Guided Profiling for Auto-Tuning Array Layouts on GPUs

Nicolas Weber, Sandra C. Amend and Michael Goesele
TU Darmstadt
Motivation

• Memory access is one of the most important performance factors in CUDA applications

• CUDA Programming Guide
  • It is one of the three basic optimization strategies to “Optimize memory usage to achieve maximum memory throughput”

• Performance difference up to an order of magnitude between best and worst implementation

• Experience alone does not guarantee to find the optimal configuration
Motivation

• Tedious to optimize in big GPU applications
  • Layouts: Array of Structs, Structure of Arrays, AoSoA
  • Transpositions of multi-dimensional arrays
  • Size of L1 cache / shared memory
  • Memory placement: Global, Texture, Shared, Local and Constant memory

• Changing GPU architectures require to reoptimize
  • Memory hierarchy was changed in every architecture

➢ Automated optimization for most GPUs and algorithm
  • We develop an open source auto-tuner to automatically optimize array access in CUDA applications (with minimal programming overhead)
What is the optimal configuration for a kernel?

- Difficult to find an analytical solution
  - Memory access can be input data sensitive
  - Different optima for varying input data
  - Many GPU architectures with different memory hierarchies

- Empirical profiling
  - Requires to compile & execute many different implementations
  - Very time intensive
High Dimensionality

- Up to several million configurations!

\[
1000000 \cdot \left(\frac{5s \text{ (Compilation time)}}{16 \text{ (Cores)}} + 0.5s \text{ (Execution time)}\right) \geq 9 \text{ days}
\]
Measured Kernel Execution Time

- $A == 0 \&\& B == 0$
- $A != B$
- $A == 1 \&\& B == 1$
Toy Example: Performance Estimation 2D

![Chart showing performance estimates for different memory layouts and AoS/SoA/ AoSoA approaches.]

- **Texture**
  - AoS: 7ms
  - SoA: 5ms
  - AoSoA: 4ms

- **Global**
  - AoS: 5ms
  - SoA: 4ms
  - AoSoA: 7ms
Toy Example: Performance Estimation 2D

- Predicted Execution Time
  - Execution time of $\text{Base} + \text{Sum}(\Delta(\text{Base}, \text{Support Configurations}))$

![Diagram showing performance comparison between different layout types: Texture (7ms), Global (5ms), AoS (5ms), SoA (4ms), AoSoA (6ms). Arrows indicate performance differences: 2ms-1ms for AoS to SoA, 2ms for AoS to AoSoA, 1ms for SoA to AoSoA.]
Non-Linear Relationship

• Not all configurations are linearly related to each other

• Shared dimensions
  • Affect all arrays
  • L1 cache size

• Independent dimensions
  • Only affect one array
  • Layout, memory and transposition
Toy Example: Prediction Domains

L1 cache: L1

L1 cache: EQ

L1 cache: SM
Toy Example: Prediction Domains

L1 cache: EQ

L1 cache: SM

L1 cache: L1
Real Example: Measured Time
Real Example: Base Configurations

The graph shows the configurations ordered by runtime, with measured and base configurations distinguished. The y-axis represents time in milliseconds, and the x-axis represents configurations ordered by runtime.
Real Example: Support Configurations

- Configurations ordered by runtime
- Measured
- Support
- Base
Real Example: Prediction

configurations ordered by runtime

Measured  Predicted  Support  Base

Time (ms)
Real Example: Prediction (zoom in)

configuration ordered by runtime

- Predicted
- Measured
Real Example: Prediction (zoom in)
Real Example: Prediction (zoom in)

Measured: 44 / 5184 (0.85%)
Our result: 72.52ms (+3.59ms)

Min: 68.93ms
Max: 526.48ms
Avg: 300.75ms
Evaluation
1. Benchmark: BitonicSort

```c
struct {
    long a;
    int b;
    short c;
    char d;
}
```

- Sorting for each field, \( A < B < C < D \)
- Values limited to \( 0 \ldots 1023 \) to cause equal columns

- 2 Kernels
- 27 configurations
2. Benchmark: KD-Tree Builder

• 9 Kernels
• > 570k configurations
3. Benchmark: REYES

- 4 Kernels
- > 2.4M configurations
Profiling Algorithms

• Exhaustive Search [Muraladinharan et al. 2014]
  • Tries all possible configurations

• Greedy Profiling [Liu et al. 2008]
  • Optimize each dimension after each other

• Evolutionary Algorithm [Jordan et al. 2012]
  • Starts with a random population of configurations
  • Good configurations are stored
  • Bad configurations are mutated, combined or randomly sampled
Evaluation

• Profiling Algorithms
  • Exhaustive Search (E)
  • Greedy Algorithm (G)
  • Evolutionary Algorithm (A)
  • Our Algorithm (P)

• GPUs
  • GeForce GTX980 (Maxwell)
  • Tesla K20 (Kepler)

• CUDA WatchDog: kills configurations which exceed the execution time of the best found
Execution Speed Up: GTX980 w/o WatchDog

- Bitonic
- KD-Tree
- Reyes
Execution Speed Up: Tesla K20 w/o WatchDog
Execution Speed Up: Tesla K20 with WatchDog
SPEED UP
Profiling Speed Up: BitonicSort

Higher is better

<table>
<thead>
<tr>
<th>Device</th>
<th>E</th>
<th>EW</th>
<th>G</th>
<th>GW</th>
<th>A</th>
<th>P</th>
<th>PW</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX980</td>
<td>1.00</td>
<td>0.96</td>
<td>0.94</td>
<td>0.96</td>
<td>0.96</td>
<td>0.95</td>
<td>0.90</td>
</tr>
<tr>
<td>K20</td>
<td>1.03</td>
<td>0.85</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
</tr>
</tbody>
</table>
Profiling Speed Up: KD-Tree Builder

Higher is better

<table>
<thead>
<tr>
<th></th>
<th>GTX980</th>
<th>K20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>138.6</td>
<td>158.1</td>
</tr>
<tr>
<td></td>
<td>70.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>374.8</td>
<td>387.5</td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>865.2</td>
<td>956.4</td>
</tr>
<tr>
<td></td>
<td>1,327.8</td>
<td>1,588.2</td>
</tr>
</tbody>
</table>

GTX980 | K20
---|---
E | P
EW | PW
G | GW
A |
Summary

• Introduced prediction guided profiling algorithm
  • up to 5.5x faster than other state of the art methods
    • while achieving comparable results
  • up to 9300x faster than exhaustive search
    • 10 days 20 hours $\rightarrow$ 1 minute 40 seconds

• Limitations
  • No global optimization $\rightarrow$ only one kernel at once is optimized
Thank you for your attention!

Source Code available @
http://tinyurl.com/matog
(BSD 3-Clause license)

Contact: matog@gris.tu-darmstadt.de