Better Than All the Rest: Finding Maximum-Performance GPU Kernels Using Auto-Tuning

GPU Technology Conference
April 7, 2016

Cedric Nugteren
How to find the best flight ticket?

Solution:
• Evaluate all combinations manually
• … or use a flight comparison website
How to find the best flight ticket?

Solution:
• Evaluate all combinations manually
• … or use a the CLTune auto-tuner
Example: convolution

Example: blur filter

Targets:
- GPUs (CUDA & OpenCL)
- Multi-core CPUs
- Other OpenCL-capable devices

\[
B_{x,y} = w \cdot \sum_{i=-1}^{i\leq1} \sum_{j=-1}^{j\leq1} F_{i,j} A_{x+i,y+j}
\]

example: 3 by 3 filter
OpenCL 2D convolution

Each thread: one output pixel

Double for-loop

Unroll loops?

Thread coarsening (2D)?

Work-group / thread-block size?

Vector data-types?

Cache in local / shared memory?
Search-space explosion

Large search-space:
• Not feasible to explore manually
• Perhaps not even feasible automatically?

3424 configurations
Why do we need an auto-tuner?

**Large search-space:**
- Not feasible to explore manually
- Perhaps not even feasible automatically?

**Wide variety of devices:**
- Different optimal kernels
- Even from the same vendor

**User-parameter dependent:**
- Examples: matrix sizes, image size, filter sizes, etc.
Why do we need an auto-tuner?

Large search-space:
- Not feasible to explore manually
- Perhaps not even feasible automatically?

Wide variety of devices:
- Different optimal kernels
- Even from the same vendor

User-parameter dependent:
- Examples: matrix sizes, image size, filter sizes, etc.

CLTune: Automatic OpenCL kernel tuning

CLTune is a C++ library which can be used to automatically tune your OpenCL and CUDA kernels. The only thing you’ll need to provide is a tunable kernel and a list of allowed parameters and values.

For example, if you would perform loop unrolling or local memory tiling through a pre-processor define, just remove the pre-processor define before compilation.
CLTune in action

Example: matrix-vector multiplication (y = Ax) with tiling

```c
#include "cltune.h"

void main() {
    constexpr auto kSizeM = 2048;
    constexpr auto kSizeN = 4096;
    std::vector<float> mat_a(kSizeN * kSizeM);
    std::vector<float> vec_x(kSizeN);
    std::vector<float> vec_y(kSizeM);
    cltune::Tuner tuner(0, 1); // platform 0, device 1
    // Adds a kernel which supports tiling by a factor TS
    auto id = tuner.AddKernel("./tiled.opencl", "matvec_tiled", {kSizeM}, {1});
    tuner.AddParameter(id, "TS", {32, 64, 128, 256});
    tuner.MulLocalSize(id, {"TS"});
    // Sets the function's arguments
    tuner.AddParameterScalar(static_cast<int>(kSizeM));
    tuner.AddParameterScalar(static_cast<int>(kSizeN));
    tuner.AddParameterInput(mat_a);
    tuner.AddParameterInput(vec_x);
    tuner.AddParameterOutput(vec_y);
    // Runs the tuner
    tuner.Tune();
    tuner.PrintToScreen();
}
```

Example input data

Kernel to tune

Tile size (TS): 4 possible values

Verifies output based on optional reference kernel
CLTune in action

```c
#include "cltune.h"

void main() {
    constexpr auto kSizeM = 2048;
    constexpr auto kSizeN = 4096;
    std::vector<float> mat_a(kSizeM * kSizeM);
    std::vector<float> vec_x(kSizeN);
    std::vector<float> vec_y(kSizeM);
    cltune::Tuner tuner(0, 1); // platform 0, device 1
    // Adds a kernel which supports tiling by a factor TS
    auto id = tuner.AddKernel("./tiled_opencl.c"), "matvec_tiled", {kSizeM}, {1});
    tuner.AddParameter(id, "TS", {32, 64, 128, 256});
    tuner.MulLocalSize(id, "TS");
    // Sets the function's arguments
    tuner.AddArgumentScalar(static_cast<int>({kSizeM}));
    tuner.AddArgumentScalar(static_cast<int>({kSizeN}));
    tuner.AddArgumentInput(mat_a);
    tuner.AddArgumentInput(vec_x);
    tuner.AddArgumentOutput(vec_y);
    // Runs the tuner
    tuner.Tune();
    tuner.PrintToScreen();
}
```

Tuneable tile-size TS

Handles all the host-device data transfers!
Tuneable tile-size TS

---

```
#include "ctune.h"

int main() {
  constEXPR auto kSizeM = 2048;
  constEXPR auto kSizeN = 4096;
  std::vector<float> mat_a(kSizeN);
  std::vector<float> vec_x(kSizeN);
  std::vector<float> vec_y(kSizeN);
  cltune::Tuner tuner(0, 1); // Adds a kernel which supports auto id = tuner.AddKernel("TS,
  tuner.AddParameter(id, "TS",
  tuner.MullLocalSize(id, {"TS"})
  // Sets the function's arguments
  tuner.AddParameterScalar(stat,
  tuner.AddParameterScalar(stat,
  tuner.AddParameterInput(mat_a)),
  tuner.AddParameterInput(vec_x),
  tuner.AddParameterOutput(vec_y)
  // Runs the tuner
  tuner.Run();
  tuner.PrintToScreen();
}
```

---

```
=== Initializing on platform 0 device 2 ===
Device name: 'AMD Radeon R9 M370X Compute Engine' (OpenCL 1.2)

---

RUN
OK
Completed matvec_reference (11 ms) - 1 out of 1

---

RUN
OK
Completed matvec_tiled (10 ms) - 1 out of 4

---

RUN
OK
Completed matvec_tiled (9 ms) - 2 out of 4

---

RUN
OK
Completed matvec_tiled (8 ms) - 3 out of 4

---

RUN
OK
Completed matvec_tiled (8 ms) - 4 out of 4

---

RESULT
matvec_tiled; 10 ms; TS 32;
RESULT
matvec_tiled; 9 ms; TS 64;
RESULT
matvec_tiled; 8 ms; TS 128;
RESULT
matvec_tiled; 8 ms; TS 256;

---

Printing best result to stdout
BEST: matvec_tiled; 8 ms; TS 128;

---

Printing results to file in JSON format

---

End of the tuning process
```
Option 0: Full search

😊 Finds optimal solution
❌ Explores all options

Search strategies

3424 convolution kernels on Tesla K40m GPU
Search strategies

Option 0: Full search
- Explores arbitrary fraction
- Performance varies

Option 1: Random search
- Performance varies

Example: 107 out of 3424 configurations (1/32th)

Colours: 3 example runs

Performance [% of best-known]

Search progress (steps)
Search strategies

Option 0: Full search
Option 1: Random search
Option 2: Simulated annealing

😊 Explores arbitrary fraction
☒ Performance varies
☒ Meta-parameter
☒ Local optima

Example: 107 out of 3424 configurations (1/32ⁿ)

Colours: 3 example runs
Search strategies

Option 0: Full search
Option 1: Random search
Option 2: Simulated annealing
Option 3: Particle swarm optim.

- ☻ Explores arbitrary fraction
- ☓ Performance varies
- ☓ Meta-parameter
- ☓ Local optima

Example: 107 out of 3424 configurations (1/32ⁿ)

Colours: 3 example runs
Line-types: 3 swarms
Search strategies evaluation

Conclusions:
- PSO performs poorly
- SA perform good, but not much better than random search

Each search: 107 out of 3424 configurations (1/32th)

average best result of 128 searches

meta-parameters for SA and PSO

Search strategies evaluation

Performance [% of best-known]

Device: K40m

Search space

Device: HD7970

Search space

Device: GTX480

Search space

Device: Iris

Search space
Solution: machine learning?
Machine learning an auto-tuner

Evaluate subset of all configurations (e.g. 100 out of 3424)

Train model with the examples

Predict execution time for all other configurations

(take best 10 and evaluate them on actual hardware)

Neural network (3-layer fully connected)

input: parameter configuration
output: execution time

Linear regression

Machine learning an auto-tuner

Trained on a random subset of convolution example (1/32\textsuperscript{th})
Machine learning an auto-tuner

Trained on a random subset of convolution example (1/32th)
GEMM case-study

Conclusions:
- Different best parameters for different devices
- Performance better than clBLAS, but not as good as assembly-tuned cuBLAS
GEMM performance comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>K40m GFLOPS</th>
<th>GTX480 GFLOPS</th>
<th>HD7970 GFLOPS</th>
<th>Iris GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>this work</td>
<td>1197</td>
<td>2380</td>
<td>3144</td>
<td>2409</td>
</tr>
<tr>
<td>clBLAS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cuBLAS</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CLBlas: The tuned OpenCL BLAS library**

CLBlas is a modern, lightweight, performant and tunable OpenCL BLAS library written in C++11. It is designed to leverage the full performance potential of a wide variety of OpenCL devices from different vendors, including desktop and laptop GPUs, embedded GPUs, and other accelerators. CLBlas implements BLAS routines: basic linear algebra subprograms operating on vectors and matrices.

**Conclusions:**

- Different best parameters for different devices
- Performance better than clBLAS, but not as good as assembly-tuned cuBLAS
Convolution case-study

<table>
<thead>
<tr>
<th>filter of size</th>
<th>best parameters for</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3x3</td>
</tr>
<tr>
<td>3x3</td>
<td>100%</td>
</tr>
<tr>
<td>7x7</td>
<td>65%</td>
</tr>
<tr>
<td>11x11</td>
<td>66%</td>
</tr>
</tbody>
</table>

Conclusions:
- Different best parameters for different:
  - devices (see paper)
  - filter-sizes
- Performance equal or better than the state-of-the-art [3]

What about CUDA support?

Written for OpenCL, but…

• Uses the high-level OpenCL API
  CLCudaAPI
• And CLCudaAPI also has a CUDA back-end!
• Switch between OpenCL and CUDA with a single include
  • Uses the CUDA 7.0 driver API and runtime compilation library (nvrtc)
Better Than All the Rest:
Finding Maximum-Performance GPU Kernels Using Auto-Tuning

Auto-tuning GPU kernels:
- Large search-space
- Wide variety of devices
- User-parameter dependent

Advanced search strategies:
- Simulated annealing
- Particle swarm optimisation

Case-studies:
- Fastest 2D convolution
- CLBlast matrix-multiplication library

Machine-learning:
- Train a model on a small subset
- Use the model to predict the remainder

Source-code on GitHub:
https://github.com/CNugteren/CLTune


Slides available @ www.cedricnugteren.nl