SQL/XML-IMDBg

GPU boosted
In-Memory Database
for
ultra fast data management

Harald Frick CEO
Future computing systems are parallel, but

- Programmers have to code the parallelism
- Lake of development systems and experience
- Radical departure from general programming wisdom
- Requires combining CPU and GPU code
- Requires “close to metal” knowledge for max. performance
- The Hardware is evolving and changing rapidly

How are we going to program such systems?
Database systems now are ubiquitous, but

- The core DB technology is from the 70s!
- Database systems now hitting a performance-wall
- Computer memory grows faster than enterprise data
- Hardware is evolving rapidly and gets more heterogeneous
- Parallel computing is a necessity for higher performance

How are we going to develop such systems?
The Database has been re-invented

Complete re-engineering of the database architecture is required to make a DB kernel multi-core ready and scalable to 100’s of processors

Cost effective upgrade path

SQL/XML-IMDBg

Unlimited scalability

Declarative parallel data management and processing

SQL/XML-IMDBg: GPU boosted In-Memory Database
Bridging the technology gap with SQL/XML-IMDBg

Domain Experts can use a standard interface (SQL) for massive parallel computation and data mining.

Cost effective

Unlimited scalability

Universal Data management

www.quilogic.com
Agenda (1)

• Overview about general database architecture

• Short introduction to SQL/XML-IMDB

• Overview of the re-engineering tasks performed to make the IMDB database kernel many-core ready

□ Three levels of re-engineering:

1. General database architecture
2. Relational algebra structures and functions
3. Coding level
Agenda (2)

- **Architecture of the database kernel**
  Explains the overall design principle chosen to make a database kernel ready for massive parallel processing and GPU ready

- **Database Table structure and outline**
  Explains the vertical partitioning layout and why this is one of the most important pre-requisites for successful GPU co-processing in database kernels

- **Memory management**
  The memory management subsystem architecture and how to manage memory on a GPU device with missing dynamic memory management facilities

- **Query Optimizer**
  Gives insights into the Split-Work architecture and explains the Optimizer architecture with a special focus on the dynamic re-optimizing steps performed during query processing

- **Query Executer structure**
  Explains the architecture of the query Executer with a special focus on processing the split-work plan for distributing the query work between CPU and GPU
Summary of presentation:

- Lessons learned from re-engineering the database kernel
- Some important conclusions drawn from our experiences regarding parallelizing and re-engineering a complex application for massive parallel platforms like GPU’s
- Outlook on future enhancements planned for the IMDB
- Questions
SQL/XML-IMDb overview

- DB Library (DLL, LIB, .NET Assembly)
- Application embedded SQL/XQuery statements
- Functional API
- Declarative universal data management
- Kind of declarative STL library
- Data exchange between process boundaries
- 8 years of market experience
- WIN32, WIN64, Linux32, Linux64, ...
SQL/XML-IMDB application practice

```c
Function1()
{
    db.ExecXql ( FOR $X ... RETURN $X)
}
//----------------------------------------

Function2()
{
    db.ExecSql ( SELECT ... FROM )
    while( db.Next() ) { ... }
}
```

---

```c
Sub Command1_OnClick
    db.ExecXql ( FOR $X ... INSERT Y )
    db.ExecSql ( SELECT ... FROM )
```

---

```c
Sub Command2_OnClick
    db.ExecXql ( LET $X ... )
    while db.Next() <> 0 Do
```

---

C++ Application in Process A

VB Application in Process B

Shared Memory

Flash Memory

C++ Application in Process A

VB Application in Process B

Local Memory

Local Memory

Export/Import (persistence)

SQL/XML Tables in disk based files
**Vision: Integrating the GPU memory**

- **GPU Co-Processor + Memory Tables**
- **Function 1()**
  ```c
  db.ExecXql ( FOR $X ... RETURN $X )
  
  //-------------------------------
  
  Function 2()  
  ```
  ```c
  db.ExecSql ( SELECT ... FROM )  
  while( db.Next() ) { ... }  
  ```
- **Sub Command 1_OnClick**
  ```c
  db.ExecXql ( FOR $X ... INSERT Y )
  
  db.ExecSql ( SELECT ... FROM )
  
  db.ExecXql ( LET $X ... )
  
  while db.Next() <> 0 Do  
  ```
- **Sub Command 2_OnClick**
  ```c
  ```

**SQL/XML Tables in disk based files**

- **Export/Import (persistence)**
- **Shared Memory**
- **Local Memory**

---

**Share Memory**

**Function1()**
```c
{  
  db.ExecXql ( FOR $X ... RETURN $X )
}

//-------------------------------

Function2()  
```
```c
{  
  db.ExecSql ( SELECT ... FROM )  
  while( db.Next() ) { ... }  
}
```
Vision: Seamless utilization of GPU power

- Automatically distribute SQL query work and relational data between standard CPU cores and high performance graphics GPU’s
- In-Memory: CPU-Local Memory, CPU-Shared Memory, GPU-Memory

**SQL/XML-IMDBg**

Place tables (and indexes) where you want:

Create Table `Local TR` (...)
Create Table `Shared TS` (...)
Create Table `GPU TG` (...)

Select * From TR, TS, TG WHERE ....
Select `TG.a` * `TG.b` / `TG.c` From TR, TS, TG WHERE ....
Re-inventing the Database by complete re-engineering of the “old” IMDB
Re-Engineering (overall)

• Vertical partitioned table structure
• Arrays instead of Lists, Hash-Tables, Trees ...
• Compression (dictionary based)
• Regular shaped data structures
• Only one Index structure
• Vector processing of Algebra functions
• Fork-Join model (OpenMP)
Re-Engineering (coding)

- Reduced program code complexity
  - Instruction cache friendlier
  - Reduced memory footprint
  - From C++ back to C

- Arrays everywhere
  - Supports memory prefetch
  - Simple access functions
  - Loop unrolling
  - Simpler buffer management

- Dynamic, array like SQL data tables
  - No space waste (no static pre-allocation)
  - Strings represented by ID’s whenever possible
  - Simple access/scan functions

- Simplified index structures
  - Array like
  - Better parallelizable
  - Co-Processor friendly
3 essentials for ultra high performance

- **RISC like DB core structure**
  - Simple and repeating data structures
  - Dynamic Arrays everywhere
  - Dramatically reduced DB-kernel complexity
  - Favors parallel algorithmic structures!

- **Split-Work Optimizer**
  - Dynamic mid-query re-optimizing
  - Schedules query execution between CPU and GPU

- **Split-Work Query Executer**
  - Work duplication and parallel execution
  - Materialized intermediate results (compressed bitmaps)
  - Vector processing of relational algebra
Agenda (2)

- **General architecture outline of database kernel**
  Explains the overall design principle chosen to make a database kernel ready for massive parallel processing and GPU ready

- **Database Table structure and outline**
  Explains the vertical partitioning layout and why this is one of the most important pre-requisites for successful GPU co-processing in database kernels

- **Memory management**
  The memory management subsystem architecture and how to manage memory on a GPU device with missing dynamic memory management facilities

- **Query Optimizer**
  Gives insights into the Split-Work architecture and explains the Optimizer architecture with a special focus on the dynamic re-optimizing steps performed during query processing

- **Query Executer structure**
  Explains the architecture of the query Executer with a special focus on processing the split-work plan for distributing the query work between CPU and GPU
Regular Shaped Structures
and
Dynamic Arrays
everywhere
Very good DB / GPU structure fit
Agenda (2)

- **General architecture outline of database kernel**
  Explains the overall design principle chosen to make a database kernel ready for massive parallel processing and GPU ready

- **Database Table structure and outline**
  Explains the vertical partitioning layout and why this is one of the most important pre-requisites for successful GPU co-processing in database kernels

- **Memory management**
  The memory management subsystem architecture and how to manage memory on a GPU device with missing dynamic memory management facilities

- **Query Optimizer**
  Gives insights into the Split-Work architecture and explains the Optimizer architecture with a special focus on the dynamic re-optimizing steps performed during query processing

- **Query Executer structure**
  Explains the architecture of the query Executer with a special focus on processing the split-work plan for distributing the query work between CPU and GPU
Table Layout (1)

Vertical decomposition - dense data storage

Array like columns allow:
- Coalesced access on GPU
- CPU cache friendly access
- Distributed table layout

Example:

**Bool Data Type**

<table>
<thead>
<tr>
<th>Row ID's</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

**Int Data Type**

<table>
<thead>
<tr>
<th>Row ID's</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

**Double Data Type**

<table>
<thead>
<tr>
<th>Row ID's</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

**Ref Data Type**

<table>
<thead>
<tr>
<th>Row ID's</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

**Uint Data Type**

<table>
<thead>
<tr>
<th>Row ID's</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

<table>
<thead>
<tr>
<th>Row ID's</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

**32**

This is a string
Big Problem

How to manage huge sized arrays
Table Layout (2)

Break array in manageable chunks

Table Descriptor

- **Header**
  - Column Count
  - Flags
  - Row IDs

- **Column 1**
  - Data Type
  - Char size
  - Col name
  - Flags
  - Data Addr

- **Column N**
  - Data Type
  - Char size
  - Col name
  - Flags
  - Data Addr

0 1024 2048 3072 4096 6020

Contains a pointer in case of variable sized data

Contains a pointer in case of variable sized data

Row IDs

fixed number of rows

dynamic

fixed

Table Structure

Table Layout (2)

Break array in manageable chunks

Table Descriptor

- **Header**
  - Column Count
  - Flags
  - Row IDs

- **Column 1**
  - Data Type
  - Char size
  - Col name
  - Flags
  - Data Addr

- **Column N**
  - Data Type
  - Char size
  - Col name
  - Flags
  - Data Addr

0 1024 2048 3072 4096 6020

Contains a pointer in case of variable sized data

Contains a pointer in case of variable sized data

Row IDs

fixed number of rows

dynamic

fixed

Table Structure
Agenda (2)

• General architecture outline of database kernel
  Explains the overall design principle chosen to make a database kernel ready for massive parallel processing and GPU ready

• Database Table structure and outline
  Explains the vertical partitioning layout and why this is one of the most important pre-requisites for successful GPU co-processing in database kernels

• Memory management
  The memory management subsystem architecture and how to manage memory on a GPU device with missing dynamic memory management facilities

• Query Optimizer
  Gives insights into the Split-Work architecture and explains the Optimizer architecture with a special focus on the dynamic re-optimizing steps performed during query processing

• Query Executer structure
  Explains the architecture of the query Executer with a special focus on processing the split-work plan for distributing the query work between CPU and GPU
Memory management (CPU)

Win-OS

VirtualAlloc

TLFS Allocator (Two-Level Segregate Fit)

- Bitmap based chunk management
- Simple code structure (instruction cache!)
- **Memory pools** based!
- Fastest allocator for shared memory
- Good cache locality
- Small size

Source:
Implementation of a constant-time dynamic storage allocator.
Memory management (GPU)

Win-OS

VirtualAlloc()

cudaMallocHost()

for data transfers

cudaMalloc(large chunk)

GPU

Data transfer mem-POOL

GPU Memory pool

CUDA Memory pool

GPU device memory

*devPtr = *hostPtr

cudaMemcpy

Host memory
Table management on CPU and GPU

Coalesced memory access

CPU

GPU

POOL 1

POOL n
Scales to multi GPU clusters

Some problems do not fit within a single GPU memory

Memory POOL 1
Split-Work Executer

Memory POOL n
Split-Work Executer

CPU

GPU
GPU / CPU Table split

Create Table TR(…)
Create Table Shared TS(…)
Create Table GPU TG(…)

Table split between CPU and GPU memory

Table split is done automatically depending on column data type: varchar, char, blob, …
Index Layout

Search procedure:
1.) Candidate search
2.) Exact search

Searching can be:
1.) Binary search
2.) Scan
3.) Position estimate
Index processing on GPU

Search procedure:
1.) Candidate search
2.) Exact search

- Binary search
- Simple scan
- Estimate * (skip over - use statistics of value distribution)

Index Descriptor

Table Row IDs …

41 768 2048 5017

GPU Kernel call

Table values …

Row ID

CPU

GPU

Thread Processors

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM

PBSM
Agenda (2)

- General architecture outline of database kernel
  Explains the overall design principle chosen to make a database kernel ready for massive parallel processing and GPU ready

- Database Table structure and outline
  Explains the vertical partitioning layout and why this is one of the most important pre-requisites for successful GPU co-processing in database kernels

- Memory management
  The memory management subsystem architecture and how to manage memory on a GPU device with missing dynamic memory management facilities

- Query Optimizer
  Gives insights into the Split-Work architecture and explains the Optimizer architecture with a special focus on the dynamic re-optimizing steps performed during query processing

- Query Executer structure
  Explains the architecture of the query Executer with a special focus on processing the split-work plan for distributing the query work between CPU and GPU
Query Optimizer and Executer

Re-optimizing steps performed during query execution

Scanning stage is parallel

Barrier sync.

Selected Row ID's as compressed bitmap

Re-optimize

Join stages are parallel

Re-optimize

QuiLogic In-Memory DB Technology

SQL/XML-IMDbg: GPU boosted In-Memory Database
Why is re-optimizing important

True size of intermediate result sets!
- Split-Work requires re-examination of intermediate results
- Schedules the query tasks either to CPU or GPU

- Should sorting be done on GPU or CPU?
- Leave intermediate results on GPU or move over?
- Perform join processing on GPU or CPU?
- Is the table persistent on GPU or on CPU?
- ...

Tight integration of optimizer and executer necessary
Agenda (2)

- **General architecture outline of database kernel**
  Explains the overall design principle chosen to make a database kernel ready for massive parallel processing and GPU ready

- **Database Table structure and outline**
  Explains the vertical partitioning layout and why this is one of the most important pre-requisites for successful GPU co-processing in database kernels

- **Memory management**
  The memory management subsystem architecture and how to manage memory on a GPU device with missing dynamic memory management facilities

- **Query Optimizer**
  Gives insights into the Split-Work architecture and explains the Optimizer architecture with a special focus on the dynamic re-optimizing steps performed during query processing

- **Query Executer structure**
  Explains the architecture of the query Executer with a special focus on processing the split-work plan for distributing the query work between CPU and GPU
Query Executer - basic principle

Operator Tree

Join

SCAN

40  T1.Col2

Operator Library

Query Compiler + Optimizer

ScanINT(*start)
{
  val = start[1]
  *p = start[2]
  *bitmap = start[3]
  #pragma omp parallel
  for(....)
}

ScanINT_GPU(*start)
{
  ....
  // call GPU kernel....
  ScanInt <<<dimGrid, dimBlock>>> (...) ....
}

Worker Thread

Thread Pool

Stop

Pre-compiled operators
QuiLogic In-Memory DB Technology

SQL/XML-IMDBg: GPU boosted In-Memory Database

Executer

Operator Library

- ScanINT(void *start)
- ScanUINT(void *start)
- ScanFLOAT(void *start)
- ScanDBL(void *start)
- ScanIntIDX(void *start)
- AddINT(void *start)
- AddDBL(void *start)

Vector processing of algebra functions

- Compiler optimization
- Filled cache lines
- Memory prefetch support
- No instruction cache misses
- Loop unrolling
- Branch prediction

4 cores

Re-optimize

Function call

ScanINT_GPU

GPU Kernel call

- Coalesced memory access
- High bandwidth
- Massive parallel scanning
- Fast sorting
- Arithmetic processing
- Data local on GPU
- Loop unrolling

240 cores
Split-Work scheduler scales to multi GPUs

Query work scales linear with GPU count

n copies of (partial) program array

CPU thread 1

GPU kernel call

CPU thread 2

GPU kernel call

CPU thread n

GPU kernel call
SQL query processing

Select age, name, (salary * tax) From T Where age > 18 and salary > 10,000 and ...

Task distribution to maximize performance!
Lessons learned

➢ Try to keep things simple and regular

- Classic Database technology **to complex** for GPU architecture
- Complete re-engineering required
- Regular structures favor parallelize and scalability
- Index and algorithmic structures needs to be revised
- Use regular shaped data structures for fast iterative access
- Re-optimizing is important for best query work scheduling
- Coalesced memory access required for high performance
- Keep the GPU bussy
- OpenMP is a good starting point
- Object oriented coding style is useless for GPU co-processing
Future plans

- GPU accelerated projection stage of SQL query is "work in progress"
  - SELECT Tg.a * Tg.b + (1.323 * Tg.c) FROM ....

- LINQ integration
  - (Language INtegrated Query for Microsoft .NET-Framework)

- GPU based XML processing

- Self tuning (Experiment mode)

- Embedded devices?
Conclusions

- The RISC like „regular-shaped“ DB technology scales to hundreds of parallel processor cores

  - GPU usage as DB co-processor is a valid concept and boosts performance orders of magnitude

  - Declarative programming style for application development is easier, faster and …
    - You get the GPU power for free!

  - QuiLogic makes GPU power available for domain experts in a simple and declarative way (SQL)
Questions?

harald.frick@chello.at
office@quilogic.com