Profiling GPU Code
Jeremy Appleyard, February 2016
What is Profiling?

Measuring Performance

- Measuring application performance
  - Usually the aim is to reduce runtime

- Simple profiling:
  - How long does an operation take?

- Advanced profiling:
  - Why does an operation take a long time?
Profiling Workflow

1. Find which parts of the code take time
2. Work out why they take time
3. Optimize
4. GOTO 1.
GPU Performance

Quick overview

• A processor has two key performance limits
  • Floating point throughput (FLOP/s)
    • Peak ~6 TFLOP/s
  • Memory throughput (GB/s)
    • Peak ~300 GB/s (DRAM)

• GPUs also need parallelism
  • This is how they can be so fast
Profiling Tools
General GPU Profiling

From NVIDIA
• nvprof
• NVIDIA Visual profiler
  • Standalone (nvvp)
  • Integrated into Nsight Eclipse Edition (nsight)
• Nsight Visual Studio Edition

Third Party
• Tau Performance System
• VampirTrace
• PAPI CUDA component
In this talk

- We will focus on nvprof and nvvp
- nvprof => NVIDIA Profiler
  - Command line
- nvvp => NVIDIA Visual Profiler
  - GUI based
Case Study
Recurrent Neural Network - LSTM

- Uses:
  - Natural language processing
  - Sequences of images (eg. video)
  - Bio/medical
- We will look at optimisation of a single iteration of LSTM
LSTM

Viewed as a black box

- Inputs and outputs are “batched vectors”.
  - ie. A minibatch
- Typical length is 256-2048
- Typical batch size is 32-128
LSTM Details

\[ h_t = \sigma(\text{mul}(\text{tanh}(\text{mul}(h_{t-1}, W_h) + b_h), b_h + x)) \]

\[ c_t = \text{tanh}(\text{mul}(h_{t-1}, W_c) + b_c + x) \]
LSTM Profile
Using nvprof

```
>> nvprof ./RNN 512 64
==6805== NVPROF is profiling process 6805, command: ./RNN 512 64
==6805== Profiling application: ./RNN 512 64
==6805== Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>88.46%</td>
<td>512.07us</td>
<td>8</td>
<td>64.009us</td>
<td>60.449us</td>
<td>75.618us</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>4.26%</td>
<td>24.673us</td>
<td>8</td>
<td>3.0840us</td>
<td>2.9120us</td>
<td>4.1600us</td>
<td>pw_biasAdd(float*, float*, int, int)</td>
</tr>
<tr>
<td>1.93%</td>
<td>11.200us</td>
<td>5</td>
<td>2.2400us</td>
<td>2.0160us</td>
<td>2.9760us</td>
<td>pw_vecAdd(float*, float*, float*, int)</td>
</tr>
<tr>
<td>1.92%</td>
<td>11.136us</td>
<td>3</td>
<td>3.7120us</td>
<td>3.4560us</td>
<td>4.1920us</td>
<td>[CUDA memcpyDtoD]</td>
</tr>
<tr>
<td>1.39%</td>
<td>8.0650us</td>
<td>3</td>
<td>2.6880us</td>
<td>2.3040us</td>
<td>3.4570us</td>
<td>pw_sigmoid(float*, float*, int)</td>
</tr>
<tr>
<td>1.15%</td>
<td>6.6560us</td>
<td>3</td>
<td>2.2180us</td>
<td>1.9840us</td>
<td>2.6560us</td>
<td>pw_vecMul(float*, float*, float*, int)</td>
</tr>
<tr>
<td>0.88%</td>
<td>5.0880us</td>
<td>2</td>
<td>2.5440us</td>
<td>2.3040us</td>
<td>2.7840us</td>
<td>pw_tanh(float*, float*, int)</td>
</tr>
</tbody>
</table>
```
LSTM Profile

Using `nvvp`

- Can run interactively
- Or use `nvprof -o a.nvp` and import file
SGEMM Performance
Back of the envelope

- SGEMM is a well known operation
- With the inputs chosen each should perform about 33 million floating point operations
- $33 \text{ million} / 64 \text{us} = \sim 516 \text{ GFLOPs}$.
  - GPU can do $\sim 6000 \text{ GFLOPs}$!
- What is wrong?
SGEMM Performance
What is wrong?

- Collect performance metrics:
  - Either via `nvprof --analysis-metrics` …
  - Or interactively

- A lot of information available
  - Guided analysis helps filter this down
  - Leads me to: “Optimization: Increase the number of blocks executed by the kernel.”
  - Expose more parallelism!
SGEMM Performance

Improvement #1

\[
\begin{align*}
[A_1][h] &= [y_1] \\
[A_2][h] &= [y_2] \\
[A_3][h] &= [y_3] \\
[A_4][h] &= [y_4]
\end{align*}
\]

- As our matrix operations share inputs we can combine them
## SGEMM Performance

### Improvement #1

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>88.46%</td>
<td>512.07us</td>
<td>8</td>
<td>64.009us</td>
<td>60.449us</td>
<td>75.618us</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.97%</td>
<td>213.19us</td>
<td>2</td>
<td>106.59us</td>
<td>104.90us</td>
<td>108.29us</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
</tbody>
</table>

**Before:**

**After:**
SGEMM Performance

Improvement #2

• We are still doing two independent matrix products
  • We can combine them
  • Or compute them simultaneously

\[
\begin{bmatrix}
A_1 \\
A_2 \\
A_3 \\
A_4
\end{bmatrix}
\begin{bmatrix}
y
\end{bmatrix}
= 
\begin{bmatrix}
B_1 \\
B_2 \\
B_3 \\
B_4
\end{bmatrix}
\begin{bmatrix}
[i] = z
\end{bmatrix}
\]
SGEMM Performance
Improvement #2

• We are still doing two independent matrix products
  • We can combine them
  • Or compute them simultaneously

\[
\begin{bmatrix}
A_1 \\
A_2 \\
A_3 \\
A_4
\end{bmatrix}
\begin{bmatrix}
B_1 \\
B_2 \\
B_3 \\
B_4
\end{bmatrix}
= \begin{bmatrix}
y \\
\end{bmatrix}
\begin{bmatrix}
h \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
B_1 \\
B_2 \\
B_3 \\
B_4
\end{bmatrix}
\begin{bmatrix}
i \\
\end{bmatrix}
= \begin{bmatrix}
z \\
\end{bmatrix}
\]
### SGEMM Performance

Matrix overlapping

<table>
<thead>
<tr>
<th>[0] Tesla M40</th>
<th>Context 1 (CUDA)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>▼ MemCpy (DtoD)</td>
</tr>
<tr>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 81.0% maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td></td>
<td>▼ 8.4% pw_biasAdd(float*, float*)</td>
</tr>
<tr>
<td></td>
<td>▼ 38% pw_vecAdd(float*, float*)</td>
</tr>
<tr>
<td></td>
<td>▼ 2.7% pw_sigmoid(float*, float*)</td>
</tr>
<tr>
<td></td>
<td>▼ 2.3% pw_vecMul(float*, float*)</td>
</tr>
<tr>
<td></td>
<td>▼ 1.8% pw_tanh(float*, float*)</td>
</tr>
<tr>
<td>Streams</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ Stream 13</td>
</tr>
<tr>
<td></td>
<td>▼ Stream 14</td>
</tr>
</tbody>
</table>
## Final optimization

### Fuse element-wise operations

<table>
<thead>
<tr>
<th>Tesla M40</th>
<th>Context 1 (CUDA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td></td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>LSTM...</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>95.1%</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>LSTM...</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>4.9% LSTM_elementWise_f...</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>Streams</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>LSTM...</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
</tbody>
</table>
## LSTM Performance

<table>
<thead>
<tr>
<th>Optimisation</th>
<th>Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>661us</td>
<td>(1.0x)</td>
</tr>
<tr>
<td>Combined matrices</td>
<td>357us</td>
<td>1.9x</td>
</tr>
<tr>
<td>Matrix streaming</td>
<td>250us</td>
<td>2.6x</td>
</tr>
<tr>
<td>Fused element-wise ops</td>
<td>136us</td>
<td>4.9x</td>
</tr>
</tbody>
</table>
Profiling

5x performance improvement

- Profiling helped to quickly identify the slow parts
- It showed that SGEMM was underusing the GPU
  - This was fixed by exposing more parallelism
- It showed that the pointwise operations were taking a significant proportion of our runtime
  - This was fixed by fusing them
The world’s most important event for GPU developers

CONNECT
Connect with technology experts from NVIDIA and other leading organizations

LEARN
Gain insight and hands-on training through the hundreds of sessions and research posters

DISCOVER
See how GPU technologies are creating amazing breakthroughs in important fields such as deep learning

INNOVATE
Hear about disruptive innovations as early-stage companies and startups present their work