Building Workload Optimized Solutions for Business Analytics

René Müller, IBM Research – Almaden
muellerr@us.ibm.com

GPU Hash Joins with Tim Kaldewey, John McPherson

BLU work with Vijayshankar Raman, Ronald Barber, Ippokratis Pandis, Richard Sidle, Guy Lohman
IBM Research—Almaden
Spectrum of Workload Optimized Solutions

Specialized Software on General-Purpose Hardware

BLU for DB2

IBM PureData System for Analytics
IBM Netezza 100

IBM Netezza 100-1
IBM Netezza 100 S-Blade
Agenda

- Workload: Business Intelligence, OLAP
- BLU Acceleration for DB2
- Study 1: Acceleration of BLU-style Predicate Evaluation
  - SIMD CPU vs. FPGA vs. GPU
- Study 2: Hash Joins on GPUs
A data warehousing query in multiple languages

- **English**: Show me the annual development of revenue from US sales of US products for the last 5 years by city
A data warehousing query in multiple languages

- **English**: Show me the **annual** development of **revenue** from **US sales** of **US products** for the last **5 years** by **city**

- **SQL**:

  ```sql
  SELECT c.city, s.city, d.year, SUM(lo.revenue)
  FROM lineorder lo, customer c, supplier s, date d
  WHERE lo.custkey = c.custkey
  AND lo.suppkey = s.suppkey
  AND lo.orderdate = d.datekey
  AND c.nation = 'UNITED STATES'
  AND s.nation = 'UNITED STATES'
  AND d.year >= 1998 AND d.year <= 2012
  GROUP BY c.city, s.city, d.year
  ORDER BY d.year asc, revenue desc;
  ```
### Query:
```
SELECT c.city, s.city, d.year, SUM(lo.revenue) FROM lineorder lo, customer c, supplier s, date d
WHERE lo.custkey = c.custkey AND lo.suppkey = s.suppkey AND lo.orderdate = d.datekey AND c.nation = 'UNITED STATES' AND s.nation = 'UNITED STATES' AND d.year >= 1998 AND d.year <= 2012
GROUP BY c.city, s.city, d.year ORDER BY d.year asc, revenue desc;
```
Workload Optimized Systems

My Definition:

*Workload Optimized System.*
A system whose architecture and design have been designed for a *specific* and *narrow* range of applications.

- Optimized for Performance (for us)
- Primary Performance Metrics
  - Response time Seconds \(\rightarrow\) minimize
  - Query throughput Queries/hour \(\rightarrow\) maximize
- Derived Metrics
  - Performance/Watt Queries/hour/Watt \((\text{Queries/Wh})\)
  - Performance/$ Queries/hour/$
- Cost of Ownership itself usually secondary goal
BLU Acceleration for DB2

Specialized Software on General-Purpose Hardware
What is DB2 with BLU Acceleration?

• **Novel Engine for analytic queries**
  – Columnar storage, single copy of the data
  – New run-time engine with SIMD processing
  – Deep multi-core optimizations and cache-aware memory management
  – “Active compression” - unique encoding for storage reduction and run-time processing without decompression

  “Revolution by Evolution”
  – Built directly into the DB2 kernel
  – BLU tables can coexist with row tables
  – Query any combination of BLU or row data
  – Memory-optimized (not “in-memory”)

• **Value : Order-of-magnitude benefits in ...**
  – Performance
  – Storage savings
  – Simplicity!

• **Available since June 2013 in DB2 v10.5 LUW**
BLU’s Columnar Store Engine

- Reduce I/O by only reading the columns that are referenced by the query.
- Traditional row stores read pages with complete rows.

- Columns are horizontally divided into strides of rows
- BLU keeps a synopsis (=summary) of min/max values in each stride
  
  ```sql
  SELECT SUM(revenue)
  FROM lineorder
  WHERE quantity BETWEEN 1 AND 10
  ```
- Skip strides without matches
  → further reduces I/O
  → Skip-sequential access pattern
Order-preserving frequency encoding

- Most frequent values are given the shortest codes
- Column values are partitioned based on code length
- Values of same code length are stored bit-aligned
- Example: encoding of state names

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>California</td>
<td>0</td>
</tr>
<tr>
<td>New York</td>
<td>1</td>
</tr>
<tr>
<td>Florida</td>
<td>0 0</td>
</tr>
<tr>
<td>Illinois</td>
<td>0 1</td>
</tr>
<tr>
<td>Michigan</td>
<td>1 0</td>
</tr>
<tr>
<td>Texas</td>
<td>1 1</td>
</tr>
<tr>
<td>Alaska</td>
<td>1 0 0 0 1</td>
</tr>
</tbody>
</table>

Column: StateName

Dict. 0  Dict. 1  Dict. 2

Page

Region 0

Region 1

Tuple Map
Software and Hardware SIMD

**Software SIMD**
- Process multiple tuples in parallel inside a machine word
- Bit-manipulation Operations using carry, borrow, mask and shift operations.
- Exploits Instruction-level Parallelism (ILP)
- Exploits Specialized Instructions
  - BPERMD on POWER Architecture
  - PEXT (BMI2) on Intel® Haswell™

**Hardware SIMD**
- Process >1 machine word at once
- SSE 4.2 on Intel (Nehalem or later)
- VMX on POWER (P7 or later)

---

"It was amazing to see the faster query times compared to the performance results with our row-organized tables. The performance of four of our queries improved by over 100-fold! The best outcome was a query that finished 137x faster by using BLU Acceleration."

- Kent Collins, Database Solutions Architect, BNSF Railway
BLU with Specialized Hardware

FPGA vs. GPU vs. BLU on Multicore SIMD CPU
What portions to accelerate?

Where does time go?

```
SELECT c.city, s.city, d.year, SUM(lo.revenue)
FROM lineorder lo, customer c, supplier s, date d
WHERE lo.custkey = c.custkey
    AND lo.suppkey = s.suppkey
    AND lo.orderdate = d.datekey
    AND c.nation = 'UNITED STATES'
    AND s.nation = 'UNITED STATES'
    AND d.year >= 1998 AND d.year <= 2012
GROUP BY c.city, s.city, d.year
ORDER BY d.year asc, revenue desc;
```

Example:

- Star Schema Benchmark Query 3.2
- No I/O (= warm buffer pool)
  All columns in memory

- Which is the heavy hitter?
- What is the acceleration potential?
Watch out for Amdahl’s Law

- What speedup is achievable in the best case when accelerating Hashing?
- Assume theoretic ideal scenario
  Hashing 21% → 0%
  (Processing time → 0, zero-cost transfers)

Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - 0.21} \approx 1.27
\]
- Even in ideal case speedup is only 27%
- Small speedup: Just buy faster CPU, e.g., Intel® Xeon® E5-2650 v2 (2.6 GHz) → E5-2667 v2 (3.3 GHz)
- HW accelerators become interesting for speedups > half order of magnitude
BLU-like Predicate Evaluation

Can FPGAs or GPUs do better than HW/SW SIMD code?
(a) What is the inherent algorithmic complexity?
(b) What is the end-to-end performance including data transfers?
CPU Multi-Threaded on 4 Socket Systems: Intel X7560 vs P7

IBM x Series x3850 (Intel®, 32 cores)

IBM p750 (P7, 32 cores)

*) constant data size, vary tuplet width
(Equality) Predicate Core on FPGA

- tuplet_width
- input_word
- predicate

Predicate Evaluator

- out_valid
- bitvector

Word with N tuplets

Bit-vector N bits per input word

Instantiate comparators for all tuplet widths
Test setup on chip for area and performance w/o data transfers

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Core

Predicate Co
Setup on Test Chip

Now, instantiate as many as possible...

1-bit reduction tree

slower output (read) clock
Implementation on Xilinx Virtex-7 485T

<table>
<thead>
<tr>
<th>#cores/chip</th>
<th>max. clock (*)</th>
<th>chip utilization</th>
<th>estimated power</th>
<th>agg. throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>388 MHz</td>
<td>0.3%</td>
<td>0.3 W</td>
<td>1.5 GiB/s</td>
</tr>
<tr>
<td>16</td>
<td>264 MHz</td>
<td>5%</td>
<td>0.6 W</td>
<td>16 GiB/s</td>
</tr>
<tr>
<td>64</td>
<td>250 MHz</td>
<td>23%</td>
<td>1.9 W</td>
<td>60 GiB/s</td>
</tr>
<tr>
<td>100</td>
<td>250 MHz</td>
<td>30%</td>
<td>2.7 W</td>
<td>93 GiB/s</td>
</tr>
<tr>
<td>256</td>
<td>113 MHz</td>
<td>67%</td>
<td>6.6 W</td>
<td>108 GiB/s</td>
</tr>
</tbody>
</table>

GTX Transceiver cap ~65 GiB/s on ingest.

→ Throughput is independent of tuplet width, by design.

*) Given by max. delay on longest signal path in post-P&R timing analysis.
GPU-based Implementation

Data Parallelism in CUDA
CUDA Implementation

**GTX TITAN ($1000)**
- 14 Streaming Multiprocessors (SMX)
- 192 SIMT cores (SPs) / SM
- Shared Memory: 48kB per SMX
- Device Memory: 6 GB (off-chip)
1 GPU Thread per Word and Update Device Memory

Example Grid: 4 blocks, 4 threads/block, 16-bit tuplets → 4 tuplets/bank

atomicOr() to Device Memory
Using fast atomics Device Memory
Atomics in the Kepler Architecture

Bit Vector in Device Memory
Device Memory to Device Memory: Ignoring PCI Express Transfers

**Graph:**
- **Y-axis:** Bandwidth [GiB/s]
- **X-axis:** Tuplet Width [bits]
- **Legend:**
  - GPU w/o transfers
  - p750 (VMX+bpermd, 128 threads)
- **Speedup = 1.61**

**Results:**
- GPU w/o transfers shows varying bandwidth across different tuplet widths.
- p750 (VMX+bpermd, 128 threads) also shows varying bandwidth but at a higher level compared to GPU w/o transfers.
- The speedup observed is 1.61, indicating a significant improvement in performance when ignoring PCI Express transfers.
With Transfers: Zero-Copy access to/from Host Memory

PCI Express becomes the bottleneck at ~11 GiB/s.
Never underestimate optimized code on a general-purpose CPU

ILP and reg-reg operations, sufficient memory bandwidth

BLU Predicate Evaluation

GPU w/ transfers

p750 (VMX+bpermd, 128 threads)

256 cores on FPGA

100 cores on FPGA

FPGA agg. GTX Bandwidth

1 core on FPGA
Hash Joins on GPUs
Taking advantage of fast device memory
Hash Joins

- Primary data access patterns:
  - Scan the input table(s) for HT creation and probe
  - Compare and swap when inserting data into HT
  - Random read when probing the HT

- Data (memory) access on

<table>
<thead>
<tr>
<th></th>
<th>GPU (GTX580)</th>
<th>CPU (i7-2600)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak memory bandwidth</td>
<td>179 GB/s</td>
<td>21 GB/s</td>
</tr>
<tr>
<td>[spec] 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak memory bandwidth</td>
<td>153 GB/s</td>
<td>18 GB/s</td>
</tr>
<tr>
<td>[measured] 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random access</td>
<td>6.6 GB/s</td>
<td>0.8 GB/s</td>
</tr>
<tr>
<td>[measured] 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare and swap</td>
<td>4.6 GB/s</td>
<td>0.4 GB/s</td>
</tr>
<tr>
<td>[measured] 3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Upper bound for:

- Probe
- Build HT

(1) Nvidia: $192.4 \times 10^6$ B/s $\approx 179.2$ GB/s
(2) 64-bit accesses over 1 GB of device memory
(3) 64-bit compare-and-swap to random locations over 1 GB device memory
Computing Hash Functions on GTX580 – No Reads

32-bit keys, 32-bit hashes

<table>
<thead>
<tr>
<th>Hash Function/Key Ingest GB/s</th>
<th>Seq keys+Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>338</td>
</tr>
<tr>
<td>Fowler-Noll-Vo 1a</td>
<td>129</td>
</tr>
<tr>
<td>Jenkins Lookup3</td>
<td>79</td>
</tr>
<tr>
<td>Murmur3</td>
<td>111</td>
</tr>
<tr>
<td>One-at-a-time</td>
<td>85</td>
</tr>
<tr>
<td>CRC32</td>
<td>78</td>
</tr>
<tr>
<td>MD5</td>
<td>4.5</td>
</tr>
<tr>
<td>SHA1</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Cryptographic message digests

- Threads generate sequential keys
- Hashes are XOR-summed locally
Hash Table Probe: Keys and Values from/to Device Memory
32-bit hashes, 32-bit values, 1 GB hash table on device memory (load factor = 0.33)

<table>
<thead>
<tr>
<th>Hash Function/Key Ingest GB/s</th>
<th>Seq keys+ Hash</th>
<th>HT Probe Keys: dev</th>
<th>Values: dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>338</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>Fowler-Noll-Vo 1a</td>
<td>129</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>Jenkins Lookup3</td>
<td>79</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>Murmur3</td>
<td>111</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>One-at-a-time</td>
<td>85</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>CRC32</td>
<td>78</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>MD5</td>
<td>4.5</td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td>SHA1</td>
<td>0.81</td>
<td>0.6</td>
<td></td>
</tr>
</tbody>
</table>

- Keys are read from device memory
- 20% of the probed keys find match in hash table
- Values are written back to device memory
## Probe with Result Cache: Keys and Values from/to Host Memory

32-bit hashes, 32-bit values, 1 GB hash table on device memory (load factor = 0.33)

<table>
<thead>
<tr>
<th>Hash Function/Key Ingest GB/s</th>
<th>Seq keys+Hash</th>
<th>HT Probe Keys: dev Values: dev</th>
<th>HT Probe Keys: host Values: host</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>338</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>Fowler-Noll-Vo 1a</td>
<td>129</td>
<td>2.5</td>
<td>2.4</td>
</tr>
<tr>
<td>Jenkins Lookup3</td>
<td>79</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>Murmur3</td>
<td>111</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>One-at-a-time</td>
<td>85</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>CRC32</td>
<td>78</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>MD5</td>
<td>4.5</td>
<td>1.8</td>
<td>1.8</td>
</tr>
<tr>
<td>SHA1</td>
<td>0.81</td>
<td>0.6</td>
<td>0.6</td>
</tr>
</tbody>
</table>

- Keys are read from **host memory (zero-copy access)**
- 20% of the probed keys find match in hash table
- Individual values are written back to buffer in shared memory and then coalesced to **host memory (zero-copy access)**
## End-to-end comparison of Hash Table Probe: GPU vs. CPU

32-bit hashes, 32-bit values, 1 GB hash table (load factor = 0.33)

<table>
<thead>
<tr>
<th>Hash Function/Key Ingest GB/s</th>
<th>GTX580 keys: host values: host</th>
<th>i7-2600 4 cores 8 threads</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>2.3</td>
<td>0.48</td>
<td>4.8×</td>
</tr>
<tr>
<td>Fowler-Noll-Vo 1a</td>
<td>2.4</td>
<td>0.47</td>
<td>5.1×</td>
</tr>
<tr>
<td>Jenkins Lookup3</td>
<td>2.3</td>
<td>0.46</td>
<td>5.0×</td>
</tr>
<tr>
<td>Murmur3</td>
<td>2.3</td>
<td>0.46</td>
<td>5.0×</td>
</tr>
<tr>
<td>One-at-a-time</td>
<td>2.3</td>
<td>0.43</td>
<td>5.3×</td>
</tr>
<tr>
<td>CRC32</td>
<td>2.3</td>
<td>0.48¹</td>
<td>4.8×</td>
</tr>
<tr>
<td>MD5</td>
<td>1.8</td>
<td>0.11</td>
<td>16×</td>
</tr>
<tr>
<td>SHA1</td>
<td>0.6</td>
<td>0.06</td>
<td>10×</td>
</tr>
</tbody>
</table>

- Result cache used in both implementations
- GPU: keys from host memory, values back to host memory
- CPU: software prefetching instructions for hash table loads

¹ Use of CRC32 instruction in SSE 4.2
Processing hundreds of Gigabytes in seconds

- Combining GPUs fast storage.
- How about reading the input tables on the fly from flash?

Storage solution delivering data at GPU join speed (>5.7 GB/s):
- 3x 900 GB IBM Texas Memory Systems RamSan-70 SSDs
- IBM Global Parallel File System (GPFS)

DEMO: At IBM Information on Demand 2012 and SIGMOD 2013
Summary and Lessons Learned

- Accelerators are not necessarily faster than well-tuned CPU code
- Don’t underestimate the compute power and the aggregate memory bandwidth of general-purpose multi-socket systems.
- Don’t forget Amdahl’s Law, it will bite you.

- Offload larger portions: Hashing only $\rightarrow$ Offload complete hash table

- Take advantage of platform-specific advantages:
  – FPGA: customized data paths, pipeline parallelism
  – GPU: fast device memory, latency hiding through large SMT degree
  – CPU: OO architecture, SIMD and caches are extremely fast if used correctly