-bit indices:

```
__m512i _mm512_i64gather_epi64 (__m512i vindex, void const* base_addr,  
int scale)
```

- load 16 8-bit or 16-bit values (zero or sign extended):

```
__m512i _mm512_i32extgather_epi32 (__m512i index, void const* mv,  
_MM_UPCONV_EPI32_ENUM conv, int scale, int hint)
```

- indices are multiplied by scale, which must be 1, 2, 4 or 8
- gathering 8 elements performs 8 loads (using the 2 load units)
- is not necessarily faster than individual loads (unless one needs the result in SIMD register anyway)

Scatter

- store 16 32-bit integers using 32-bit indices:

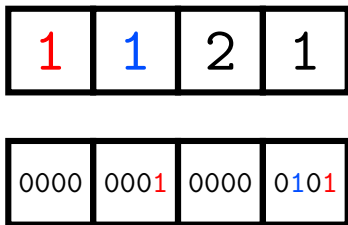
```
void _mm512_i32scatter_epi32 (void* base_addr, __m512i vindex, __m512i  
a, int scale)
```

- store 8 64-bit integers using 64-bit indices:

```
void _mm512_i64scatter_epi64 (void* base_addr, __m512i vindex, __m512i  
a, int scale)
```

Conflict Detection

- problem: during a scatter, when multiple indices have the same value, bad things can happen
- test each element for equality with all other elements⁵:
`__m512i _mm512_conflict_epi32 (__m512i a)`



⁵only available for 32-bit and 64-bit values

Conflict Detection With Masking

- conflict detection instructions also support masking
- however, an element selected by the mask is still compared with *all* other elements (even ones that are not selected by the mask)
- possible solution: set elements one wants to ignore to a value that does not occur in the vector, mask

Cross-Lane Operations, Reductions

- often one would like to perform operations *across* lanes
- except for some exceptions (e.g., permute), this cannot be done efficiently
- horizontally vectorized operations and violate the SIMD paradigm (i.e., are slow)
- for example, there is an intrinsic for adding register values:

```
int _mm512_reduce_add_epi32 (__m512i a)
```
- but (if supported by the compiler), it is translated to multiple SIMD instructions (by the compiler) and is therefore less efficient than `_mm512_add_epi32`

Other Useful Instructions

- most SSE/AVX256 instructions are strictly worse than their AVX-512 equivalents
- SSE4.2 is an exception, provides CRC and parallel string comparisons
- allows comparing two 16-byte chunks, e.g., to find out whether (and where) a string contains a set of characters
- very flexible: len/zero-terminated strings, any/substring search, bitmask/index result, etc.
- tool for finding the correct settings: <http://halobates.de/pcmpstr-js/pcmp.html>
- BMI instruction set is another exception
 - ▶ extract bits indicated by mask from 64-bit value a:
`uint64_t _pext_u64 (uint64_t a, uint64_t mask)`
 - ▶ unsigned `__int64` `_pdep_u64 (unsigned __int64 a, unsigned __int64 mask)`
 - ▶ like compress/expand, but at bit level

Other Optimization Issues

- C++ aliasing semantics (`__restrict__`)
- vectorization vs. guaranteed floating point semantics: (`-ffast-math`)

Further Reading

- *Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation*, Lang and Mühlbauer and Funke and Boncz and Neumann and Kemper, SIGMOD 2016
- *Rethinking SIMD Vectorization for In-Memory Databases*, Polychroniou and Raghavan and Ross, SIGMOD 2015
- *Efficient Lightweight Compression Alongside Fast Scans*, Polychroniou and Ross, DaMoN 2015
- *SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units* Willhalm and Popovici and Boshmaf and Plattner and Zeier and Schaffner, PVLDB 2(1) 2009
- *Decoding billions of integers per second through vectorization*, Lemire and Boytsov, Software-Practice and Experience 45