Data Processing on Modern Hardware

Jana Giceva

Lecture 6: Data-Level Parallelism (DLP)
We distinguish two kinds of parallelism in applications:

- **Data-Level parallelism (DLP)**
  - many data items are processed at the same time
- **Task-Level parallelism (TLP)**
  - different tasks operate independently and in parallel

Computer hardware can exploit them in four major ways:

- **Instruction-level parallelism** (ILP) – exploits DLP using pipelining and speculative execution
- **Vector architectures**, SIMD, GPUs – exploit DLP by applying the same instructions to a collection of data
- **Thread-level parallelism** – exploit DLP and/or TLP using hardware threads that work in parallel
- **Request-level parallelism** – exploits DLP and TLP using mainly de-coupled tasks specified by the programmer (usually at large scale, think data-center)
Flynn’s taxonomy

- **SISD – Single instruction stream, single data stream**
  - Standard sequential computer, that can exploit ILP (last week’s lecture)
  - Example: uni-processor

- **SIMD – Single instruction stream, multiple data streams**
  - The same instruction is executed by multiple processors or special instruction sets using potentially multiple data streams
  - Example: vector, SIMD extensions to standard ISAs, GPUs

- **MISD – Multiple instruction streams, single data stream**

- **MIMD – Multiple instruction streams, multiple data streams**
  - Each processor fetches its own instructions and operates on its own data, targets TLP
  - Example: multi-socket, multicore machines (thread-level parallelism) or rack- / warehouse-scale computers (request-level parallelism)
Most modern processors include a **SIMD** unit.

- Single Instruction stream, Multiple Data streams
  - Execute the same assembly instructions on a set of values.
  - Also called **vector unit**; **vector processors** are entire systems built on that idea.
Vector processor

- Vector registers

- Vector functional units

- Vector load / store units

- Set of scalar registers
  - Provide data as input to vector functional units
  - Compute addresses to pass to the vector load/store units

- Can program the vector processor with vector instructions – apply the same operation on vectors of data

- When the compiler produces vector instructions, and the resulting code spends much of its time running in vector mode, the code is said to be vectorized
  - Loops can be vectorized when they do not have dependencies between the iterations
SIMD ISA extensions for multimedia

- **MultiMedia Extensions** (MMX) (1996) repurposed existing 64-bit floating point registers
- **Streaming SIMD Extensions** (SSE) (1999) introduced 16 x 128-bit wide registers (XMM)
- **Advanced Vector Extensions** (AVX) (2010) new 16 x 256-bit wide registers (YMM)
  - vaddpd, vsubpd, vmladpd, vdivpd, vfmaddpd, vfmsubpd, vmpxx, vmovapd, vbroadcastsd
- **AVX2** (2013) added 30 new instructions
  - e.g., gather (vgather) and vector shifts (vpsll, vpsrl, vpsra)
- **AVX-512** (2017) doubled the width to 512 bits (32 x ZMM registers) and added 250 new instructions
  - e.g., scatter (vpsscatter) and mask registers (opmask)

The goal of these extensions are to **accelerate** carefully written **libraries** rather than for the compiler to generate them

Recent **x86 compilers** try to **generate** such **code** for **floating point intensive applications**

Since the opcode determines the width of the SIMD register, every time the width doubles so must the number of SIMD instructions
The processing model is typically based on **SIMD registers** or **vectors**.

**Typical values (e.g., on Intel Skylake):**

- $32 \times 512$ bit-wide registers (zmm0 through zmm31)
- Usable as $64 \times 8$-bit integers, $32 \times 16$-bit integers, $16 \times 32$-bit integers, $8 \times 64$-bit integers, $16 \times 32$-bit floats, or $8 \times 64$-bit floats
SIMD Programming Model

- SIMD instructions make **independence** explicit
  - No data hazards within a vector instruction
  - Check for data hazards only between vectors.
  - **Data parallelism**

- Data **may** need to be **aligned** in memory to the **width** of the SIMD unit to prevent the compiler generating scalar instructions for otherwise vectorizable code.

- Parallel execution promises n-fold performance advantage
  - Not quite achievable in practice, however.

- Vector code sometimes uses more instructions on trivial things:
  - Converting and moving data to the right position in the register
  - Emulating branches with conditional moves
Coding for SIMD

How can I make use of SIMD instructions as a programmer?

- **Auto-vectorization**
  - Some compilers automatically detect opportunities to use SIMD
  - Approach is rather limited, do not rely on it (check with objdump or godbolt.org)
  - Advantage: platform independent

- **Compiler attributes**
  - Use `__attribute__((vector_size (...)))` annotations to state your intentions
  - Advantage: platform independent
  - Compiler will generate non-SIMD code if the platform does not support it
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char **argv){
    int a[256], b[256], c[256];
    for(unsigned int i = 0; i < 256; i++){
        a[i] = i + 1;
        b[i] = 100 * (i + 1);
    }
    for(unsigned int i = 0; i < 256; i++){
        c[i] = a[i] + b[i];
    }
    printf("c = [ %i, %i, %i, %i ]\n", c[0], c[1], c[2], c[3]);
    return EXIT_SUCCESS;
}

- Using x86-64 gcc 10.1 (flag –O3)
  - movdqa – move aligned packed integer value
  - padd – parallel add packed integers (d stands for 32-bit values)
  - movaps – move aligned packed single precision FP value

Register | Use
----------|----------
%rax      | i        
%xmm5     | b[i]     
%xmm0     | a[i], c[i]

Increment i by SIMD length of 16 bytes, and check the loop condition.
Using x86-64 gcc 10.1 (flag –O0)

- Data transfers scalar ↔ SIMD is done through memory

```c
#include <stdlib.h>
#include <stdio.h>

typedef int v4si __attribute__((vector_size (16)));
union int_vec { int val[4]; v4si vec;};

typedef union int_vec int_vec;
int main (int argc, char **argv){
    int_vec a, b, c;
    a.val[0] = 1;  a.val[1] = 2;
    b.val[0] = 100; b.val[1] = 200;
    c.vec = a.vec + b.vec;
    printf("c = [ %i, %i, %i, %i ]\n",
        c.val[0], c.val[1], c.val[2], c.val[3]);
    return EXIT_SUCCESS;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%xmm0</td>
<td>b, c</td>
</tr>
<tr>
<td>%xmm1</td>
<td>a</td>
</tr>
</tbody>
</table>

- `movl $1, -16(%rbp)`
- `movl $2, -12(%rbp)`
- `movl $3, -8(%rbp)`
- `movl $4, -4(%rbp)`
- `movl $100, -32(%rbp)`
- `movl $200, -28(%rbp)`
- `movl $300, -24(%rbp)`
- `movl $400, -20(%rbp)`
- `movdqa -16(%rbp), %xmm1`  
- `movdqa -32(%rbp), %xmm0`  
- `padd %xmm1, %xmm0`  
- `movaps %xmm0, -48(%rbp)`
Coding for SIMD - intrinsics

- Use the SIMD registers explicitly using **intrinsics**, without having to write assembly code

**Advantages:**
- As a programmer, you have good control over the instructions that are generated
- The compiler will manage the register allocation (better than hand-written assembly)

**Disadvantages:**
- Code no longer portable to different architectures or you need to provide alternative code for non-SIMD processing
- If not done carefully, automatic glue code (e.g., cast, etc.) may make the code inefficient.
- Code readability decreases
Compiler intrinsics

```c
#include <stdlib.h>
#include <stdio.h>
#include <xmmintrin.h>

int main (int argc, char **argv){
    int a[4], b[4], c[4];
    __m128i x, y;
    b[0] = 100; b[1] = 200; b[2] = 300; b[3] = 400;
    x = _mm_loadu_si128((__m128i*)a);
    y = _mm_loadu_si128((__m128i*)b);
    x = _mm_add_epi32 (x, y);
    _mm_storeu_si128((__m128i*)c, x);
    printf("c = [ %i, %i, %i, %i ]\n", c[0], c[1], c[2], c[3]);

    return EXIT_SUCCESS;
}
```

**Intrinsics:** compilers wrap up asm instructions as functions
- Can use them by calling a function with the right parameters
- `_<vector_size>_<intrin_op>_suffix`<br>`<vector size>`: mm for 128-bit, mm256 and mm512<br>`<intrin_op>`: add, sub, mul, etc.<br>`<suffix>`: ps (float), pd (double), epi32 (signed int), epu16 (unsigned 16-bit integer), etc.

**Example** with AVX 128-bit intrinsics
- `_m128i` ← XMM register for all integer types
- `_mm_loadu_si128` ← load (unaligned) 128-bits from memory<br>  - corresponds to vmovdqu assembler instruction
- `_mm_add_epi32` ← vector add four signed 32-bit integers<br>  - corresponds to vpadd assembler instruction
- `_mm_storeu_si128` ← store (unaligned) 218-bits to memory
Exploiting SIMD for data processing and databases
Working with masks

- Conditional execution of vector elements
- Starting from AVX-512 almost all operations support masking

**Zero Masking** (selectively ignore some of the SIMD lanes)

- Example: add elements, but set those not selected by mask to zero:
  - vector add_vector_mask (mask k, vector a, vector b)
  - `__m512i _mm512_maskz_add_epi32 (__mask16 k, __m512i a, __m512i b)`

```
<table>
<thead>
<tr>
<th>a</th>
<th>4</th>
<th>2</th>
<th>1</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>k=0101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>
```
Masking with Merging / Blending

- Blend only instruction
  - vector blend_vector_mask(mask k, vector a, vector b)
  - __m512i _mm512_mask_blend_epi32 (__mmask16 k, __m512i a, __m512i b)

- Blend new result with previous result (“merge”)
  - vector add_vector_mask(vector src, mask k, vector a, vector b)
  - __m512i _mm512_mask_add_epi32 (__m512i src, __mmask16 k, __m512i a, __m512i b)

Note the difference between mask and maskz from previous slide.
Working with masks

**Compress and expand**

- Compress: `__m512i _mm512_maskz_compress_epi32(__mmask16 k, __m512i a)`
  - Also to memory: `compressstoreu`

- Expand: `__m512i _mm512_maskz_expand_epi32(__mmask16 k, __m512i a)`
  - Also to memory: `expandloadu`
Fundamental operations

- **Selective store**
  - Write a specific subset of the vector to a memory location contiguously. The subset is determined using vector/scalar register as the mask.

- **Selective load**
  - Loading from a memory location contiguously to a subset of vector lanes based on a mask. The lanes that are inactive in the mask, retain the old values.

- **Gather operation**
  - Load values from non-contiguous location.

- **Scatter operation**
  - Scatter executes stores to multiple locations.

Gather and scatter are not executed in parallel because the cache allows limited distinct accesses per cycle.
SIMD naturally fits a number of scan-based database tasks:

- **Arithmetics**

  ```sql
  SELECT price + tax AS net_price
  FROM orders
  ```

  This is what the code examples in the previous slide (10-13) were doing.

- **Aggregation**

  ```sql
  SELECT MIN(quantity)
  FROM lineitem
  ```

- How can this be done efficiently?
  - Similarly for `sum(·)`, `max(·)`, etc.
Selection queries are slightly more tricky:

- There are **no branching primitives** for SIMD registers
  - What would the semantics be anyhow?

- **Moving data** between SIMD and scalar registers is quite expensive
  - Either **go through memory**, move one data item at a time
  - Or **extract sign mask** from SIMD registers

```sql
SELECT quantity
FROM lineitem
WHERE price > 42
```
Selection example

```c
uint32_t scalar_sel(int32_t* in, int32_t count, int32_t val, int32_t* out){
    uint32_t out_pos = 0;
    for (int32_t i=0; i < count; i++)
        out[out_pos] = in[i];
    out_pos += (in[i] < val);
    return out_pos;
}

uint32_t vector_sel(int32_t* in, int32_t count, int32_t val, int32_t* out){
    uint32_t out_pos = 0;
    vector cmp = load_vector(val);
    for (int32_t i=0; i < count; i+=16) {
        vector inV = load_vector(in+i);
        mask mask = compare_vector(inV, cmp);
        compress_store(out+out_pos, mask, inV);
        uint32_t count = count_vector(mask);
        out_pos += count;
    }
    return out_pos;
}
```
Use-case: Tree-search

- In-memory tree look-ups
- Base case: **binary tree**, scalar implementation

```c
for (unsigned int i=0; i<n_items; i++) {
    k = 1; /* tree[1] is root node */
    for (unsigned int lvl=0; lvl<height; lvl++)
        k = 2 * k + (item[i] <= tree[k]);
    result[i] = data[k];
}
```

- Represent binary tree as an array tree[.] such that children of n are at positions 2n and 2n + 1

**Q1: Can we vectorize the outer loop?** (find matches for four input items in parallel)
- Iterations of the outer loop are independent, there is no branch in the loop body
- Need to use the scatter/gather instructions.

**Q2: Can we vectorize the inner loop?**
- Data dependency between loop iterations (variable k)
- Can speculatively navigate levels ahead.
Idea: Do comparisons for two levels in parallel

- Compare with nodes 1, 2 and 3 in parallel
- Follow link to node 6 and compare with nodes 6, 12 and 13
- etc.

src. Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. SIGMOD 2010
SIMD Blocking

- Pack tree sub-regions into SIMD registers

- Re-arrange data in memory for this.

src. Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. SIGMOD 2010
SIMD and scalar registers

e.g., search key 59

- SIMD to compare, scalar to navigate, movemask in-between

\[
\begin{array}{c|c|c|c|c|c|c|c}
 & 41 & 23 & 61 & 59 & 59 & 59 & 1 \cdots 1 & 1 \cdots 1 & 0 \cdots 0 \\
\hline
\text{SIMD cmp} & & & & 41 & 23 & 61 & & & \\
\hline
\text{movemask} & & & & 1 \cdots 1 & 1 \cdots 1 & 0 \cdots 0 & & & \\
\hline
\end{array}
\]

[Source: Kim et al., FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs, SIGMOD 2010]
Use mask value as **index** in **lookup** table.

Tree Navigation

Use mask value as index in lookup table.

src. Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. SIGMOD 2010
**Blocking** is a good idea also beyond SIMD

**Diagram:**
- **SIMD Blocking**
- **Cache line Blocking**
- **Page Blocking**

**Notations:**
- \(dK\): Depth of SIMD Blocking
- \(dL\): Depth of Cache Line Blocking
- \(dP\): Depth of Page Blocking
- \(dN\): Depth of Index Tree

**Source:** Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. SIGMOD 2010
SIMD Tree Search: Performance

Small tree – 64k keys
Large tree – 64M keys
Corei7 CPU with total 12.8 GHz frequency

Normalized Search Time

0 0.2 0.4 0.6 0.8 1

large tree small tree large tree small tree

CPU GPU

Default + Page Blocking + SW Pipelining
+ SIMD + Cache line Blocking

src. Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. SIGMOD 2010
Use case: Decompression

- Light-weight compressions schemes (e.g., numeric compression (NC), string compression (SC), dictionary compression (DC), etc.) is a good candidate for vectorized decompression.

- LWC NC is based on null suppression and encoding the resulting length of the compressed integer.
  - The integer value “$3_d$” can be stored by storing only “$11_b$” and ignoring the other thirty “$0_b$” bits.

src. Willhalm et al. *SIMD-Scan: Ultra-Fast in-Memory Table Scan using on-Chip Vector Processing Units*. VLDB 2009
Decompression – Step 1: Copy Values

**Step 1:** Bring data into proper 32-bit words.

- For this example we assume 128-bit wide SIMD registers (but, the method generalizes).

- Use **shuffle instructions** and a **mask** to move **bytes** within SIMD registers.

src. Willhalm et al. *SIMD-Scan: Ultra-Fast in-Memory Table Scan using on-Chip Vector Processing Units.* VLDB 2009
Step 2: Make all four words identically bit-aligned.

Need a 32-bit SIMD shift instruction with 4 variable shift amounts
As 128-bit SIMD shift with variable amount is not supported

src. Willhalm et al. *SIMD-Scan: Ultra-Fast in-Memory Table Scan using on-Chip Vector Processing Units*. VLDB 2009
Decompression – Step 3: Shift and Mask

**Step 3:** Word-aligned data and mask out invalid bits.

- Shift right for 3 bits to make it 32-bit aligned:
  - `__m128i shifted = _mm_srli_epi32(in, 3);`
- Mask out the invalid bits:
  - `__m128i result = _mm_si128(shifted, maskval);`

src. Willhalm et al. *SIMD-Scan: Ultra-Fast in-Memory Table Scan using on-Chip Vector Processing Units.* VLDB 2009
Decompression -- Performance

Time to decompress 1 billion integers on Xeon X5560, 2.8 GHz

Optimized scalar – minimized cache miss rate and massive loop unrolling.

8/16/24-bit compressed data has better performance

src. Willhalm et al. *SIMD-Scan: Ultra-Fast in-Memory Table Scan using on-Chip Vector Processing Units*. VLDB 2009
Sometimes it may be sufficient to decompress only partially.

- e.g., selection queries on compressed data

Performance is higher for the bit cases up to 8-bits, where 8 values can be processed in parallel with one SSE register.

src. Willhalm et al. *SIMD-Scan: Ultra-Fast in-Memory Table Scan using on-Chip Vector Processing Units. VLDB 2009*
Use case: sort

Merge-sort can benefit from SIMD acceleration
- Block 1: run generation
- Block 2: merging of pre-sorted runs

Odd-even sorting network for four inputs

Bitonic merge network

The comparators can be implemented using min/max:
- Input variables $a$, $b$, $c$, and $d$
- Output variables $w$, $x$, $y$, and $z$

This will sort input items across SIMD registers, but not within a vector.

Before writing back to memory, SIMD register must be transposed (i.e., $w_2$ must be swapped with $x_1$, $w_3$ with $y_1$, etc.) with SIMD shuffle.

Sort – bitonic merge network

- **Idea:** larger networks can be built with help of merging networks that combine two pre-sorted inputs.

- Each comparator stage can be implemented using one `max` and one `min` SIMD instruction.

- Shuffle instructions in-between the three stages bring vector elements into their proper positions.

```plaintext
// A and B are input registers
B = shuffle_vector(B, B, imm1) – reverses vector B
L1 = min_vector(A, B)
H1 = max_vector(A, B) – L1 comparisons
L1p = shuffle_vector(L1, H1, imm2)
H1p = shuffle_vector(L1, H1, imm3) – L1 shuffles
L2 = min_vector(L1p, H1p)
H2 = max_vector(L1p, H1p) – L2 comparisons
L2p = shuffle_vector(L2, H2, imm4)
H2p = shuffle_vector(L2, H2, imm5) – L2 shuffles
L3 = min_vector(L2p, H2p)
H3 = max_vector(L2p, H2p) – L3 comparisons
L3p = shuffle_vector(L3, H3, imm6)
H3p = shuffle_vector(L3, H3, imm7) – L3 shuffles
```

The exact number of shuffles depends on the bit-width of the input items and the size of the registers (AVX, AVX-512).

Multi-way merge

Cache-conscious sort-merge

Multi-way merge sort the number of merge stages from $\log_2 N$ to $\log_k N$, where $k$ is the number of ways, and $N$ is the total number of records to sort.

Sort – results

1 thread, input table from 8MB to 2GB
Machine: Intel SandyBridge with 256-bit AVX instructions.

References

- Various papers cross-referenced in the slides
  - Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. SIGMOD 2010
  - Willhalm et al. SIMD-Scan: Ultra Fast In-Memory Table Scan using on-Chip Vector Processing Units. VLDB 2009
  - Chhugani et al. Efficient Implementation of Sorting on Multi-core SIMD CPU Architectures. VLDB 2008
  - Balkesen et al. Multi-core, Main-memory joins: sort vs. hash revisited. VLDB 2014
  - Polychroniou and Ross. Rethinking SIMD Vectorization for In-Memory Databases. SIGMOD 2015

- Lecture: *Data Processing on Modern Hardware* by Prof. Jens Teubner (TU Dortmund, past ETH)
- Lecture: *Data Processing in Modern Hardware* by Prof. Viktor Leis (Uni Jena, past TUM)

- Book: *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson
  - Chapter 4 and Appendix H

- Book: *An optimization guide for x86 platforms* by Agner Fog (TU Denmark).
  - Chapter 2: Optimizing sub-routines in assembly Language, Section 13: Vector Programming

- Intel 64 and IA-32 Architectures Software Developer’s Manual
  - Chapter 15 (Programming with Intel AVX-512)

- AMD64 Architecture Programmer’s Manual

- Check out the generated code from the compiler on various machines with godbolt.org
  - Compile the code with the appropriate flag –march=skylake-avx512 –O3
Appendix – source code examples
Intrinsics – Data Types

- 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 does the register allocation
Loading / storing 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); but, there is a convenient (but slow) intrinsic for that:
- \_\_m512i \_mmset\_epi32(int e15, ..., int e0)

- **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 / modulo (division by 2 can be emulated using shift)
- No overflow detection

```c
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);
    }
}
```
Logical and Bitwise Operations

- Logical: and, andnot, or, xor
- Rotate left (right) by some value: rol (ror)
- Rotate left (right) by different value: rolv (rorv)
- Shift left (right) by same value: slli (srli)
- Shift left (right) by different value: sllv (srlv)
- Convert different sizes (zero/sign-extend, truncate): cvt
  - 32 to 64: _m512i _mm512_cvtepi32_epi64 (__m256i a) (sign extend)
  - 32 to 64: _m512i _mm512_cvtepu32_epi64(__m256i a) (zero extend)
  - 64 to 32: _m256i _mm512_cvtepi64_epi32 (__m512i a) (truncate)
- Count leading zeros:lzcnt
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 taken a mask as input, which is equivalent to performing AND on the masks

- Assumes signed integers
  - to compare unsigned integers, flip the most significant bit of inputs using xor

- Result is a bitmap stored in a special “opmask” register (K1-K7) and is available as special data type (\_mmask8 to \_mmask64)
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)
Permute

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

```
a  4 2 1 7
   \  \  \  \
  1 0 3 0
  2 4 7 4
```
Gather

- 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 necessary faster than individual loads (unless one needs the result in SIMD register anyway)
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, __512i vindex, __m512i a, int scale)

Watch out for conflicts:
- During a scatter, when multiple indices have the same value, bad things can happen
- Test each element for equality with all other elements
- __m512i _mm512_conflict_epi32 (__m512i a)
Selection example

```c
uint32_t scalar_sel(int32_t* in, int32_t count, int32_t val, int32_t* out){
    uint32_t out_pos = 0;
    for (int32_t i=0; i < count; i++)
        if (in[i] < val)
            out[out_pos] = i;
    return out_pos;
}

uint32_t vector_sel(int32_t* in, int32_t count, int32_t val, int32_t* out){
    uint32_t out_pos = 0;
    __m512i cmp = _mm512_set1_epi32(val); for (int32_t i=0; i < count; i+=16) {
        __m512i inV = _mm512_loadu_si512(in+i);
        __mmask16 mask = _mm512_cmplt_epi32_mask(inV, cmp);
        _mm512_mask_compressstoreu_epi32(out+out_pos, mask, inV);
        uint32_t count = __builtin_popcount(mask);
        out_pos += count;
    }
    return out_pos;
}
```
Decompression example (n-bit compression)

```plaintext
set k to 0
for i from 0 to max_index/simd_width_bits {
    for j from 0 to simd_width_bytes {
        parallel_load ba from input[k*simd_width_bytes + j*n];
        shuffle ba to ca using shuffle_mask(ma0, ..., ma{simd_width_bytes-1});
        parallel_shift ca by (sa0, ..., sa{bits_shifted});
        parallel_store ca in output[i*simd_width_bytes + j*8];
        parallel_load bb from input[k*simd_width_bytes + j*n + n/2];
        shuffle bb to cb using shuffle_mask(mb0, ..., mb{simd_width_bytes-1});
        parallel_shift bc by (sb0, ..., sb{bits_shifted});
        parallel_store cb in output[i*simd_width_bytes + j*8 + 4];
    }
    increase k by n;
}
```