Hardware-Sensitive Database Operations - II

Balasubramanian (Bala) Gurumurthy
So Far...

- Hardware evolution and current challenges
- Hardware-oblivious vs. hardware-sensitive programming
- Pipelining in RISC computing
- Pipeline Hazards
  - Structural hazard
  - Data hazard
  - Control hazard
- Resolving hazards
  - Loop-Unrolling
  - Predication
Part II

We will see

- **Vectorization**
  - SIMD Execution
  - SIMD in DBMS Operation

- **GPUs in DBMSs**
  - Processing Model
  - Handling Synchronization
Vectorization

Leveraging Modern Processing Capabilities
Hardware Parallelism

One we know already: **Pipelining**

- Separate chip regions for individual tasks to execute independently
- Advantage: parallelism + sequential execution semantics
- We discussed problems of **hazards**
- VLSI tech. limits degree up to which pipelining is feasible  
  [Kaeslin, 2008]
Hardware Parallelism

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

![Diagram of hardware parallelism](image)

Computer systems typically use identical hardware circuits, but their function may be controlled by different instruction stream $S_i$:

![Diagram of instruction stream](image)
Special instances

Example of this architecture?
Special instances

Example of this architecture?

- This is your Multi-core CPU!
- Execution - Multiple Instruction Multiple Data (MIMD) fashion
- Single core is SISD: Single Instruction, Single Data
SIMD: Single Instruction Multiple Data

- Vectorized Execution
Special Instances (SIMD)

Modern Processors include 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**: 

32 Bits

\[
\begin{align*}
& a_1 \\
& b_1 \\
& + \\
& a_1 + b_1
\end{align*}
\]
The processing model is typically based on **SIMD registers** or **vectors**:

![Diagram of SIMD Processing Model](image)

**Used as:** 16x8 bit, 8x16 bit…

**128 Bits**
SIMD Programming Model

- Processor control logic depends on No. of instructions and/or No. of registers, **but not on size of registers**
  - Scheduling, register renaming, dependency tracking,...
SIMD Programming Model

- Processor control logic depends on No. of instructions and/or No. of registers, **but not on size of registers**
  - Scheduling, register renaming, dependency tracking,…

- SIMD make independence explicit
  - **Data parallelism**
  - No data hazard within vector execution
  - Data hazard check only within vector
SIMD Programming Model

- Processor control logic depends on No. of instructions and/or No. of registers, **but not on size of registers**
  - Scheduling, register renaming, dependency tracking,…

- SIMD make independence explicit
  - **Data parallelism**
    - No data hazard within vector execution
    - Data hazard check only within vector

- Parallel execution promises n-fold performance advantage
  - Not in practise however!
Coding SIMD

How can a programmer exploit SIMD instructions?

1. **Auto-vectorization**
   - Some compilers auto-detect opportunities for using SIMD
   - Approach rather limited; don’t rely it.
   - Advantage: platform independent
/*
 * 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
  addl $1, %esi
  movdqu (%r9,%rcx), %xmm1
  paddd %xmm1, %xmm0
  movdqa %xmm0, (%rax,%rcx)
  addq $16, %rcx
  cmpl %r11d, %esi
  jb  loop
```
Resulting Assembly Code (gcc 4.3.4, x86-64):

```asm
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
```

 xmm___ -SIMD registers (128 bits)

https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
Coding SIMD

How can a programmer exploit SIMD instructions?

1. **Auto-vectorization**
   - Some compilers auto-detect opportunities for using SIMD
   - Approach rather limited; don’t rely it.
   - Advantage: platform independent

2. **Compiler attributes**
   - Use `__attribute__((vector_size(...)))` annotations to state your intentions
   - Advantage: platform independent
     
     (Compiler generates non-SIMD code if platform does not support it)
/* 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 4.3.4, x86-64):

```
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
paddd -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
```
Coding SIMD

3. **Use C Compiler Intrinsics**
   - Invoke SIMD instructions directly via *compiler macros*
   - Programmer has good control over instructions generated
   - No code portability!
   - Benefit: Compiler manager register allocation
   - Risk: If not done carefully, automatic glue code (e.g. casts) may make code inefficient
/* Invoke SIMD instructions explicitly via intrinsics. */
#include <stdlib.h>
#include <stdio.h>

#include <xmmintrin.h>

#include <xmmmintrin.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):

```
mmovdqu -16(%rbp), %xmm1 ; _mm_loadu_si128()
movdqu -32(%rbp), %xmm0 ; _mm_loadu_si128()
paddd %xmm0, %xmm1 ; _mm_add_epi32()
movdqu %xmm1, -48(%rbp) ; _mm_storeu_si128()
```
SIMD Instruction Sets - History

Initially introduced with Intel’s MMX in desktop PCs (1997)
- 8 registers (MM0 - MM7)
- Each 64-bit wide

“3DNow!” AMD counterpart (1998)
- Floating point support
- Dropped support in 2010

AltiVec instruction set (1996 - 1998)
- By Apple, IBM, Motorola
- 32 registers (128 bit wide)

https://en.wikipedia.org/wiki/AltiVec
https://en.wikipedia.org/wiki/3DNow!
SIMD Instruction Sets - History

Intel’s evolution: SSE instruction set (1999)
- Streaming SIMD Extensions
- 8 - 16 registers (XMM0-XMM15)
- 128 bit wide
- SSE, SSE2, SSE3, SSSE3...

Increasing widths: AVX (2008)
- Advanced Vector eXtensions
- from 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

Data Types
- _m128d - double-precision floating point
- _m128 - single-precision floating point
- _m128i - non-floating point

Arithmetics
- _mm_add, _mm_mul,...
- Horizontal operations for SSE3 and higher

Comparisons
- _mm_cmplt, _mm_cmpgt,...
- Creates bit mask

Logical Operation
- _mm_and, _mm_or, _mm_andnot, _mm_xor
SSE Instruction Set\textsuperscript{2} - Cont.

Move / blend

- Move points of a float value
- Blending: only selected values are copied (uses masks to select value)
- Shifting in zeros / ones
- Shuffle data in registers

Load/Store operation

- special “h” or “l” operations for float
- often also unaligned access (gather operation)

\textsuperscript{2}https://software.intel.com/sites/landingpage/IntrinsicsGuide/
SSE Instruction Set\(^2\) - Cont.

Move / blend

- Move points of a float value
- Blending: only selected values are copied (uses masks to select value)
- Shifting in zeros / ones
- Shuffle data in registers

Load/Store operation

- special “h” or “l” operations for float
- often also unaligned access (gather operation)

What else are we missing?

What are the limitations?
SSE Instruction - Limitations

- There are no **branching primitives** for SIMD registers
  - How to perform them anyway?
- Some SIMD instructions require hard-coded parameters
  - Code must be explicitly expanded for possible values of N.
  - E.g. `_mm_blend_epi16`
SSE Instruction - Limitations 2

Data alignment - **Alignment hazard**
- Operates best on 16-byte aligned data
- Unaligned access is slower
SSE Instruction - Limitations 2

Data alignment - **Alignment hazard**

- Operates best on 16-byte aligned data
- Unaligned access is slower
SSE Instruction - Limitations 2

Data alignment - **Alignment hazard**

- Operates best on 16-byte aligned data
- Unaligned access is slower
SSE Instruction - Alignment hazard

How to avoid it?
SSE Instruction - Alignment hazard

How to avoid it?

- 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
```
SSE Instruction - Alignment hazard

How to avoid it?

- 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

/* 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);*/
SIMD for Database Tasks
SIMD fits naturally for scan-based tasks:

- **arithmetics**
  
  ```sql
  SELECT price + tax AS net_price
  FROM orders
  ```

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

How can we do this efficiently?

also, SUM(.), MAX(.), MIN(.)...
SIMD in DBMS - Scanning

- **Selection** - slightly tricky
  - No branching primitives!

  ```c
  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 move one data item at a time or extract sign mark from SIMD register

Thus:

Use SIMD to generate **bit vector; interpret** in scalar mode
SIMD in DBMS - Scanning

- **selection** - slightly tricky
  - No branching primitives!

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

**Why not use pos++ trick for non-branching selection?**

```c
for (unsigned int i=0; i<num_tuples; ++i)
  poslist[pos] = i;
  pos += (lineitem[i].quantity < n);
```
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 in DBMS - Sorting

- Sorting is a compute intensive operation
- Involves lot of control flow
  - E.g. Quick sort, insertion sort, radix sort
- Is there a sorting involving less control flow and more arithmetical operations?
SIMD in DBMS - Sorting

- Sorting is a compute intensive operation
- Involves lot of control flow
  - E.g. Quick sort, insertion sort, radix sort
- Is there a sorting involving less control flow and more arithmetical operations?
- Merge sort using sorting / merging networks
SIMD in DBMS - Sorting

- Merge-sort uses three phases
  - In-register sorting
    - Sorting networks
  - In-cache sorting
    - Merging networks
  - Out-of-cache sorting
    - k-way merging
Sorting Network

- Data passes several comparators
- Comparator emits:
  - Smaller value at top
  - Bigger value at bottom
<table>
<thead>
<tr>
<th>9</th>
<th>15</th>
<th>3</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>4</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>2</td>
<td>18</td>
</tr>
</tbody>
</table>

Input registers

adapted from [Chhugani et al., 2008]
SIMD with Sorting Network

adapted from [Chhugani et al., 2008]
SIMD with Sorting Network

adapted from [Chhugani et al., 2008]
SIMD with Sorting Network

adapted from [Chhugani et al., 2008]
SIMD with Sorting Network

adapted from [Chhugani et al., 2008]
SIMD with Sorting Network

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

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

Odd-Even Merge Network

- Input sorted in same order
- 6 min/max operations

Bala Gurumurthy | Hardware-Sensitive Database Operations
**SIMD Accelerated Merge Network**

**Odd-Even Merge Network**

- Input sorted in same order
- 6 min/max operations
- **Masking/blending needed**

Bala Gurumurthy | Hardware-Sensitive Database Operations
SIMD Accelerated Merge Network

Bitonic Merge Network

- Second input in reverse order - 1 shift needed
- 6 min/max operations
- No masking/blending required
SIMD Accelerated Merge Network

Bitonic Merge Network

- Second input in reverse order - 1 shift needed
- 6 min/max operations
- No masking/blending required

Better suited for SIMD!
Conclusion - Second Part

- SIMD = Single Instruction, Multiple Data
- Programming for SIMD
- Limitations for SIMD
- SIMD for database tasks
  - Scan
  - Sorting
Exploring Graphical Processing Units (GPUs)

Adding Heterogeneity into DBMS processing
Motivation

- Physical limitations in traditional processor (remember Moore’s law, power wall)
- Limited parallelism
- Data volume increases
Motivation

- Physical limitations in traditional processor (remember Moore’s law, power wall)
- Limited parallelism
- Data volume increases

Specialized processor tuned for high parallelism required
GPU

- Graphical Processing systems
- Traditionally used for accelerating graphical computation
- Contain many “light-weight“ cores
- Cores between 16 - 128
- Matured into programmable hardware

Characteristics

- SIMD fashioned execution
- Lower clock speeds than normal CPU core
- Improves performance by hiding latency
- Implements core graphics operators (mostly matrix calculations)
- **Connected external to a CPU via PCI-e Express**
- However, *execution starts* only when *data is available* in GPU Memory!
GPU Architecture and Programming

Instructions towards parallel processing
Architecture

● **Core**
  ○ Multi-threaded Massively data-parallel
  ○ SIMT (Single Instruction Multiple Thread) style processing

● **Memory**
  ○ Off-chip
    ■ device memory
    ■ texture memory
  ○ On-chip
    ■ local memory
    ■ caches
Architecture - Core

- Compute core grouped into workgroup
  - light-weight
  - numeric computation
  - Share a common L1 cache

- On-chip cache shared among all compute core

- Cores execute instruction in warps

Threads - GPU

- Traditional threads
  - Heavyweight
  - 10s of threads on 10s of cores
  - managed explicitly

- GPU warps
  - lightweight threads
  - 10000s threads on 100s cores
  - scheduled in batches
  - runs same code
Threads - GPU

- Traditional threads
  - Heavyweight
  - 10s of threads on 10s of cores
  - managed explicitly
- GPU warps
  - lightweight threads
  - 10000s threads on 100s cores
  - scheduled in batches
  - runs same code
- Handling threads
  - Not reducing latency - rather hide it
  - Data parallel execution and limited synchronization
  - In-order scheduling of batches
Programming Model

- Host-worker model
- CPU pushes data
- Once data transferred, CPU launches execution
- Fetch back results when done
- CPU perform other operation while GPU executes

- **Programming languages**
  - CUDA
  - OpenCL
Programming Model

- Instruction written as data-parallel kernel
- A simple loop invoked as data-parallel operation

```c
for (i = 0; i < nitems; i++)
    do_something(i);

status = clEnqueueNDRangeKernel (commandQueue,
                                 do_something_kernel,...,&nitems,...);

__kernel void do_something_kernel(...) {
    int i = get_global_id(0);
    ...
}
```
Pitfalls in GPU programming

- Data-parallel processing
- Only compute-intensive operations
- No complex decision-making operations
- Less synchronization of memory accesses as possible
DBMS operations in GPU

RDBMS operations for GPU can be

- Re-written from scratch
  - High performance
  - complex re-working required
  - hard to parallelize

- Split into granular operations and combined
  - competes with optimal
  - easy to parallelize
  - Functionally isolated (always provide same sematical result)
Granular Operations for GPU

Based on Functional Programming

- map
- reduce
- sort
- hash and etc.,

An SQL statement is represented as a DFG of these smaller operations

```sql
SELECT ( a * b )
FROM table1
WHERE a < 50;
```
RDBMS processing in GPU

GPUs are used to accelerate processing queries of

- Relational operations [Bakkum and Skadron, 2010, Diamos et al., 2012]
- Online aggregation [Lauer et al., 2011]
- Compression [Andrzejewski and Wrembel, 2010, Fang et al., 2010]
- Scans [Beier et al., 2012]

As well as for accelerating query optimization

Selection in GPU

Traditional selection looks like

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

However, branching is not optimal in GPU!

- Concurrent writes might corrupt data
- Locking might serialize threads reducing performance

Pre-computing write locations
Prefix Scan - Synchronization Primitive

Important building block of parallel programs
Applies binary operator to an input array
Commonly used: prefix sum
Parallel Selection

Three steps

- Map - evaluate the filter condition and return 0|1 flag
- Prefix-sum - compute write location for results
- Scatter - store the results on the location
Joins

Three-step scheme

- Count no. of join partners/thread
- Compute prefix-sum using result size of each thread
  - host allocates result space
- threads write their results into the corresponding write location
Joins

Three-step scheme

- Count no. of join partners/thread
- Compute prefix-sum using result size of each thread
  - host allocates result space
- threads write their results into the corresponding write location

Lock-free execution, but with initialization delays!
Open Research Questions

- How to integrate GPU into mainstream RDBMS?
- How to optimize for heterogeneous platform environment?
- Which parts of database engine has to hardware-sensitive and which parts can be hardware-oblivious?
- Is it feasible to add GPU-acceleration directly into an existing system? or heavy re-work in the system leads to developing a new DBMS from scratch?

[Breß et al., 2013]
Conclusion - Third part

- GPU support massive parallel processing
- Data has to be transferred to GPU memory - bottleneck
- Complex operations provide low throughputs
- Results have to be also synchronized
  - Using prefix sum
- RDBMS operations executed by chaining granular primitives
Invitation

You are invited to join our research on GPU-accelerated data management, e.g., in form of:

- Master thesis
- Scientific team project
- Scientific individual project

Contact: david.broneske@ovgu.de; bala.gurumurthy@ovgu.de
Thank you for your attention!

Are there any questions or suggestions?


Bakkum, P. and Skadron, K. (2010). **Accelerating SQL database operations on a GPU with CUDA.** In GPGPU, pages 94 - 103. ACM.


Beier, F., Kilias, T., and Sattler, K.-U. (2012). **GiST Scan Acceleration using Coprocessors.** In DaMoN, pages 63 - 69. ACM.

