Part IV

Vectorization
Pipelining is one technique to leverage available hardware parallelism.

- Separate chip regions for individual tasks execute independently.
- Advantage: Use parallelism, but maintain sequential execution semantics at front-end (here: assembly instruction stream).
- We discussed problems around hazards in the previous chapter.
- VLSI technology limits the degree up to which pipelining is feasible. (H. Kaeslin. Digital Integrated Circuit Design. Cambridge Univ. Press.)
Hardware Parallelism

Chip area can as well be used for **other types of parallelism**:

```
+--------+          +--------+
|  in1   |          |  out1  |
|--------|          |--------|
|  in2   |          |  out2  |
|--------|          |--------|
|  in3   |          |  out3  |
```

```
<table>
<thead>
<tr>
<th>Task 1</th>
<th></th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Computer systems typically use identical hardware circuits, but their function may be controlled by different **instruction streams** $s_i$:

```
+--------+          +--------+
|  in1   |          |  out1  |
|--------|          |--------|
|  in2   |          |  out2  |
|--------|          |--------|
|  in3   |          |  out3  |
```

```
<table>
<thead>
<tr>
<th>PU</th>
<th></th>
<th>PU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
| PU     |          |        | PU     |
|--------|----------|--------|
|        |          |        |        |
|        |          |        |        |
| s1     | s2       | s3     |
```

Do you know an example of this architecture?
Most modern processors also include a **SIMD** unit:

- Execute same assembly instruction on a set of values.
- Also called **vector unit**; **vector processors** are entire systems built on that idea.
The processing model is typically based on **SIMD registers** or **vectors**:

\[
\begin{array}{cccc}
    a_1 & a_2 & \ldots & a_n \\
    b_1 & b_2 & \ldots & b_n \\
    a_1 + b_1 & a_2 + b_2 & \ldots & a_n + b_n \\
\end{array}
\]

Typical values (e.g., x86-64):

- 128 bit-wide registers ($\text{xmm0}$ through $\text{xmm15}$).
- Usable as $16 \times 8$ bit, $8 \times 16$ bit, $4 \times 32$ bit, or $2 \times 64$ bit.
Much of a processor’s **control logic** depends on the number of in-flight instructions and/or the number of registers, but **not** on the size of registers.

→ scheduling, register renaming, dependency tracking, . . .

**SIMD instructions** make **independence** explicit.

→ No data hazards within a vector instruction.
→ Check for data hazards only between vectors.
→ **data parallelism**

Parallel execution promises *n*-fold performance advantage.

→ (Not quite achievable in practice, however.)
How can I make use of SIMD instructions as a programmer?

1. **Auto-Vectorization**
   - Some compiler automatically detect opportunities to use SIMD.
   - Approach rather limited; don’t rely on it.
   - Advantage: platform independent

2. **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.)
/*
 * Auto vectorization example (tried with gcc 4.3.4)
 */
#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;
}
Resulting assembly code (gcc 4.3.4, x86-64):

```assembly
loop:
    movdqu (%r8,%rcx), %xmm0 ; load a and b
    addl $1, %esi
    movdqu (%r9,%rcx), %xmm1 ; into SIMD registers
    paddd %xmm1, %xmm0 ; parallel add
    movdqa %xmm0, (%rax,%rcx) ; write result to memory
    addq $16, %rcx ; loop (increment by
    cmpl %r11d, %esi ; SIMD length of 16 bytes)
    jb loop
```

© Jens Teubner · Data Processing on Modern Hardware · Winter 2018/19
/* Use attributes to trigger vectorization */
#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;

    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;
}
Resulting assembly code (gcc, x86-64):

```assembly
movl $1, -16(%rbp) ; assign constants
movl $2, -12(%rbp) ; and write them
movl $3, -8(%rbp) ; to memory
movl $4, -4(%rbp)
movl $100, -32(%rbp)
movl $200, -28(%rbp)
movl $300, -24(%rbp)
movl $400, -20(%rbp)
movdqa -32(%rbp), %xmm0 ; load b into SIMD register xmm0
paddq -16(%rbp), %xmm0 ; SIMD xmm0 = xmm0 + a
movdqa %xmm0, -48(%rbp) ; write SIMD xmm0 back to memory
movl -40(%rbp), %ecx ; load c into scalar
movl -44(%rbp), %edx ; registers (from memory)
movl -48(%rbp), %esi
movl -36(%rbp), %r8d
```

- Data transfers scalar ↔ SIMD go **through memory.**
3 Use C Compiler Intrinsics

- Invoke SIMD instructions directly via compiler macros.
- Programmer has good control over instructions generated.
- Code no longer portable to different architecture.
- Benefit (over hand-written assembly): compiler manages register allocation.
- Risk: If not done carefully, automatic glue code (casts, etc.) may make code inefficient.
/*
 * Invoke SIMD instructions explicitly via intrinsics.
 */
#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;
}
Resulting assembly code (gcc, x86-64):

```assembly
movdqu -16(%rbp), %xmm1 ; _mm_loadu_si128()
movdqu -32(%rbp), %xmm0 ; _mm_loadu_si128()
padd %xmm0, %xmm1 ; _mm_add_epi32()
movdqu %xmm1, -48(%rbp) ; _mm_storeu_si128()
```
SIMD and Databases: Scan-Based Tasks

SIMD functionality 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 on the previous slides did.

- **aggregation**

  ```sql
  SELECT COUNT(*)
  FROM lineitem
  WHERE price > 42
  ```

  How can this be done efficiently?

  Similar: \( \text{SUM}() \), \( \text{MAX}() \), \( \text{MIN}() \), \ldots
Selection queries are a slightly more tricky:

- There are no branching primitives for SIMD registers.
  → What would their 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.

Thus:

- Use SIMD to generate bit vector; interpret it in scalar mode.

ุม If we can count with SIMD, why can’t we play the $j += (\cdots)$ trick?
Decompression

Column decompression (↗slides 125ff.) is a good candidate for SIMD optimization.

- Use case: $n$-bit fixed-width frame of reference compression; phase 1 (ignore exception values).
  → no branching, no data dependence

- With 128-bit SIMD registers (9-bit compression):

$\begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\hat{v}_{13} & \hat{v}_{12} & \hat{v}_{11} & \hat{v}_{10} & \hat{v}_{9} & \hat{v}_{8} & \hat{v}_{7} & \hat{v}_{6} & \hat{v}_{5} & \hat{v}_{4} & \hat{v}_{3} & \hat{v}_{2} & \hat{v}_{1} & \hat{v}_{0} \\
\end{array}$

↗Willhalm et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. VLDB 2009.
Step 1: Bring data into proper 32-bit words:

- Use **shuffle instructions** to move **bytes** within SIMD registers.
- `__m128i out = __mm_shuffle_epi8(in, shufmask);`
Step 2: Make all four words identically bit-aligned:

SIMD shift instructions do not support variable shift amounts!
Decompression—Step 3: Shift and Mask

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

\[
\begin{align*}
&v_3 & & v_2 & & v_1 & & v_0 \\
&v_3 & & v_2 & & v_1 & & v_0 \\
&v_3 & & v_2 & & v_1 & & v_0 \\
\end{align*}
\]

- `__m128i shifted = _mm_srli_epi32(in, 3);`
- `__m128i result = _mm_and_si128(shifted, maskval);`
Decompression Performance

- Time to decompress 1 billion integers (Xeon X5560, 2.8 GHz).

Source: Wilhalm et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. VLDB 2009.
Some SIMD instructions require hard-coded parameters. Thus: **Expand** code explicitly for all possible values of $n$.

→ There are at most 32 of them.

→ Fits with operator specialization in column-oriented DBMSs.

Loading **constants** into SIMD registers can be relatively expensive (and the number of registers limited).

→ One register for shuffle mask and one register to shift data (step 2) is enough.

For larger $n$, a compressed word may span **more than 4 bytes**.

→ Additional tricks needed (shift and blend).
Sometimes it may be sufficient to decompress only partially.

*E.g.*, search queries $v_i < c$:

- Only shuffle and mask (but don’t shift).
Vectorized Predicate Handling: Performance

The performance improvement is generally higher for the bit cases up to 8 bits, where 8 values can be processed in parallel in one SSE register. There, the average speedup factor is 1.58 over all bit cases.

Figure 12. Speedup for decompression by vectorization

The speedup of the SIMD implementation for searching a value (full table scan) in 1B records is shown in Figure 13. The experimental test set for bit case \( \text{4g3588} \) consists of the natural numbers modulo \( 2^{4g_\times434g_25754g_2589} \). Again, the measurements were performed 10 times on a test program executing the search routine as described in Section 4.2, and the median of the 10 runs was used for computing the speedup. For the lower bit compression cases, the search result is very large for a single search value (e.g. if 2 bits are used, a quarter of our test data set is returned). For bit cases 27 onwards, special care is needed to handle compressed values that span across 5 Bytes as shown in Figure 7. As a result, this reduces the performance advantage to the extent that for bit case 31, the vectorized implementation was slower than the scalar version. However, the average speedup factor of a full table scan is still 2.16. In practice, the SIMD implementation is only used in bit cases where it is faster than its scalar counterpart, which is the dominant scenario.

Figure 13. Speedup of full table scan by vectorization

If the result of a full table scan is returned as a bit vector, the running time is independent of the number of hits. However, in case a list of indexes is returned, the running time increases for large results as storing the results cannot fully exploit the benefit of storing vector instructions. The best speedup is therefore achieved for very selective queries as graphed in Figure 14, which displays the Speedup vs. Selectivity. Again 1B entries were processed 10 times and the median was recorded. Each point in the graph displays the average speedup over all bit cases. The overall speedup average is 1.63.

Figure 14. Speedup of full table scan by selectivity

In real world scenarios, and according to our experience at SAP, the compression bits used to compact database columns are mainly in the range of 8 to 16 bits. Figure 15 shows the practical distribution of the compression bit cases against the running time contribution of the table scan routines for a typical customer scenario. Taking this distribution into account, the (weighted) speedup factor for a full table scan is 2.45 over all bit cases.

Figure 15: Running time distribution for customer workload

Finally, we executed the vectorized search in parallel on different processor cores to verify its scalability. Figure 16 shows that the vectorized search scales almost linearly up to eight cores that are installed on the evaluation system. The memory bandwidth leaves sufficient headroom for future processors with...
Use Case: Tree Search

Another SIMD application: in-memory **tree lookups**.

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 array `tree[·]` such that children of $n$ are at positions $2n$ and $2n + 1$. 

Can we vectorize the outer loop? (i.e., find matches for four input items in parallel)

- Iterations of the outer loop are independent.
- There is no branch in the loop body.

Many SIMD implementations do not support scatter/gather!
Can we vectorize the inner loop?

- **Data dependency** between loop iterations (variable k).
- Intuitively: Cannot navigate multiple steps at a time, since first navigation steps are not (yet) known.

But:

- Could *speculatively* navigate levels ahead.
“Speculative” Tree Navigation

Idea: Do comparisons for two levels in parallel.

E.g.,
1. Compare with nodes 1, 2, and 3 in parallel.
2. Follow link to node 6 and compare with nodes 6, 12, and 13.
3. ....
Pack tree sub-regions into SIMD registers.

\[ \sim \quad \text{Re-arrange data in memory for this.} \]

\[ \rightarrow \quad \text{Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. } \textit{SIGMOD 2010}. \]
SIMD and Scalar Registers

E.g., search key 59:

SIMD cmp

\[
\begin{array}{ccc}
41 & 23 & 61 \\
59 & 59 & 59 \\
1\ldots1 & 1\ldots1 & 0\ldots0 \\
\end{array}
\]

\[00001100\] scalar register

\[\rightarrow\] SIMD to compare, scalar to navigate, \text{movemask} in-between.
Tree Navigation

Use scalar movemask result as index in lookup table:

![Image](https://via.placeholder.com/150)

Use mask value as index

Lookup Table

<table>
<thead>
<tr>
<th>Lookup Index</th>
<th>Child Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>N/A</td>
</tr>
<tr>
<td>010</td>
<td>N/A</td>
</tr>
<tr>
<td>110</td>
<td>N/A</td>
</tr>
<tr>
<td>001</td>
<td>N/A</td>
</tr>
<tr>
<td>101</td>
<td>N/A</td>
</tr>
<tr>
<td>011</td>
<td>N/A</td>
</tr>
<tr>
<td>111</td>
<td>3</td>
</tr>
</tbody>
</table>

Image source: Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. SIGMOD 2010.
Hierarchical Blocking

**Blocking** is a good idea also beyond SIMD.

- **dN** Depth of Index Tree
- **dP** Depth of Page Blocking
- **dL** Depth of Cache Line Blocking
- **dK** Depth of SIMD Blocking

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

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