Alexandros Koliousis
a.koliousis@imperial.ac.uk

Joint work with Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, Peter Pietzuch and, more recently, George Theodorakis, Panos Garefalakis and Holger Pirk

Large-Scale Data & Systems Group
Department of Computing, Imperial College London
http://lsds.doc.ic.ac.uk
Streams of Data Everywhere

Many new data sources are now available:

- Linear access patterns make data processing a streaming problem.
High-Throughput Low-Latency Analytics

Facebook Insights
9GB of page metrics/s
In less than 10 s

Google Zeitgeist
40K user queries/s
Within ms

Feedzai
40K card trans/s
In 25 ms

NovaSparks
150M stock options/s
In less than 1 ms

$\text{window}$

$t+1$

$t$
Algorithmic Complexity Increases

- Topic-based filtering
- Content-based filtering
- Complex pattern matching
- Stream queries
- Online machine learning, data mining

- Publish/Subscribe
- Complex Event Processing (CEP)
- Stream processing
Design Space for Data-Intensive Systems

Tension between performance & algorithmic complexity

- Easy for most algorithms
- Hard for machine learning algorithms

Result latency:
- 10s
- 1s
- 100ms
- 10ms
- 1ms

Data amount:
- TBs
- GBs
- MBs
Scale Out in Data Centres
Task vs Data Parallelism

Task parallelism:
Multiple data processing jobs

Data parallelism:
Single data processing job

Input data

Servers in data centre

Results
Distributed Dataflow Systems

Idea: Execute data-parallel tasks on cluster nodes

Tasks organised as dataflow graph

Almost all big data systems do this:
- Apache Hadoop, Apache Spark, Apache Storm, Apache Flink, Google TensorFlow, ...
“Nobody Ever Got Fired for Using a Hadoop Cluster” [HotCDP’12]

Or Flink or Spark ;)

2012 study of MapReduce workloads

- Microsoft: median job size < 14 GB
- Yahoo: median job size < 12.5 GB
- Facebook: 90% of jobs < 100 GB

The size of the workloads has changed, but so has the size/price of memory!

Many data-intensive jobs easily fit into memory

It’s expensive to scale-out in terms of hardware and engineering!

In many cases a single server is cheaper/more efficient than a cluster
Exploit Single-Node Heterogeneous Hardware

Servers with CPUs and GPUs now common
  - 10x higher linear memory access throughput
  - Limited data transfer throughput

Use both CPU & GPU resources for stream processing
CQL: **SQL-based declarative language** for continuous queries [Arasu et al., VLDBJ’06]

Credit card fraud detection example:

- Find attempts to use same card in different regions within 5-min window

```sql
select distinct W.cid
from Payments [range 300 seconds] as W,
Payments [partition-by 1 row] as L
where W.cid = L.cid and W.region != L.region
```

CQL offers correct window semantics
Challenges & Contributions

1. How to parallelise sliding-window queries across CPU and GPU?
   Decouple query semantics from system parameters

2. When to use CPU or GPU for a CQL operator?
   Hybrid processing: offload tasks to both CPU and GPU

3. How to reduce GPU data movement costs?
   Amortise data movement delays with deep pipelining

Details omitted
Problem: Window semantics affect system throughput and latency

– Pick task size based on window size?

Window-based parallelism results in redundant computation
How to Parallelise Window Computation?

Problem: Window semantics affect system throughput and latency

- Pick task size based on window size? On window slide?

Slide-based parallelism limits GPU parallelism

Compose window results from partial results
Avoid coupling throughput/latency of queries to window definition

- e.g. **Spark** imposes lower bound on window slide:

![Graph showing throughput vs. window slide](image)

- Window slide limited by min. latency (~500 ms)
- Micro-batch size limited by window slide
Idea: Decouple task size from window size/slide
   - Pick based on underlying hardware features
     • e.g. PCIe throughput

   - Task contains one or more window fragments
     • E.g. closing/pending/opening windows in $T_2$
Idea: Decouple task size from window size/slide
  – Assemble window fragment results
  – Output them in correct order

Worker A: $T_1$

Worker B: $T_2$

Worker B stores $T_2$ results, merges window fragment results and forwards complete windows downstream
Operator Implementations / API

Fragment function, $f_f$
Processes window fragments

Assembly function, $f_a$
Merges partial window results

Batch function, $f_b$
Composes fragment functions within a task
Allows incremental processing

- Size: 7 rows
- Slide: 2 rows
- 5 tuples/sec
How to Pick the Task Size?

![Graph showing CPU and GPU throughput vs task size](image)

- **Throughput (GB/s)**
  - CPU
  - GPU

- **Task Size (KB)**
  - 32
  - 64
  - 128
  - 256
  - 512
  - 1024
  - 2048
  - 4096
How Does Window Slide Affect Performance?

Performance of window-based queries remains predictable

Aggregation \( \text{avg} \) \([\text{rows } 1024, \text{ slide } x]\)
Challenges & Contributions

1. How to parallelise sliding-window queries across CPU and GPU?
   Decouple query semantics from system parameters

2. When to use CPU or GPU for a CQL operator?
   Hybrid processing: offload tasks to both CPU and GPU

3. How to reduce GPU data movement costs?
   Amortise data movement delays with deep pipelining
Idea: Enable tasks to run on both processors
   – Scheduler assigns tasks to idle processors

Past behavior:

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>QA</td>
<td>3 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>QB</td>
<td>3 ms</td>
<td>1 ms</td>
</tr>
</tbody>
</table>

Task Queue:

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10</td>
<td>T9</td>
</tr>
<tr>
<td>T8</td>
<td>T7</td>
</tr>
<tr>
<td>T6</td>
<td>T5</td>
</tr>
<tr>
<td>T4</td>
<td>T3</td>
</tr>
<tr>
<td>T2</td>
<td>T1</td>
</tr>
</tbody>
</table>

First-Come First-Served

FCFS ignores effectiveness of processor for given task
Idea: Idle processor *skips* tasks that could be executed faster by another processor

– Decision based on observed *query task throughput*

<table>
<thead>
<tr>
<th>Past behavior:</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>QA</td>
<td>3 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>QB</td>
<td>3 ms</td>
<td>1 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Queue:</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁₀</td>
</tr>
<tr>
<td>QA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heterogeneous Look-Ahead Scheduler (HLS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU comes first</td>
</tr>
</tbody>
</table>

**HLS**

<table>
<thead>
<tr>
<th>CPU</th>
<th>T₃</th>
<th>T₇</th>
<th>T₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>T₁</td>
<td>T₂</td>
<td>T₄</td>
</tr>
</tbody>
</table>

**HLS fully utilises processors**
The SABER Architecture

1. Dispatching stage
   - Dispatch fixed-size tasks

2. Scheduling & execution stage
   - Dequeue tasks based on HLS

3. Result stage
   - Merge & forward partial window results

Java
15K LOC
C & OpenCL
4K LOC
Evaluation: Set-up & Workloads

- **10 Gbps NIC**
- **Intel Xeon** 2.6 GHz
  - 16 cores
  - 64GB RAM
  - Ubuntu Linux 14.04
- **NVIDIA Quadro K5200**
  - 2,304 cores
  - 8GB RAM
  - NVIDIA driver 346.47

- **Google Cluster Data**
  - 144M jobs events from Google infrastructure

- **SmartGrid Measurements**
  - 974M plug measurements from houses

- **Linear Road Benchmark**
  - 11M car positions and speed on highway
Is Hybrid Stream Processing Effective?

Different queries result in different CPU:GPU processing split that is hard to predict offline.

- `select`
- `group-by_avg`
- `aggr_avg`
- `group-by_avg`
- `group-by_cnt`
- `group-by_avg`
- `select`
- `group-by_cnt`

Throughput (10^6 tuples/s)

- **SABER (CPU contrib.)**
  - Intel Xeon 2.6 GHz
  - 16 cores

- **SABER (GPU contrib.)**
  - NVIDIA Quadro K5200
  - 2,304 cores
Is Hybrid Stream Processing Effective?

Aggregate throughput of CPU and GPU always higher than its counterparts

- GPU is faster
- CPU is faster
- Not additive due to queue contention

Throughput (GB/s)

- Aggregation
- Group-by
- θ-join

SABER (CPU only)
SABER (GPU only)
SABER
What is the CPU/GPU Trade-Off?

Hybrid processing model benefits from GPU ability to process complex predicates fast.

Throughput (GB/s)

- **SABER (CPU only)**
- **SABER (GPU only)**
- **SABER**

![Graph showing throughput vs. number of selection and join predicates]

- **# selection predicates**
  - [rows 1024, slide 1024]

- **# join predicates**
  - [rows 1024, slide 1024]
Is Heterogeneous Look-Ahead Scheduling Effective?

\[ W_1 \] benefits from static scheduling but HLS fully utilises GPU:
- GPU also runs \( \sim 1 \) of of \( \gamma \) tasks

\[ W_2 \]

Throughput (GB/s)

<table>
<thead>
<tr>
<th></th>
<th>FCFS</th>
<th>Static</th>
<th>HLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi )</td>
<td>3.5</td>
<td>3.5</td>
<td>5.5</td>
</tr>
<tr>
<td>( \gamma )</td>
<td>2.0</td>
<td>2.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

\( [\text{rows 1024, slide 512}] \)

\( [\text{rows 1024, slide 1024}] \)
HLS periodically uses idle, non-preferred processor to run tasks to update query task throughput.

Example: higher selectivity, more predicates evaluated, GPU is preferred.
To begin with, can SABER compete with popular distributed stream processing systems?

https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing
Enter Yahoo! Stream Benchmark

An industry standard (wannabe)
Storm, Flink, Spark, Apex, Drizzle, Diff. Dataflow

Tumbling-window query, bottlenecked by factors other than computation

How many times a campaign has been seen in a tumbling window
Systems Compared

Apache Flink (1.3.2)

Apache Spark Streaming (2.4.0)

SABER (1.0), without GPU support

StreamBox: a single-server system with emphasis on out-of-order processing
Experimental Setup

6 servers (1 master and 5 slaves): 2 Intel Xeon E5-2660 v3 2.60 GHz CPUs
  ○ 20 physical CPU cores
  ○ 25 MB LLC

32 GB of memory

10 GigE connection between the nodes

In-memory generation

8 cores per node
On a Single Server…

Reduced serialization costs; keeping data in LLC
On Multiple Servers...

![Graph showing throughput vs. number of threads and nodes for different platforms like Flink, SABER, Spark, and StreamBox.](image-url)
On Multiple Servers…

Flink outperforms Spark!

64 millions/sec with 6 cores!
### Throughput (million tuples/sec)

<table>
<thead>
<tr>
<th></th>
<th>Spark</th>
<th>Flink</th>
<th>SABER</th>
<th>Handwritten C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>2</td>
<td>4.8</td>
<td>11.8</td>
<td>39</td>
</tr>
</tbody>
</table>

**Pipeline Strategy** [Hyper, VLDB’11]:
- keep data in CPU registers
- as many sequential operations as possible per tuple
- maximize data locality

Do better than LLC? 

With a compiler-based approach to generate custom code based on a set of hardware-specific optimisations for any given query
H/W-Efficient Streaming Operators

Hammer Slide: Work- and CPU-efficient Streaming Window Aggregation [ADMS’18]

- Incremental computation for both invertible and non-invertible functions
- Parallel processing within a slide (>1) with SIMD instructions
- Bridge the gap between sliding and tumbling window computation
HammerSlide + SABER
Window processing model
Decouples query semantics from system parameters

Hybrid stream processing model
Can achieve aggregate throughput of heterogeneous processors

Hybrid Look-ahead Scheduling (HLS)
Allows use of both CPU and GPU opportunistically for arbitrary workloads

Thank you! Any Questions?
Alexandros Kolioussis
github.com/lsds/saber