New Hardware Architectures for Data Management

Cagri Balkesen, Louis Woods, and Jens Teubner
ETH Zurich, Systems Group
{firstname.lastname}@inf.ethz.ch

March 12, 2013
1965: “Moore’s Law”: \cite{Moore1965}  
- Number of transistors/chip doubles every two years.  
  → Microarchitecture → 40% faster (Pollack’s rule)

1974: “Dennard Scaling”: \cite{Dennard1974}  
- Reduced CMOS gate length:  
  → faster switching (higher frequency)  
  → reduced supply voltage and capacity  
  → power/area remains constant!

→ Performance doubles every two years “at not cost.”
Dennard scaling is reaching its limits.

- Supply Voltage ↓ → Threshold Voltage ↓
- Threshold Voltage ↓ → Leakage Current ↑
- Leakage Current ↑ → power consumption ↑

All modern chip designs are power-limited!
Constrained by Power

Moore’s Law still prevails.

→ More and more transistors to spend.
→ But how (without exceeding the energy budget)?
Design Space

1. Parallelism
   → Lower clock, better energy efficiency

2. Locality
   → Moving data costs a lot of energy

3. Heterogeneous Hardware
   → Specialized hardware orders of magnitude more energy efficient
   ~ Dark silicon [Esmaeilzadeh et al. 2013]

Today:
1. Join Processing on Multi-Cores
2. Graphics Processors (GPUs)
3. Field-Programmable Gate Arrays (FPGAs)
Part II

Multi-Core Architectures
Key Challenges

Key challenges:
- memory wall
- parallelism
  - task-level parallelism (SMT, multi-core)
  - data-level parallelism (SIMD)

Today:
- in-memory joins on modern multi-core machines
Approach 1: Sort (and Merge)

✓ Can be done as external sort
✓ $O(N \log N)$
Approach 2: Hash

\[ R \xrightarrow{\text{scan}} h \xrightarrow{\text{probe}} S \]

\[ \text{hash table} \]

\[ b_1 \quad b_2 \quad \cdots \quad b_k \]

\( \mathcal{O}(N) \) (approx.)

✓ Easy to parallelize
Modern Hardware?
Parallel Hash Join

Parallel Hash Join ("no partitioning" join of [Blanas et al. 2011])

- Protect using locks; **very low contention**
Random access pattern

→ Every hash table access a cache miss

Cost per tuple (build phase):

- 34 assembly instructions
- 1.5 cache misses
- 3.3 TLB misses

hash join is severely latency-bound
Thus: **partitioned hash join** [Shatdal *et al.* 1994]

![Diagram illustrating partitioned hash join]

1. **Partition**
2. **Build**
3. **Probe**

(parallelism: assign partitions to threads → no locking needed)
Build/probe now contained within caches:
- 15/21 instructions per tuple (build/probe)
- \( \approx 0.01 \) cache misses per tuple
- almost no TLB misses

Partitioning is now critical
- Many partitions, far apart
- Each one will reside on its own page
- Run out of TLB entries (100–500)
Cost of Partitioning

for all input tuples $t$ do
  $h \leftarrow \text{hash}(t.key)$
  $\text{out}[\text{pos}[h]] \leftarrow t$
  $\text{pos}[h] \leftarrow \text{pos}[h] + 1$
end for

→ Expensive beyond $\approx 2^8-2^9$ partitions.
Multi-pass partitioning ("radix partitioning")

One hash table per partition

\[
\begin{align*}
R \rightarrow h_{1,1} & \rightarrow h_{1,2} & \rightarrow \cdots & \rightarrow h_2 \\
& \vdots & & \vdots \\
r_1 \rightarrow h_2 & \rightarrow \cdots & \vdots & \leftarrow s_1 \\
r_2 & \vdots & \vdots & \leftarrow s_2 \\
r_3 & \vdots & \vdots & \leftarrow s_3 \\
r_4 \rightarrow h_2 & \rightarrow \cdots & h_2 & \leftarrow s_4 \\
& \vdots & & h_2 \\
& \vdots & & \vdots \\
& \vdots & & \vdots \\
S \rightarrow h_{1,1} & \rightarrow h_{1,2} & \rightarrow \cdots & \rightarrow h_2 \\
& \vdots & & \vdots \\
& \vdots & & \vdots \\
& \vdots & & \vdots 
\end{align*}
\]

Pass 1
- Partition

Pass 2
- Build
- Probe
- Partition

\[© 2013 Cagri Balkesen, Louis Woods, and Jens Teubner · New Hardware Architectures for Data Management\]
Two-pass partitioning

Throughput [million tuples/sec] vs. radix bits

- Single-pass partitioning
- Two-pass partitioning

© 2013 Cagri Balkesen, Louis Woods, and Jens Teubner · New Hardware Architectures for Data Management
Hash join is $O(N \log N)$!
for all input tuples \( t \) do
  \( h \leftarrow \text{hash}(t.\text{key}) \)
  copy \( t \) to \( \text{out}[\text{pos}[h]] \)
  \( \text{pos}[h] \leftarrow \text{pos}[h] + 1 \)
end for

Naïve partitioning (cf. slide 16)

Software-Managed Buffers

→ TLB miss only every \( \text{bufsiz} \) tuples
→ Choose \( \text{bufsiz} \) to match cache line size
Software-Managed Buffers

throughput [million tuples/sec]

radix bits

single-pass partitioning
two-pass partitioning
sw-managed buffers

© 2013 Cagri Balkesen, Louis Woods, and Jens Teubner · New Hardware Architectures for Data Management
Plugging it Together

977 MiB \times 977 MiB

e.g., Nehalem: 25 cy/tpl \approx 90 million tuples per second
A Word on “Scalability”

[Blanas et al. 2011]
[Balkesen et al. 2013]
Sort-Merge Join

Critical part of sort-merge join is sorting.

- Method of choice: **merge sort**
  - two parts: **run generation** and **merging**

→ Both are good candidates for **SIMD acceleration**
Sorting networks

→ branch free, support data parallelism

E.g., network for four elements (“even-odd network”):

\[
\begin{array}{c}
9 & \rightarrow & 3 \\
5 & \rightarrow & 5 \\
3 & \rightarrow & 6 \\
6 & \rightarrow & 9 \\
\end{array}
\]

→ Build larger networks by merging sorted runs.
**SIMD instructions**

**E.g., four words per SIMD register:**

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_3$</th>
<th>$a_4$</th>
<th>xmm0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>$b_2$</td>
<td>$b_3$</td>
<td>$b_4$</td>
<td>xmm1</td>
</tr>
</tbody>
</table>

\[
\text{max}(a_1, b_1) \quad \text{max}(a_2, b_2) \quad \text{max}(a_3, b_3) \quad \text{max}(a_4, b_4)
\]

\[
\text{simd\_max}(\text{xmm0}, \text{xmm1})
\]

Operations **across** registers, **not within**

**But:** Can **shuffle** across and within
Run generation

- 10 min/max, 8 shuffle, 8 load/store
- 64 bytes in, 64 bytes out (128-bit SIMD)
Merging

Two sorted runs, four items each:

- Input: two SIMD registers \( a \) and \( b \), sorted
- 6 min/max, 10 shuffle, 4 load/store

\[
\begin{align*}
& a_1 & & out_1 \\
& a_2 & & out_2 \\
& a_3 & & out_3 \\
& a_4 & & out_4 \\
& b_4 & & out_5 \\
& b_3 & & out_6 \\
& b_2 & & out_7 \\
& b_1 & & out_8 
\end{align*}
\]
1. Load SIMD set from both runs in registers a and b.
2. Perform **SIMD merge** of a and b (→ result in [a, b]).
3. Write a to output.
4. Fetch **next** SIMD set from run where head is smaller; replace a.
5. Goto 2 while there is still input to process.

**E.g.,**

- run 1: 3, 7, 14, 29, 37, 48, 52, 67, 69, 74, 89, 91
- run 2: 9, 11, 16, 21, 25, 39, 46, 71, 79, 86, 88, 95
- output: 3, 7, 9, 11, 14, 16, 21, 25, 29, 37, 39, 46, 48, 52, 67, 69, 71, 74, 79, 86, 88, 89, 91, 95
Sorting and NUMA

input relation

local sort

merge

local merge

local sort

local sort

local sort

local sort
Problem: Merging is **bandwidth-bound**.

→ Merge multiple runs (from NUMA regions) at once
→ Might need **more instructions**, but brings bandwidth and compute **into balance**.
Sorting vs. Hashing

- Throughput [M tuples/sec]

- Hash: 350
- Sort-merge: 300

- "naive": 120
- Radix: 310
- "naive": 110
- SIMD: 240
- m-way: 280

- 12.8 GB x 12.8 GB
- Intel E5-4640 ("Sandy Bridge"), 2.4 GHz, 32 cores

© 2013 Cagri Balkesen, Louis Woods, and Jens Teubner · New Hardware Architectures for Data Management
Part III

Graphics Processors (GPUs)
Graphics Processors (GPUs) ↔ CPUs

**CPU:** Optimize for **instruction latency** (→ control logic and caches)
- Decreasing die share performs actual work (ALUs).

**GPU:** Use chip space to perform work, not for infrastructure
- Simple logic, massive parallelism; optimize for **throughput**.
Parallelism

**CPU: task parallelism**
- heavyweight threads
- 10s of threads, 10s of cores
- threads managed explicitly
- threads run different code

**GPU: data parallelism**
- lightweight threads
- 10,000s of threads, 100s of cores
- scheduled in batches
- all threads run same code
  → SPMD, single program, multiple data
Rationale for high-degree parallelism:

Don’t try to reduce latency, but hide it.

→ While a thread is waiting for memory, execute other threads to hide that latency.

→ Hardware thread scheduling (simple, in-order).

→ Schedule in batches (“warps”) to reduce hardware cost.
Threads are scheduled in units of 32, called **warps**.

- **Warp**: Set of 32 threads that run identical code and start at same program address.
- **SIMT**: Single Instruction Multiple Threads.
- *e.g.*, NVIDIA Kepler: up to $15 \times 64$ warps $= 30$ k threads
- Scoreboard tracks which warps are ready to execute.
**SPMD / SIMT Processing**

- **All** threads in one warp execute the **same** instruction.
- At each time step scheduler selects warp ready to execute (i.e., all its data are available).
- Scheduling decided at **instruction level**.
- NVIDIA Fermi: dual issue; Kepler: quad issue.

**Branch divergence**
Warps and Latency Hiding

Some runtime characteristics (CUDA 1.3):

- Issuing a warp instruction takes 4 cycles.
- Register write-read latency: 24 cycles.
- Global (off-chip) memory access: \( \approx 400 \) cycles.

Threads are executed in-order.

→ Hide latencies by executing other warps when one is paused.
→ Need enough warps to fully hide latency.

E.g.,

- Need \( 24/4 = 6 \) warps to hide register dependency latency.
- Need \( 400/4 = 100 \) instructions to hide memory access latency. If every 8th instruction is a memory access, \( 100/8 \approx 13 \) warps would be enough.
NVIDIA Kepler Architecture
NVIDIA Kepler:
- 15 SMX per chip
- 192 “cores” per SMX (≡ ALU; integer and single-precision float)
- 64 double-precision units
- 32 “special function units” (sine, cosine, etc.)
- issue four warps, two instructions per warp

source: NVIDIA Kepler GK110 White Paper
Host system and \textbf{co-processor} (GPU is only one possible co-processor.)

- Host triggers
  - data copying host $\leftrightarrow$ co-processor,
  - invocations of \textbf{compute kernels}.

- Host interface: \textbf{command queue}.
A traditional loop

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

becomes a **data parallel kernel invocation** in OpenCL (→ **map**):

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

__kernel void do_something_kernel(...) {
    int i = get_global_id(0);
    ...
}
```
OpenCL defines a **C99-like** language for compute kernels.

- Compiled **at runtime** to particular core type.
- Additional set of built-in functions:
  - Context (e.g., `get_global_id()`) , math routines, ...

```c
__kernel void square (__global float *in, 
                      __global float *out)
{
    int i = get_global_id(0);
    out[i] = in[i] * in[i];
}
```

- Very limited **thread interaction** (eases parallel execution)
OpenCL Memory Model

compute device

private memory

work item 1

work item 2

compute unit 1

private memory

private memory

local memory

global memory

compute unit 2

private memory

private memory

work item 1

work item 2

local memory

host

host memory

© 2013 Cagri Balkesen, Louis Woods, and Jens Teubner · New Hardware Architectures for Data Management
Part IV

Field-Programmable Gate Arrays
Field-Programmable Gate Arrays (FPGAs)

- Array of logic gates
- Functionality fully programmable
- Re-programmable after deployment (“in the field”)

- Technology already invented in the 80s
- Today’s chip sizes allow designs of serious complexity
- Projected FPGA revenue in 2013: USD 3.5 billion
Reconfigurable Hardware

Configuration Layer:
- Configuration, stored in SRAM.

Logic Layer:
- Actual hardware logic (LUTs and flip-flops)

→ Reconfiguration ≡ SRAM update
Hardware Circuits

Electronic circuits consist of three fundamental ingredients:

- combinational logic (gates)
- memory elements
- wiring (interconnect)
Reprogrammable Logic: Lookup Tables

### 4-input LUT

- **Input:** $in_0$, $in_1$, $in_2$, $in_3$
- **Output:** out

### Table: Input vs. Output

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

© 2013 Cagri Balkesen, Louis Woods, and Jens Teubner · New Hardware Architectures for Data Management
Elementary Logic Unit (Slice)
Basic FPGA Architecture

- Chip layout: 2D array
- Components
  - **CLB**: Configurable Logic Block (collection of slices)
  - **IOB**: Input/Output Block
  - **DCM**: Digital Clock Manager
- Interconnect Network
  - Signal lines
  - Configurable switch boxes
Configurable Wires (Interconnect)

programmable Switch Box and bundle of lines

programmable intersection point

programmable switch with memory cell

SRAM cell
Programming FPGAs

- FPGA reconfiguration ≡ SRAM update
- Generate new SRAM content (as a “bitstream”) using design tools.
- Input: high-level circuit description
- Typically: using a hardware description language (HDL)
  - Verilog
  - VHDL

```
HDL
foo.vhd

design tools

bitstream
foo.bit

upload to FPGA
```
architecture Behavioral of compare is begin
  process (A, B)
  begin
    if ( A = B ) then
      C <= '1';
    else
      C <= '0';
    end if;
  end process;
end Behavioral;

This is not a sequential program!
FPGA Design Flow

HDL code → synthesis → translate/map → place & route → bitstream

constraints

device-independent netlist (RTL) → device-specific netlist → allocation of individual LUTs, paths,…
FPGA Design Cost

Notes:

- The FPGA design flow is **heavily compute-intensive**
  - Think of minutes, even hours
  - Cost increases dramatically with design size
  - Full circuit re-compilation is something you’ll want to do **off-line** only

- **Device reconfiguration** is faster
  - After all, it’s changing a few bits in SRAM only
  - Think of milli-seconds (however, current hardware is not optimized for fast re-configuration)
Circuit Simulation

Circuits can be simulated in software:

- cycle-accurate simulation
- at any design stage (“behavioral” vs. “post-routing” simulation)

In practice, you rarely need a physical device even
What To Use FPGAs For

FPGAs are good at:

- massive throughput
  - leverage high pin count

- data flow-style processing
  - data “flows through chip,” flows and tasks map naturally to wires and components

- meeting tight performance guarantees
  - Often, the performance of a circuit is fully predictable.
  - important, e.g., for real-time tasks

- regular expressions, state machines
  - FPGA $\equiv$ generic hardware state machine
What NOT To Use FPGAs For

FPGAs are **not so good at:**

- **floating point operations**
  - floating point requires lots of chip space
  - Use a GPU if you really need floating point.

- **branching and runtime flexibility**
  - low clock speed makes runtime decisions rather slow

- likewise: **complex and long algorithms**
  - If you need a full-fledged instruction set processor, use an instruction set processor.
Use Case: Sorting with FPGAs

- Sorting Networks (Revisited) [Mueller et al. 2012]
  - High-throughput sorting for small working sets
  - Data parallelism
  - Pipeline parallelism

- FIFO-based Merge Sort [Koch et al. 2010]
  - Using embedded RAM blocks for larger problems

- External Large Problem Sorting [Koch et al. 2010]
  - Resorting to DRAM for even larger problems
The *compare-and-swap* element is the basic building block of hardware sorting networks. It consists of a comparator circuit and two wide multiplexers.

![Diagram of compare-and-swap element](image-url)
Even-odd Sorting Network

- Sorts eight values using 19 compare-and-swap elements.

```
5 1 1 5 3 8 7 4 6
8 2 2 3 3 4 5 6 7
3 1 1 5 8 7 2 4 3
1 3 8 7 5 4 6 7 8
2 2 3 4 5 6 7 6 7
7 6 4 7 5 7 6 7 8
4 6 6 7 8 7 8 7 8
6 7 8
```
Pipeline Parallelism

- Longest signal path via six compare-and-swap elements

\[ x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 \rightarrow y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7 \]

- Pipelined version \( f_{clk} = 267 \text{ MHz}, 8 \times 32 \text{ bit} \rightarrow 8.5 \text{ GB/s} \)
Sorting Larger Working Sets

- BRAM = fast embedded RAM blocks (~ 4 KB)
- Programmable size and word width
- Dual-ported
- Can be configured as FIFO queues

CLB  BRAM  DSP unit
FIFO-based Merge Sorter

sorted runs

Select-Value Component

merged run
Cascade of FIFO-based Merge Sorters

- Processing at each stage can start once first FIFO is filled
- Only one FIFO is read per cycle at each stage
- BRAM-based FIFOs allow simultaneous reading and writing
  - one FIFO should be enough
  - need to be able to read from different positions in FIFO
  - when done right → streaming is possible
  - problem size ∼ 40K 64-bit keys → 2 GB/s [D. Koch et. al.]
- What if problem size exceeds BRAM capacity?
- For larger problems we can resort to external memory
- Merge sorter tree using the same *select-value component*

© 2013 Cagri Balkesen, Louis Woods, and Jens Teubner · New Hardware Architectures for Data Management
Partial Reconfiguration

- logic frame
- BRAM frame
- DSP frame

- partially reconfigurable region A
- partially reconfigurable region B

- internal configuration access port

- partial bitstream A1
- partial bitstream A2
- partial bitstream B1
- partial bitstream B2

© 2013 Cagri Balkesen, Louis Woods, and Jens Teubner · New Hardware Architectures for Data Management
Max. reconfiguration speed = 400 MB/s
Reconfiguration data (here) = 3 MB
Reconfiguration cost ≡ sorting 15 MB (2 GB/s)
Trade-off: larger problems favor dynamic reconfiguration!
Part V

Summary
Summary

Hardware technology is hitting limits.

- **Frequency scaling** halted years ago.
- **Multi-Core scaling** not sustainable either (power!)

Specialize to further benefit from Moore’s Law:

- Leverage **parallelism** and **locality**.
- Hardware/software co-design

Moore’s Law?

- **Might** slow down for **economic reasons** (but not yet).
Summary

Today:

1. Modern Multi-Core Systems
   - Leverage parallelism (SIMD, multi-core)
   - Preserve locality (cache awareness, NUMA)

2. Graphics Processors (GPUs)
   - Throughput instead of instruction latency
   - Restricted form of parallelism \(\sim\) locality

3. Field-Programmable Gate Arrays (FPGAs)
   - Tailor-made hardware, re-configure at runtime
   - Low frequency (\(\sim\) low power); high bandwidth
Interested in these topics?

- I’m hiring **PhD students**
- Contact me:
  
  Jens Teubner, jens.teubner@cs.tu-dortmund.de
Cagri Balkesen, Jens Teubner, and Gustavo Alonso. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In Proc. of the 29th IEEE Conf. on Data Engineering (ICDE), Brisbane, Australia, April 2013.


René Müller, Jens Teubner, and Gustavo Alonso. Sorting networks on FPGAs.