Exercise Session 6

Data Processing on Modern Hardware
263-3502-00L — Fall Semester 2012

Cagri Balkesen
cagri.balkesen@inf.ethz.ch

Department of Computer Science ETH Zurich, Switzerland

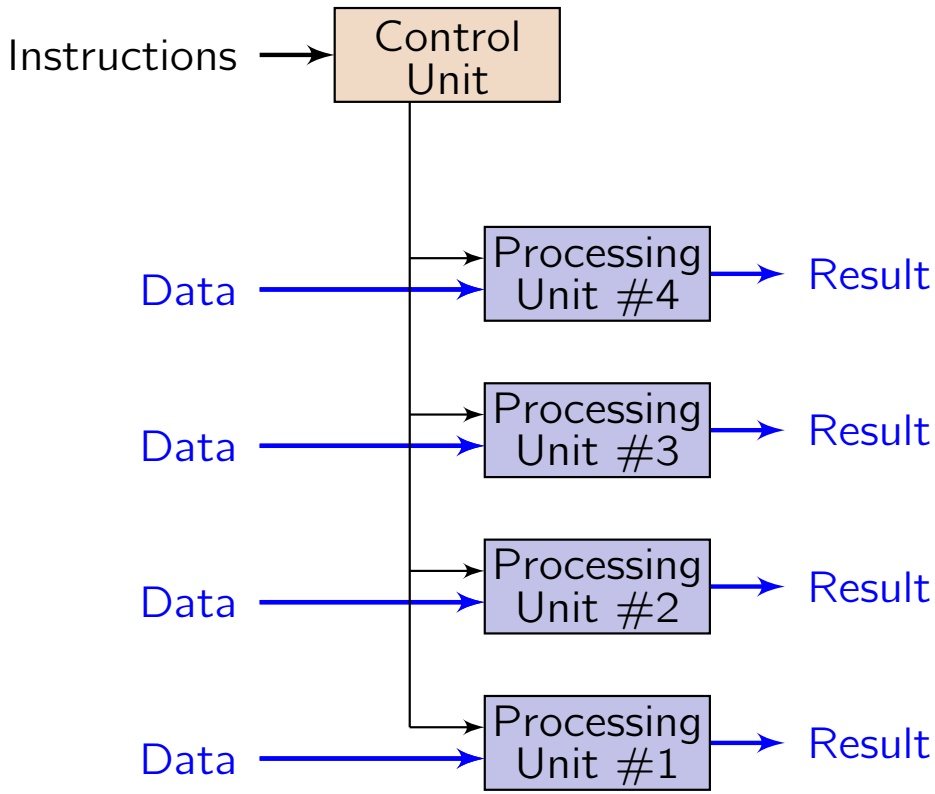
25 October 2012

Flynn's Taxonomy

Computer architecture classification according to Flynn [Fly72]:

- SISD: Single Instruction Single Data Stream
- **SIMD**: Single Instruction Multiple Data Streams
- MISD: Mutiple Instruction Single Data Stream
- MIMD: Multipe Instruction Multiple Data Streams

SIMD:



Early SIMD Machines

CDC STAR-100

- Released 1974
- Vector super computer supporting memory-to-memory vector operations

Cray X-MP/28 (CAB)

- Introduction 1982
- Word length: 64 bit
- Memory: 8M words
- Register-to-register vector operations
- 8 vector registes with **up to** 64 words each

ETH Cray X-MP/28 (CAB)

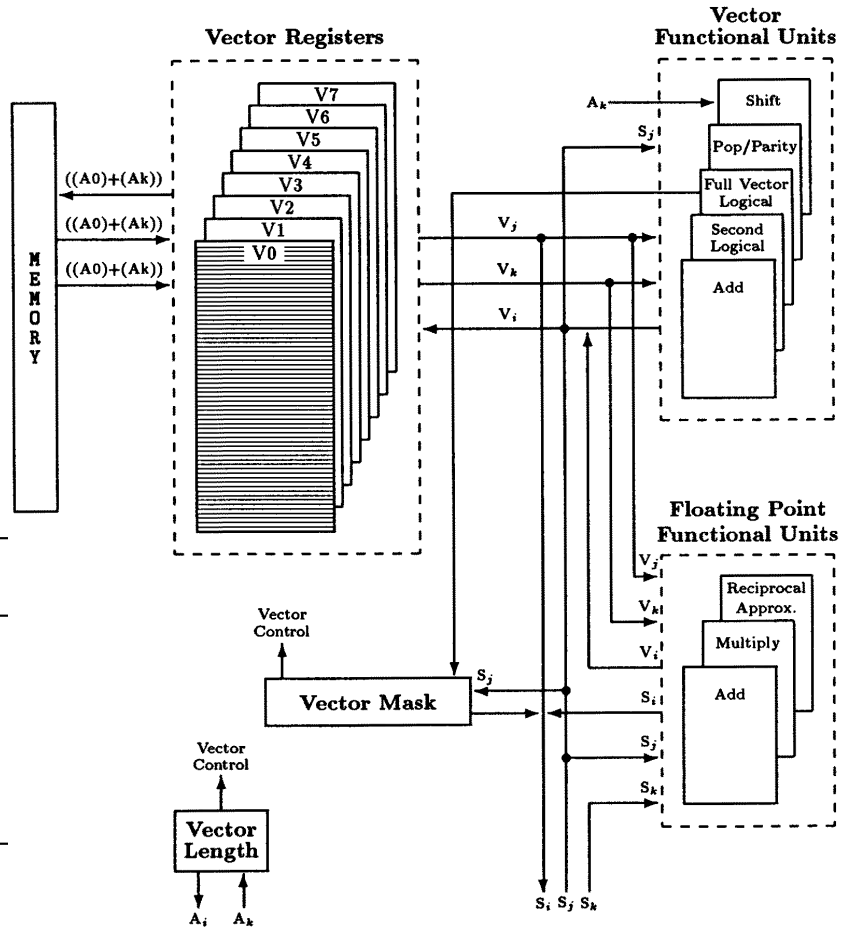
Variable Vector Length on a Cray X-MP

Example:

- Vector integer addition of first 53 elements of two vector registers
- `V1`, `V2`, `V3`: Vector Register
- `V3[0:52]` ← `V1[0:52] + V2[0:52]`

Line	Instr.	Description
1	`A1 53`	Set addr. reg. A1 to 53
2	`VL A1`	Set vector length to 53
3	`V3 V1+V2`	Perform addition

- Latency of add: VL+8 = 61 cycles (1 cycle ≡ 9.5 ns)



source [RR89]

Vector Units in Modern CPUs

PowerPC

- AltiVec (Motorola/Freescale)
- VMX (IBM), Velocity Engine (Apple)
- PowerPC G4, G5 and Cell BE
- 32 × 128-bit vector registers
- Dedicated vector unit

UltraSPARC

- VIS: Visual Instruction Set
- Uses 64-bit FPU registers

AMD

- 3D Now! (since AMD K6-2)
- Integer + single precision floating point

Intel

- MMX
  - Since Pentium MMX
  - 8 × 64-bit registers (alias to FPU stack)
  - Before Pentium III no vector unit
- SSE–SSE4
  - Introduced in Pentium III
  - Dedicated vector unit, combined with MMX
- AVX
  - 256-bit registers
  - 3 operand, non-destructive instructions

Intel SSE

SSE

- Since Pentium III
- 8× 128-bit vector registers `xmm0`, . . . , `xmm7`
- Single precision FP
- Dedicated vector unit but shared resources with FPU

SSE2

- Since Pentium IV
- Double precision FP
- Extends MMX registers to 128 bit
- Full integer support on XMM registers (without using MMX registers)

SSE3

- Since Pentium IV (Prescott)
- New "horizontal" operations

SSSE3

- Since Core 2 (Merom)
- New permutaiton instructions

SSE4

- Since Core 2 (Penryn)
- Dot product

x86-64

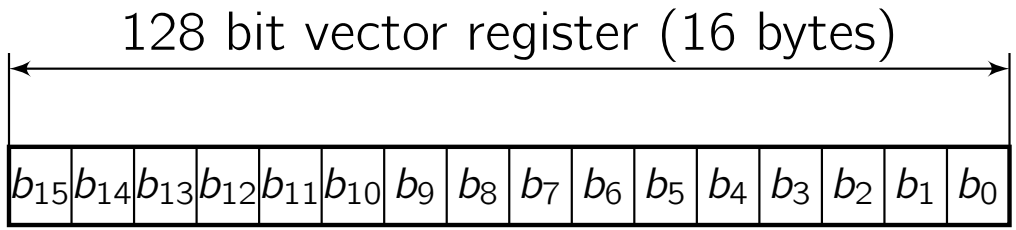
- Adds additional vector registers `xmm8`, . . . , `xmm15`

AVX

- 256-bits → `ymm8`, . . . , `ymm15`

SSE2 Vector Registers

Integer Data Types



Floating Point Data Types

Assignment: Warm-up Exercise

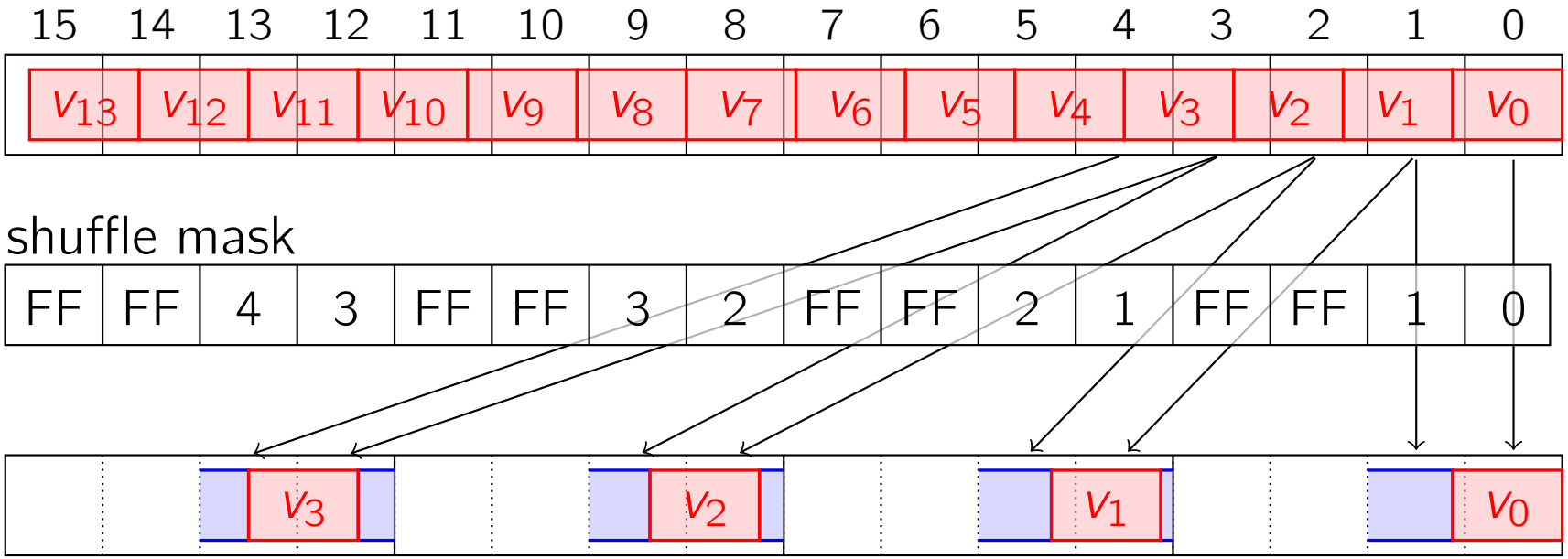
- Download skeleton C code from course website
- `compareandcount.c` does the following:
  - Array is filled with 100 mio numbers $\in [0, 99]$
  - Counts how many values > 42
- Implement SIMD-accelerated version using SSE Intrinsics
- Measure speed-up
- Caveats:
  - Result of SIMD-comparison $\rightarrow$ 0 if `true` but all 1's if false
  - In order to count, either shift or exploit that $111\ldots111 = -1$

Assignment: Column Value De(compression)

- In file `compression.c` the following compression is implemented
  - 32-to-8 bit
  - 32-to-9 bit
  - 32-to-7 bit
- Use C macros to switch between versions
- Serial decompression is implemented
- Execution time is measured and decompressed values validated
- Implement the following functions using SSE Intrinsics
  - `SIMD_decompress8to32(...)`
  - `SIMD_decompress9to32(...)`
  - `SIMD_decompress7to32(...)`

Decompression—Step 1: Copy Values

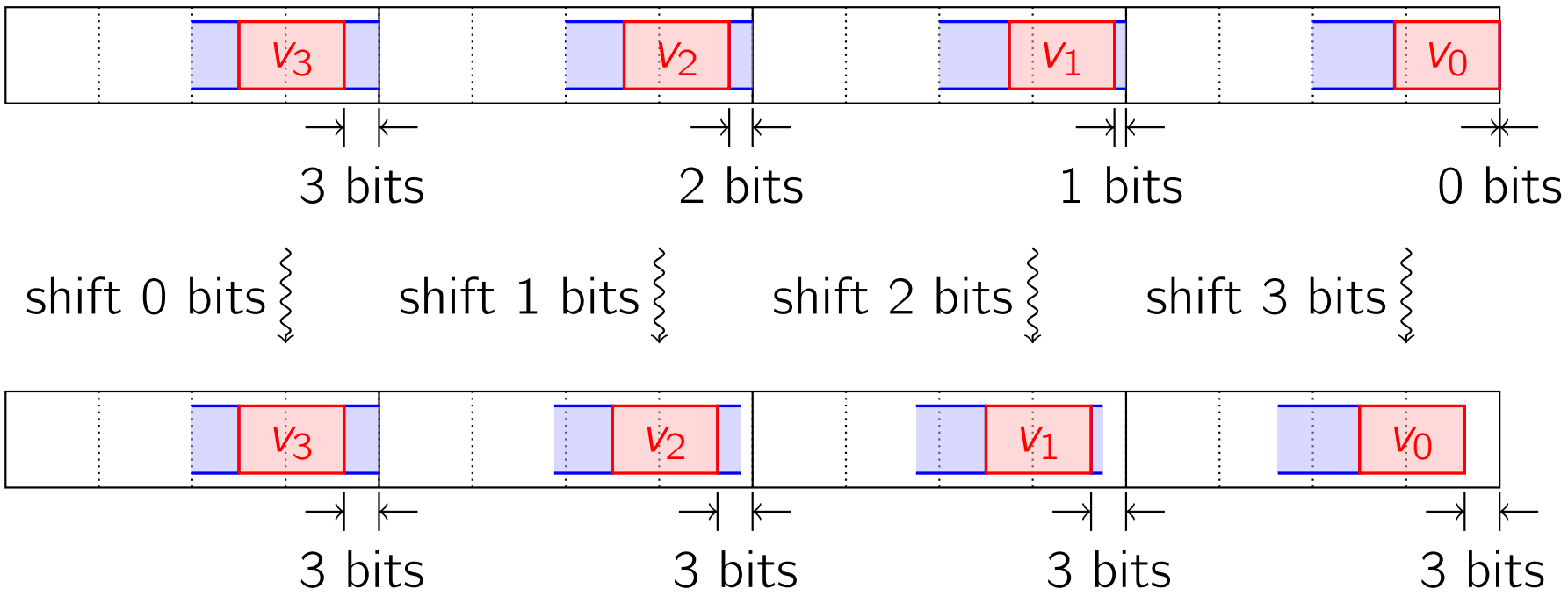
Step 1: Bring data into proper 32-bit words:



- Use **shuffle instructions** to move **bytes** within SIMD registers.
- `__m128i out = _mm_shuffle_epi8(in, shufmask);`

Decompression—Step 2: Establish Same Bit Alignment

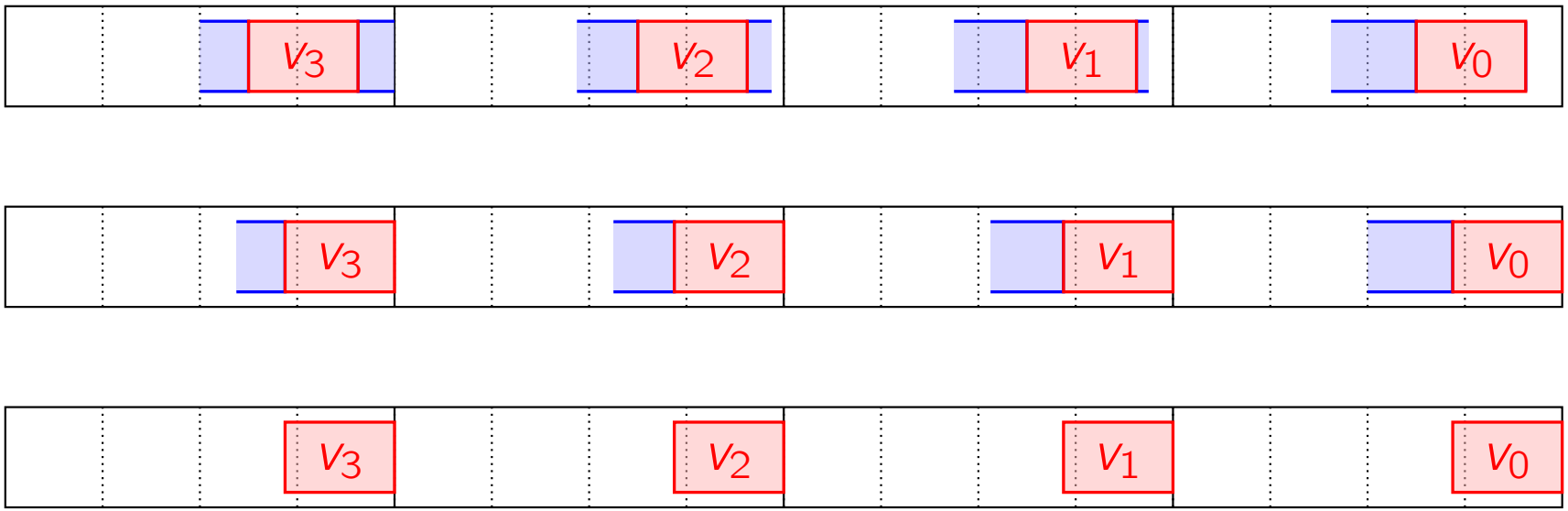
Step 2: Make all four words identically bit-aligned:



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:



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

Useful collection of SSE Intrinsics

Intrinsic Function	*Description*
`_mm_loadu_si128(src)`	Load data from memory into register
`_mm_storeu_si128(dest, reg)`	Store data back in memory
`_mm_set_epi32(v0,v1,v2,v3)`	Load four 32-bit integers into register
`_mm_set_epi8(v0,...,v15)`	Load sixteen 8-bit integers into register
`_mm_cmpgt_epi32(reg1, reg2)`	Greater compare of the four 32-bit values in the registers
`_mm_add_epi32(reg1, reg2)`	Addition of the four 32-bit values in the registers
`_mm_and_si128(reg, mask)`	Bitwise and of two registers → masking
`_mm_mullo_epi32(reg1, reg2)`	Multiplication of the four 32-bit values in the registers
`_mm_extract_epi32(reg, pos)`	Extract 32-bit integer at position `pos` form register
`_mm_shuffle_epi32(reg, mask)`	Shuffle 32-bit integers according to the shuffle mask
`_mm_srli_si128(reg, bytecnt)`	Shift entire register right (bytewise)
`_mm_slli_si128(reg, bytecnt)`	Shift entire register left (bytewise)
`_mm_srli_epi32(reg, bitcnt)`	Shift 32-bit integers in register to the right (bitwise)
`_mm_slli_epi32(reg, bitcnt)`	Shift 32-bit integers in register to the left (bitwise)

Rerfer to `http://www.intel.com/products/processor/manuals/` for details.

Auto-Vectorization in gcc

- Recent versions of gcc can auto-vectorize C code
- Use command line options:
  - `-ftree-vectorize` turn on auto-vectorization (default for `-O3`)
  - `-ftree-vectorizer-verbose=x` set reporting verbosity level of vectorizer
  - `-msse` to generate SSE code
  - `-msse2` to generate SSE2 code
  - `-msse3` to generate SSE3 code

gcc does not vectorize

- if code contains braches.
- unconstrained pointers are used (aliasing).
- uncountable loops.

Auto-Vectorization of Volcano Column-Store

Implementation of artithmetic operator in `engine.c`:

```
int next_arithop(int32_t *tuples,
        arithop_t *op)
{
...
switch (op->operation) {
case ADD:
  for (n=0; n<minnum; n++)
    tuples[n] = op->left_input[n]+
      op->right_input[n];
  break;
case SUB:
  for (n=0; n<minnum; n++) {
    tuples[n] = op->left_input[n]-
      op->right_input[n];
  break;
...
```

Compilation:

```
$ gcc -m64 -O3 -msse2 -ftree-vectorize
    -ftree-vectorizer-verbose=3
    -c engine.c
...
engine.c:104:  note:not vectorized:
unhandled data-ref
engine.c:109:  note:not vectorizer:
unhandled data-ref
...
engine.c:95:  note:  vectorized 0 loops
in function.
```

- Auto-vectorization failed
- op pointer, possible aliasing?
- $\Rightarrow$ restrict pointer

Vectorized Volcano Column-Store using SSE

Using restricted pointers in loops:

```
int next_arithop(int32_t *tuples,
        arithop_t *op)
{
...
int32_t* __restrict l=op->left_input;
int32_t* __restrict r=op->right_input;
...
switch (op->operation) {
case ADD:
  for (n=0; n<minnum; n++)
    tuples[n] = l[n]+r[n];
  break;
case SUB:
  for (n=0; n<minnum; n++) {
    tuples[n] = l[n]-r[n];
  break;
...
```

Compilation:

```
$ gcc -m64 -O3 -msse2 -ftree-vectorize
    -ftree-vectorizer-verbose=3
    -c engine.c
...
engine.c:113:  note:  Alignment of
access forced using peeling.
engine.c:113:  note:  LOOP VECTORIZED
engine.c:121:  note:  Alignment of
access forced using peeling.
engine.c:121:  note:  LOOP VECTORIZED
...
engine.c:102:  note:  vectorized 5
loops in function.
```

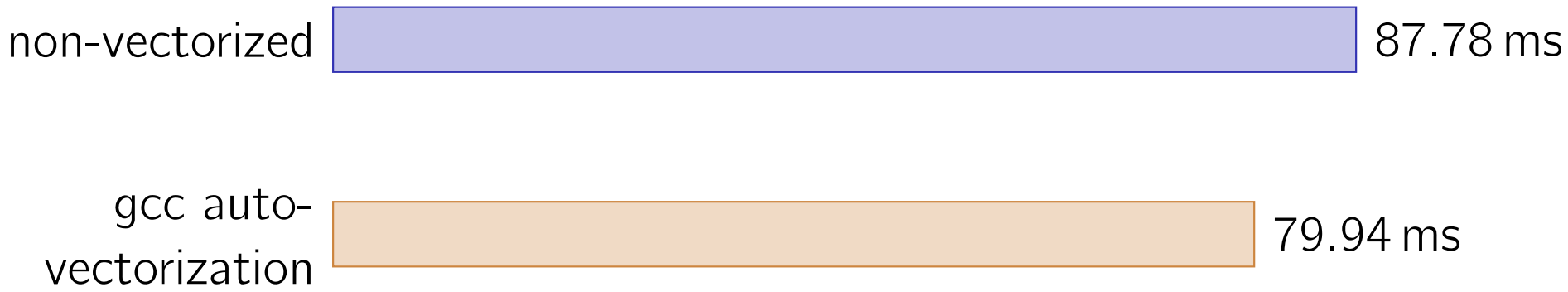
- $\Rightarrow$ successful vectorization

Speedup SSE Vectorization of Volcano Column-Store

Query:

```
SELECT sum(orderkey+linenumber*shipdate)
FROM lineitems
```

data set: 6 million rows
CPU: Core 2 Quad Q6700 2.66 GHz



→ Speedup = 1.10 Load is memory-bound, **not** CPU-bound.

Hand in your Results

- E-Mail to cagri.balkesen@inf.ethz.ch
- Subject = DPMH:assignment4_{your netzname}
- Body = Description of CPU you tested on, e.g., Xeon QuadCore L5520,(4x 2267MHz)
- Attach plots + raw data
- Attach source code (optional)

References

[Fly72] Michael J. Flynn.

Some computer organizations and their effectiveness.

IEEE Transactions on Computers, 21(9):948–960, September 1972.

[RR89] Kay A. Robbins and Steven Robbins.

The Cray X-MP/Model 24, chapter 5, pages 21–43.

Springer LNCS, 1989.

[WPB+09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner.

Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units.

PVLDB, 2(1):385–394, 2009.

[ZR02] Jingren Zhou and Kenneth A. Ross.

Implementing database operations using SIMD instructions.

In *SIGMOD '02*, pages 145–156, Madison, Wisconsin, USA, 2002.