Vectorization

Leveraging Modern Processing Capabilities
Hardware Parallelism

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 before.
- VLSI technology limits the degree up to which pipelining is feasible [Kaeslin, 2008]
Hardware Parallelism

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

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

Do you know an example of this architecture?
Vectorization

SIMD: Single Instruction, Multiple Data

–

Vectorized Execution
Special Instances (SIMD)

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.
SIMD Programming Model

The processing model is typically based on SIMD registers or vectors:

Typical values (e.g., x86-64):
- 128 bit-wide registers (xmm0 through xmm15).
- Usable as $16 \times 8$ bit, $8 \times 16$ bit, $4 \times 32$ bit, or $2 \times 64$ bit.
SIMD Programming Model

- 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.)
Coding for SIMD

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):

```
loop:
  movdqu (%r8,%rcx), %xmm0 ; load a and b
  addl $1, %esi
  movdqu (%r9,%rcx), %xmm1 ; into SIMD registers
  padd %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
```
/* 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.
Coding for SIMD

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()
paddq    %xmm0, %xmm1         ; _mm_add_epi32()
movdqu   %xmm1, -48(%rbp)     ; _mm_storeu_si128()
```
SIMD Instruction Sets – History

Started for desktop PCs with Intel’s MMX (1996)
- 8 registers (MM0 - MM7)
- Each 64-bit wide

3DNow! by AMD (1998)

AltiVec instruction set (between 1996 and 1998)
- By Apple, IBM, Motorola
- 32 registers
- Each 128-bit wide
SIMD Instruction Sets – History

Intel’s answer: SSE instruction set (1999)

- *Streaming SIMD Extensions*
- 8-16 registers (XMM0-XMM15)
- Each 128-bit wide
- SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2

Increasing SIMD widths: AVX (2008)

- *Advanced Vector Extensions*
- By Intel and AMD
- 8-16 registers (XMM0,YMM0 – XMM15,YMM15)
- Each 256-bit wide
- Extension of SSE instructions to operate on 256-bit
- AVX, AVX2, AVX-512 (2013)
SSE Instruction Set\(^2\) – Contents

Data types
- \texttt{\_m128d} for double-precision floating point
- \texttt{\_m128} for single-precision floating point
- \texttt{\_m128i} non floating-point data

Arithmetical operations
- \texttt{\_mm\_add, \_mm\_mul, \_mm\_div, \_mm\_sub}
- Horizontal operations for SSE3 and higher
- Comparison operators (create bit mask)

Logical operations
- \texttt{\_mm\_and, \_mm\_or, \_mm\_andnot, \_mm\_xor}

\(^2\)https://software.intel.com/sites/landingpage/IntrinsicsGuide/
SSE Instruction Set\(^2\) – Contents II

Move/Blend operations

- Move parts of a float value
- Blending: only selected values are copied
- Shifting in zeros/ones
- Shuffle/permute data in registers

Load/Store operations

- Loading/storing different data types
- Often special "h" or "l" operations for float
- Often also unaligned access

\(^2\)https://software.intel.com/sites/landingpage/InstructionGuide/
SIMD Instructions – Limitations

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

- Some SIMD instructions require hard-coded parameters.
  Thus: **Expand** code explicitly for all possible values of $n$.
  → Fits with operator specialization in column-oriented DBMSs
SIMD Instructions – Limitations 2

Data alignment – **Alignment Hazard**

- Operates mostly on 16-byte (128-bit) aligned data
- Unaligned access much slower

![Diagram showing data alignment and cache line](image-url)
SIMD Instructions – Alignment Hazard

How to avoid alignment hazards?

• Process unaligned data beforehand

```c
int alignment_offset = ((intptr_t)sse_array)%sizeof(__m128i); for(unsigned int i=0;i<alignment_offset/sizeof(int);i++){ //Process unaligned data } // Process aligned data using SIMD
```

• Align pointer of allocated memory to aligned address:

```c
/* Make newp a pointer to a 64-bit aligned array
   of NUM_ELEMENTS 64-bit elements. */
double *p, *newp;
p = (double*)malloc (sizeof(double)*(NUM_ELEMENTS+1));
newp = (p+7) & (~0x7);
```
Vectorization

SIMD for Database Tasks
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}(\cdot), \text{MAX}(\cdot), \text{MIN}(\cdot), \ldots
SIMD and Databases: Scan-Based Tasks

Selection queries are a slightly more tricky:

- **Missing branching primitives** for SIMD registers.

```cpp
for (unsigned int i = 0; i < num_tuples; ++i)
    if (lineitem[i].quantity < n)
        poslist[pos++] = i;
```

- **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.

If we can count with SIMD, why can’t we play the pos ++ trick?

```cpp
for (unsigned int i = 0; i < num_tuples; ++i)
    poslist[pos] = i;
    pos += (lineitem[i].quantity < n);
```
SIMD Scan with Bit Mask Evaluation

```c
for(unsigned int i=0;i<sse_array_length;i++){
    read_value=_mm_load_si128(&sse_array[i]);
    __m128 comp_result = (__m128) mm_cmplt_epi32(read_value,comp_val);
    int mask= _mm_movemask_ps(comp_result);
    if(mask){
        for(unsigned j=0;j<sizeof(__m128i)/sizeof(int);++j){
            if((mask >> j) & 1)
                result_array[pos++]=BASE_TID+j;
        }
    }
}
```
SIMD and Databases: Sorting

- Sorting is a compute intensive task
- Often involves control flow:
  - Quick sort
  - Insertion sort
  - Radix sort
- Is there a sorting strategy involving less control flow and more arithmetical operations?
- Merge sort using sorting/merging networks
SIMD Accelerated Merge Sort

- Merge sort uses 3 phases:
  - In-register sorting
    → Sorting networks
  - In-cache sorting
    → Merging networks
  - Out-of-cache sorting
    → Multi-way merging
A sorting network maps to a sequence of min/max operations with input $a, b, c, d$ and output $w, x, y, z$

\[
\begin{align*}
e & = \min(a, b) \\
f & = \max(a, b) \\
g & = \min(c, d) \\
h & = \max(c, d) \\
i & = \max(e, g) \\
j & = \min(f, h) \\
w & = \min(e, g) \\
x & = \min(i, j) \\
y & = \max(i, j) \\
z & = \max(f, h)
\end{align*}
\]

- Data passes several comparators
- Comparator emits:
  - Smaller value at top
  - Bigger value at bottom

adapted from [Balkesen et al., 2013]
SIMD-Accelerated Sorting Network

- Each SIMD lane is sorted
- Sorted lists have to be merged

adapted from [Chhugani et al., 2008]
**SIMD-Accelerated Merge Network**

**Odd-Even Merge Network**

- Inputs sorted in same order
- 6 min/max operations
- Masking/blending needed
SIMD-Accelerated Merge Network

Bitonic Merge Network

- Second input in reverse order – 1 shift needed
- 6 min/max operations
- No masking/blending needed – each register is rewritten

Better suited for SIMD!
Conclusion – Second Part

- SIMD = Single Instruction, Multiple Data
- Programming for SIMD
- Limitations of SIMD
- SIMD for database tasks
  - Scans
  - Sorting
CoGaDB

CogaDB (Column Oriented GPU accelerated DBMS):
- Main-memory DBMS
- Column store
- Several variants of scan operator includes:
  - Loop unrolled
  - Branch free
  - SIMD accelerated
  - Parallelized
  - Combinations of them
Invitation

Your are invited to join our research on code optimizations and databases on new hardware, e.g., in form of:

- Bachelor or master thesis
  → Survey / implementation of further code optimizations

- “Scientific Project: Data Management on new Hardware”

- Scientific individual project

Contact me! david.broneske@iti.cs.uni-magdeburg.de
/*
 * 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);
    // What is the output of this print?
    printf ("c = [ %i, %i, %i, %i ]\n", c[0], c[1], c[2], c[3]);

    return EXIT_SUCCESS;
}


