Application-independent Autotuning for GPUs
Martin Tillmann, Thomas Karcher, Carsten Dachsbacher, Walter F. Tichy
Introduction

- Autotuning is an effective technique for optimizing parallel applications.
- Most work in the autotuning area has concentrated on application specific tuning parameters.
- On graphic processing units (GPUs), there are various tunable parameters that are *application independent*.
  - Number of threads per block
  - Loop unrolling
  - Workload per thread

⇒ Application-independent Autotuning for GPUs
Online Autotuning - AtuneRT

- Our autotuner AtuneRT uses a feedback loop of:
  1. measuring execution time of a program section and
  2. adjusting the tuning parameter configuration.

- A search-based algorithm such as Nelder-Mead guides the process.

- As a general-purpose run-time autotuner, AtuneRT does not use application-specific knowledge about tuning parameters.
AtuneRT - API

AtuneRT is controlled with the following three functions:

- `addParameter(&pararm, range, default)`
- `startMeasurement()`
- `stopMeasurement()`
Modern GPUs offer massive parallelism for a wide range of algorithms.

Optimizing GPUs is hard.
- Align memory access patterns
- Minimize control flow costs
- Balance workloads

Even though GPU compilers got better, most optimization is still done by hand using trial and error.
In our work we employed NVIDIA’s CUDA architecture. CUDA is parallel programming model that organizes threads in a hierarchy.
Autotuning on the GPU - Block Size

- The number of overall threads is dictated by the number of data elements.
- The number of threads per block (block size) is variable.
- Due to limited resources for each block not all threads can be active.

Common approach:
Trial and error hand-optimized code for one GPU generation.
Online autotuning removes the need for hand-tuning and has multiple advantages:

- No knowledge of the GPUs specification is required.
- React to changes at run-time.
- Tune multiple non-independent parameters.
- Fast and easy to use.
Results

Evaluation on three GPUs:

<table>
<thead>
<tr>
<th>Model</th>
<th>Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geforce GTX 680</td>
<td>GK104</td>
</tr>
<tr>
<td>Geforce GTX 470</td>
<td>GF100</td>
</tr>
<tr>
<td>Quadro 6000</td>
<td>GF100GL</td>
</tr>
</tbody>
</table>

We examined the performance of four applications with all possible tuning configurations. We also measured the iterations it took AtuneRT to reach the optimal values and resulting speed-up.
Results - Marching Cubes

Marching Cubes - block size of one kernel - complete parameter space

What is optimal for one GPU can be the worst case for another.
Results - Merge Sort

Merge Sort - block sizes of three kernels - complete parameter space

Even with similar behavior the optimum is different.
Tuning parameters (here: \textbf{degree of loop unrolling} and \textbf{block size}) are not independent of each other. Optimization matters – over 20\% speed-up only through loop unrolling.
Thrust is a C++ template library for CUDA, similar to the Standard Template Library (STL).

Thrust’s algorithms either use hard-coded values or simple heuristics to determine the **block size** of the CUDA kernels.

<table>
<thead>
<tr>
<th>GPU Type</th>
<th>Time (s)</th>
<th>Block Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 680, no tuner</td>
<td>2.566</td>
<td>224</td>
</tr>
<tr>
<td>GTX 680, with tuner</td>
<td>2.223</td>
<td>128</td>
</tr>
<tr>
<td>Quadro 6000, no tuner</td>
<td>10.025</td>
<td>224</td>
</tr>
<tr>
<td>Quadro 6000, with tuner</td>
<td>10.026</td>
<td>224</td>
</tr>
</tbody>
</table>

Tuning the `thrust::inclusive_scan`-function results in a speed-up of 13% on the GTX 680.

Thrust optimizes for older generation GPUs like the Quadro 6000.
Conclusion

- Tuning application independent parameters is important.
- Autotuning is feasible for optimizing GPU applications on multiple platforms.
- Preparing the applications for AtuneRT was easy: Three calls to the tuner sufficed.
- The tuner could be integrated in the kernel API call.
- No knowledge of hardware specifications is required.

We expect autotuning to become an essential part in determining tuning parameters at run-time on the GPU.
References I


Mark Harris et al. “Optimizing parallel reduction in CUDA”. In: *NVIDIA Developer Technology* 2 (2007).


## Results - Merge Sort

<table>
<thead>
<tr>
<th></th>
<th>default time</th>
<th>optimal time</th>
<th>speed-up</th>
<th>autotuning iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadro 6000</td>
<td>0.0483</td>
<td>0.0324</td>
<td>32.81%</td>
<td>17</td>
</tr>
<tr>
<td>GTX 470</td>
<td>0.0457</td>
<td>0.0307</td>
<td>32.85%</td>
<td>20</td>
</tr>
<tr>
<td>GTX 680</td>
<td>0.0328</td>
<td>0.0234</td>
<td>28.64%</td>
<td>20</td>
</tr>
</tbody>
</table>

Execution times in seconds. Speed-up of the optimal configuration relative to the default parameters.
Measuring time on the GPU is not accurate. GPU computation is initiated via driver calls (with internal scheduling).

Time is measured by inserting events in the execution pipeline.

References
Martin Tillmann, Thomas Karcher, Carsten Dachsbacher, Walter F. Tichy – Application-independent Autotuning for GPUs