Query Compilation of Dataflow Programs for Heterogeneous Platforms

Felix Beier & Kai-Uwe Sattler
DBIS@TU Ilmenau
www.tu-ilmenau.de/dbis
Outline

1. Requirements of Engineering Applications
2. Dataflow Programming in PipeFlow
3. Query Compilation
4. Outlook
Motivation

• Data-intensive engineering applications
  – Improved sensing technology
  – Complex models
  – „Smart“ devices

• Relying on database technology?
  – Not really, but Matlab & friends
Nanopositioning Machine

- Find defects in electronic components
- 3D surface measurement (x,y position, z measure)
- Resolution: nm (100s MB - 10s TBs per scan)
- Static data CAD circuit models (if available)
Nanopositioning Machine
Online Source Localization

- Identification of neurophysiologically active areas
- N EEG/MEG sensors, data rate: 600-5000 Hz
- Static brain model with K - M vertices, brain atlas
- Complex analysis pipeline: signal filtering, decomposition, matrix inversion, time-based folding
- Indexing for brain model & decomposition dictionary
Particle Simulation

- Blood simulation for medical filters
- N particles of different kinds (blood, water, ...)
- Static scene with dynamic particles
- KNN-queries with index-rebuild in each step
DB Technology Inside?

• Database technology = providing abstractions
  – Data manipulation (query language etc.)
  – Hardware (storage, storage hierarchies, …)
  – Scalability (indexing, parallel processing, MapReduce)

• Simplifies development
  – Code reuse
  – Providing correctness and performance guarantees

But, are these still the right abstractions?
Lessons Learned

• Very large, spatiotemporal data sets
• Large gaps between computer science and engineering contexts
  – Domain-specific languages
  – Tooling ecosystem
• Reinventing the wheel
  – Data management algorithms
  – Specializations for parallel hardware
  – Optimizations
Requirements

• Extensibility by user-defined operations
  – Optimization, parallelization, recovery, …

• Complex data types
  – Spatio-temporal data, time series, matrices, …

• Low latency, online processing
  – Data stream processing, CEP

• Dealing with uncertainty in measurements and models
Dataflow Programming in PipeFlow

Big Dynamic Engineering Data
Dataflow Programming

• Model flow of data between transformation steps
• Inject data management & processing primitives
  – Partitioning & merging steps
  – Parallelization
  – Mapping to (specialized) hardware
  – Distributed workload management at cluster-scale
  – Flow optimizations
  – Fault tolerance
• Keeping framework usable for engineers
  – No new languages / paradigms
  – Reusable programs
  – Integration of domain-specific data types & libraries
Programming Model

Store & Process

Continuous Query Processing

Query Compilation of Dataflow Programs for Heterogeneous Platforms

Publish-Subscribe Pattern

Source  Operator  Sink

typed pipes

Workshop on System Software Support for Big Data

Query Compilation of Dataflow Programs for Heterogeneous Platforms

25.09.2014  Felix Beier  14 / 29

Technische Universität Ilmenau
PipeFlow

Dataflow specification → PipeFlow Compiler

graph checking and rewriting

C++ code generation

embedded in application

standalone process

distributed processes in clusters

PipeFlow

Query Compilation of Dataflow Programs for Heterogeneous Platforms
25.09.2014  Felix Beier  15 / 29
Operator Model

- Encapsulated functionality
  - Computation (possibly stateful)
  - Implementation in domain-specific languages
  - Specialization for hardware platforms (CPU/GPU)
  - Wrapper for library functions

- Meta-information
  - Typed input, output and parameter channels
  - State handling
  - Operator location
  - Profiling information
PipeFlow: Overview

- Dataflow specification language inspired by Pig
- Specification = sequence of operators connected by typed „pipes“
- Large set of predefined operators (sources, joins, aggregation, windows, CEP, …)

```
$pipe1 := operator1(…) params;
$pipe2 := operator2(…) params;
$out := operator3($pipe1, $pipe2, …) params;
```

- Supported by the PipeFabric engine: C++ library of operator templates & utility functions
Query Compilation
PipeFlow: Operators & Expressions

• Type-specific template instantiation for operators

• Expressions are compiled into native code

```cpp
typedef Tuple<int, double, MatrixXd> MyTuple;
auto op = new Filter<MyTuple>([&](MyTuplePtr tp) {
    return std::get<0>(*tp) > 10;
});
```

```plaintext
$o := \text{filter}(\text{in}) \text{ by } i1 > 10;
```
Dataflow Parallelization

- Providing abstraction for data parallelism
  - Partitioning of input data stream: tuple-wise, batch-wise, column-wise
  - Execution environment: threads for multi-core CPU, threads for GPU, distributed processes for compute cluster
- Result merging
- Supporting user-defined operators!

Semantics

![Diagram showing split, merge, and operators](image-url)
Parallelization in PipeFlow

- Make parallelization explicit but hide the implementation details by a parallelize operator.

```plaintext
define calc_statistics ($in) returns $out {
    $x := myOp($in);
    $out := mySecondOp($x) ...;
}

$res := parallelize($in) on slice(x) 
    do calc_statistics 
    using (mode = "thread", partitions = 10);
```
Slice, Split & Merge

- Physical algebra operators
  - **Slice**: split a single tuple or tuple value into multiple instances, i.e. vertical partitioning, vector/matrix decomposition
  - **Scatter**: route tuples to subqueries based on PartitionID
  - **Gather**: collect partial results from parallel subqueries
  - **Merge**: combine partial results, i.e. merge streams, join tuple components or even values, final aggregation

---

Workshop on System Software Support for Big Data

Query Compilation of Dataflow Programs for Heterogeneous Platforms

25.09.2014   Felix Beier   22 / 29
GPU Processing

- GPU: vector processor attached to host system
  - SIMD / SIMT operations
  - Input copy -> compute -> output copy
GPU Processing Example

- Vector addition and scalar multiplication
  - \( R = (V_1 + V_2) \times c \)
- Generate parallel GPU Kernel
- Parallelize for multiple GPUs

\[
\begin{align*}
V_1 & \\
V_2 & \\
c & \\
+ & \\
* & \\
R &
\end{align*}
\]
GPU Processing Example

- Fine granular parallelism: SIMD
- Vectorize on element index
- Mapping to thread ID

```cpp
template< typename E >
__global__ void gpuVecAddMul(c,v1,v2,scatter,gather){
    int t = calculateThreadID(); // consider grid
    E e1 = scatter(v1,t); // read v1[t]
    E e2 = scatter(v2,t); // read v2[t]
    E r = c * (e1 + e2);
    gather(r,t); // write to result[t]
};
```
GPU Processing Example

- Coarse granular parallelism: multi-GPU
- Partition input vectors
- Mapping (disjoint) partitions to each GPU
- Collect & merge (partial) results

```c
// Multi-GPU vector addition and multiplication example

template< typename E >
void multiGpuVecAddMul(c,v1,v2,slice,scatter,proc,gather,merge){
    slices = slice(c,v1,v2); // partition input vectors
    scatter(slices,gpus); // host-to-device copy partitions
    results = in_parallel_do(gpuVecAddMul(...)); // launch kernels
    collected = gather(results); // device-to-host copy
    merge(collected); // combine in result vector and publish
};
```
Query Compilation: Rewriting

- Option 1: determine functions based on datatype + operation + X? automatically
- Option 2: user-provided functions

<table>
<thead>
<tr>
<th>datatype</th>
<th>operation</th>
<th>slice</th>
<th>scatter</th>
<th>gather</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic</td>
<td>filter, projection, ..</td>
<td>stream</td>
<td>hash, key</td>
<td>union</td>
<td>-</td>
</tr>
<tr>
<td>atomic</td>
<td>aggregate</td>
<td>stream</td>
<td>hash, key</td>
<td>union</td>
<td>post-aggregation</td>
</tr>
<tr>
<td>vector, matrix</td>
<td>+, scalar mult.</td>
<td>1-dim decomposition</td>
<td>slice-id</td>
<td>union</td>
<td>compostion</td>
</tr>
<tr>
<td>matrix</td>
<td>advanced</td>
<td>decomposition</td>
<td>slice-id</td>
<td>union</td>
<td>problem-specific</td>
</tr>
</tbody>
</table>
Outlook
What’s next?

• Modules for domain-specific types
  – First-class types, e.g., events, signals, images, matrices, tensors, graphs, …
  – Library integration, e.g., OpenCV, Pregel, …
  – Modeling uncertainty

• Aspect-orientation for injecting data management routines
  – Functional models as monads, arrows
  – Parallelization primitives
  – Automatic & manual partitioning
  – Elastic scaling

• Multi-level optimizations
  – Rule-based graph optimizations
  – Domain-specific optimization rules
  – Machine-specific optimizations for hardware
Discussion
References


(2) http://www.tu-ilmenau.de/cc-npmm/projekte/

(3) http://www.itwissen.info/definition/lexikon/Chip-chip.html


(5) http://people.ee.ethz.ch/~cattin/MIA-ETH/02-IntensityBasedRegistration-media/figs/labelled-brain.png


(7) http://www.nvidia.de/object/tesla_c1060_de.html