Operator Variant Selection on Heterogeneous Hardware

Viktor Rosenfeld, Max Heimel, Christoph Viebig, Volker Markl

ADMS '15 in conjunction with VLDB '41
Kohala Coast, Hawai'i – August 31, 2015
Challenge
High performance or small implementations

Experimental demonstration
Lack of performance portability in OpenCL

Solution sketch
Performance-portable database operators

Current work
Learning fast operator implementations
Challenge

High performance or small implementations
Hardware-sensitive Approach

dedicated operator implementations for each device

pro: optimal implementations for each device
contra: development and maintenance overhead
Hardware-oblivious Approach

support multiple devices from a single implementation

pro: smaller code base
pro: support for unknown devices
contra: lost optimization opportunities

OpenCL vendor driver / compiler

c_single operator implementation

CPUs
GPUs
Xeon Phi
...
OpenCL Portability

- OpenCL offers *functional portability*
- But not *performance portability*
- Many parameters to tweak: thread workload, memory access, special functions, ... 
- Hardware-specific OpenCL implementations?

lack of performance portability limits the value of functional portability
Experimental demonstration

Lack of performance portability in OpenCL
Selection Kernels

Basic algorithm

• scan over column

• evaluate predicate for each value, $x < \text{const}$

• return bitmap indicating satisfying values
Variant Dimensions

Code modifications  ~ 60 variants

- Basic algorithm (memory access & result bitmap construction): sequential, atomic-global, atomic-local, reduce, collect, transpose
- Result bitmap granularity: 8 bit, 16 bit, 32 bit, 64 bit
- Loop unrolling: yes, no
- Predication: yes, no

Workload parameters

- Local size: 1, 2, 4, 8, ..., max
- Elements per thread: 1, 2, 4, 8, ..., 1024

~5000 of selection kernel variants
Competitive Variants

percentage of variants that are at most 2x slower than fastest variant for each device
Competitive Variants

percentage of variants that are at most 2x slower than fastest variant for each device

"easy" and "difficult" devices
Solution sketch

Performance-portable database operators
Automatic Variant Tuning

1. specify operators in generic fashion
2. derive different implementations
3. learn best implementation per device

let the system generate and find the best variant
Current work

Learning fast operator implementations
learn best operator implementation for current workload

Implementation details
- Loop Unrolling
- Branch Elimination
- ...

no optimal operator implementation, even for a single query
Micro Adaptivity With Many Variants

- runtime distribution of 300 queries with 1K chunks
- for each query: select pool of size X from ~5000 different variants
Micro Adaptivity With Many Variants

- runtime distribution of 300 queries with 1K chunks
- for each query: select pool of size X from ~5000 different variants

- 1 variant in the pool
- can be good, bad, or average
Micro Adaptivity With Many Variants

- runtime distribution of 300 queries with 1K chunks
- for each query: select pool of size X from ~5000 different variants

- 16 variants in the pool
- every variant is used (no explore/exploit)
- distribution is compacted
Micro Adaptivity With Many Variants

- runtime distribution of 300 selection queries with 1K chunks
- for each query: select pool of size X from ~5000 different variants

as the number of variants in the pool increases, we converge on the mean runtime of the variant universe
Micro Adaptivity With Many Variants

- runtime distribution of 300 queries with 1K chunks
- for each query: select pool of size X from ~5000 different variants

- small pool sizes
- distribution is shifted to optimal value
- distribution is compacted

- Micro Adaptivity
- Random
- Optimal

Variant Pool Size

Chunk runtime [ms] (log-scaled)
Micro Adaptivity With Many Variants

- large pool sizes
- overhead of explore destroys benefit of Micro Adaptivity

- runtime distribution of 300 queries with 1K chunks
- for each query: select pool of size X from ~5000 different variants
Search Strategies

improve the pool between queries

Greedy
• keep 2 fastest variants
• randomly replace others

Genetic
• keep 2 fastest variants
• replace others by combining attributes from 2 parents currently in pool
• chance of becoming a parent depends on performance
• mutate variants to get out of local minima
Competitive Variants

percentage of variants that are at most 2x slower than fastest variant for each device

"easy" and "difficult" devices
Influence of Search Strategies

Intel Xeon CPU

- Close to optimal performance for "easy" devices

- 100 series of 10 consecutive selection queries
- Working pool: 8 variants chosen randomly at start of series
- Baseline None: no updates of working pool between queries
Influence of Search Strategies

Intel Xeon Phi Accelerator

Average slowdown (compared to optimal)

Query Number

• 100 series of 10 consecutive selection queries
• working pool: 8 variants chosen randomly at start of series
• baseline None: no updates of working pool between queries

room for improvement for "difficult" devices
Summary and Outlook

• OpenCL offers functional portability, but lack of performance portability limits usefulness

• we can use query feedback to learn fast data processing operators

• generate variants automatically (step 1 and 2)

• improve search strategies (micro benchmarks, source code metrics, …)
Learning Framework

- Universe: universe of all possible variants
- Variant Generator: generates new variants
- Search Strategy: updates variants in pool after each query
- Working Pool: pool of variants that are currently explored
- VW-Greedy: picks variant from pool for each vector call
- Runtime: runs selected variant, monitors performance
Processor Characteristics

**fast CPU variants**

- **Sequential**
  - Interleaved-atomic-global and interleaved-atomic-local
  - Interleaved-reduce
  - Interleaved-collect
  - Interleaved-transpose

**fast GPU variants**

- **Interleaved-atomic-global and interleaved-atomic-local**
- **Interleaved-reduce**
- **Interleaved-collect**
- **Interleaved-transpose**

**manufacturer and architecture differences**

<table>
<thead>
<tr>
<th>Variant</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7-870 (Nehalem)</td>
<td>92</td>
</tr>
<tr>
<td>Intel Xeon E5-2650 v2 (Ivy Bridge)</td>
<td>92</td>
</tr>
<tr>
<td>Intel Core i7-4900MQ (Haswell)</td>
<td>92</td>
</tr>
<tr>
<td>AMD Opteron 2356 (Barcelona)</td>
<td>92</td>
</tr>
<tr>
<td>AMD Opteron 6128 HE (Magny-Cours)</td>
<td>92</td>
</tr>
<tr>
<td>IBM S231-E2B (POWER7)</td>
<td>92</td>
</tr>
<tr>
<td>Intel Xeon Phi SE10/7120 (Knights Corner)</td>
<td>92</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 460 (Fermi)</td>
<td>92</td>
</tr>
<tr>
<td>NVIDIA Quadro K2100M (Kepler)</td>
<td>92</td>
</tr>
<tr>
<td>NVIDIA Tesla K40M (Kepler)</td>
<td>92</td>
</tr>
<tr>
<td>AMD Radeon HD 6950 (Terascale3)</td>
<td>92</td>
</tr>
<tr>
<td>Intel Iris 5100 (Gen7.5)</td>
<td>92</td>
</tr>
</tbody>
</table>
Best Variants By Device

<table>
<thead>
<tr>
<th>Device</th>
<th>Variant</th>
<th>Elements per thread</th>
<th>Local size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel CPUs</td>
<td>64-bit sequential PU</td>
<td>device-specific</td>
<td></td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 460</td>
<td>32-bit transpose U</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>NVIDIA Quadro K2100M</td>
<td>16-bit transpose (P/PU)</td>
<td>1/2/4</td>
<td>128</td>
</tr>
<tr>
<td>NVIDIA Tesla K40M</td>
<td>16-bit transpose (U)</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>AMD Radeon HD 6950</td>
<td>8-bit collect (U)</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>Intel Iris 5100</td>
<td>64-bit transpose P/PU</td>
<td>1024</td>
<td>64/128</td>
</tr>
</tbody>
</table>

P: predicated, U: unrolled

The best variant is hardware-specific