GPU Profiling for Deep Learning
Jeremy Appleyard, October 2015
What is Profiling?

- Measuring application performance
- Simple profiling:
  - How long does an operation take?
- Advanced profiling:
  - Why does an operation take a long time?
Why Profile?

- Performance is important
- The first step to improve performance is to understand what parts are slow
- No point in improving performance of a region which doesn’t take much time!
Profiling Workflow

- Profile the code
- Find the slow parts
- Work out why they’re slow
- Make them fast
- Profile the code again
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
Profiling via Frameworks

- Some deep learning frameworks have internal profiling options
  - caffe time
  - theano ProfileMode
  - torch7 Timer
In this talk

- We will focus on `nvprof` and `nvvp`
  - `nvprof` => **NVIDIA Profiler**
    - Command line
  - `nvvp` => **NVIDIA Visual Profiler**
    - GUI based
Case Study

LSTM

- Long Short Term Memory
  - First published in 1997 by Sepp Hochreiter and Jürgen Schmidhuber
- Recurrent Neural Network with potential for long-term memory
  - Designed to deal with the vanishing gradient problem
- Often used in natural language processing
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

Many recurrent cells
LSTM

Potentially many layers
LSTM
Forward pass from a computational perspective

Computational Requirements
- Four matrix products with input $h$
- Four matrix products with input $i$
- Element-wise operations on the results of these matrix products
cudaProfilerStart();

// LSTM Forward pass
float alpha = 1.f;
float beta = 0.f;

// SGEMM operations
for (int i = 0; i < 4; i++) {
    cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N,
                hiddenSize, miniBatch, hiddenSize,
                &alpha, 
                &T[i * hiddenSize * hiddenSize],
                hiddenSize, 
                in_h, 
                hiddenSize,
                &beta,
                &tmp[i * hiddenSize * miniBatch],
                hiddenSize);

cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N,
            hiddenSize, miniBatch, hiddenSize,
            &alpha, 
            &T[(i + 4) * hiddenSize * hiddenSize],
            hiddenSize, 
            in_i, 
            hiddenSize,
            &beta,
            &tmp[(i + 4) * hiddenSize * miniBatch],
            hiddenSize);
}

// Elementwise operations
for (int i = 0; i < 8; i++) {
    addBias <<< miniBatch, hiddenSize >>>
        (&tmp[i * hiddenSize * miniBatch],
         &bias[i * hiddenSize * miniBatch]);
}

for (int i = 0; i < 4; i++) {
    vecAdd <<< miniBatch, hiddenSize >>>
        (&tmp[i * hiddenSize * miniBatch],
         &tmp[(i + 4) * hiddenSize * miniBatch],
         &linearGates[i * hiddenSize * miniBatch]);
}

nonLin <<< miniBatch, hiddenSize >>>
    (linearGates, in_h, out_h, out_i);

// Completed forward pass
cudaProfilerStop();
nvprof
nvprof

Simple usage

- nvprof ./<executable> <args>

- I will profile with:
  - A hidden layer size of 512
  - A minibatch size of 64
  - nvprof ./LSTM 512 64

- cudaProfilerStart and cudaProfilerStop can be used to profile only regions of interest
NVPROF is profiling process 22964, command: ./LSTM 512 64

Profiling application: ./LSTM 512 64

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>93.93%</td>
<td>575.72us</td>
<td>8</td>
<td>71.964us</td>
<td>70.241us</td>
<td>78.945us</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>3.60%</td>
<td>22.080us</td>
<td>8</td>
<td>2.7600us</td>
<td>2.3360us</td>
<td>3.5840us</td>
<td>addBias(float*, float*)</td>
</tr>
<tr>
<td>1.43%</td>
<td>8.7680us</td>
<td>4</td>
<td>2.1920us</td>
<td>2.0800us</td>
<td>2.4640us</td>
<td>vecAdd(float*, float*, float*)</td>
</tr>
<tr>
<td>1.04%</td>
<td>6.3680us</td>
<td>1</td>
<td>6.3680us</td>
<td>6.3680us</td>
<td>6.3680us</td>
<td>nonLin(float*, float*, float*, float*)</td>
</tr>
</tbody>
</table>

API calls:

<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>97.04%</td>
<td>103.55ms</td>
<td>21</td>
<td>4.9308ms</td>
<td>10.606us</td>
<td>103.30ms</td>
<td>cudaLaunch</td>
</tr>
<tr>
<td>2.08%</td>
<td>2.2189ms</td>
<td>249</td>
<td>8.9110us</td>
<td>202ns</td>
<td>350.88us</td>
<td>cuDeviceGetAttribute</td>
</tr>
<tr>
<td>0.53%</td>
<td>568.27us</td>
<td>21</td>
<td>27.060us</td>
<td>286ns</td>
<td>557.64us</td>
<td>cudaConfigureCall</td>
</tr>
<tr>
<td>0.17%</td>
<td>176.23us</td>
<td>3</td>
<td>58.741us</td>
<td>57.818us</td>
<td>59.862us</td>
<td>cuDeviceTotalMem</td>
</tr>
<tr>
<td>0.14%</td>
<td>147.11us</td>
<td>3</td>
<td>49.036us</td>
<td>46.468us</td>
<td>52.966us</td>
<td>cuDeviceGetName</td>
</tr>
<tr>
<td>0.04%</td>
<td>42.216us</td>
<td>128</td>
<td>329ns</td>
<td>240ns</td>
<td>5.1400us</td>
<td>cudaSetupArgument</td>
</tr>
<tr>
<td>0.00%</td>
<td>4.3100us</td>
<td>8</td>
<td>538ns</td>
<td>354ns</td>
<td>1.7220us</td>
<td>cudaGetLastError</td>
</tr>
<tr>
<td>0.00%</td>
<td>3.7640us</td>
<td>2</td>
<td>1.8820us</td>
<td>308ns</td>
<td>3.4560us</td>
<td>cuDeviceGetCount</td>
</tr>
<tr>
<td>0.00%</td>
<td>1.8660us</td>
<td>6</td>
<td>311ns</td>
<td>204ns</td>
<td>648ns</td>
<td>cuDeviceGet</td>
</tr>
</tbody>
</table>
SGEMM Performance

Back of the envelope

- GEMM performs ~2mnk floating point operations
  - In this case, m and k are the hidden layer size, n is the minibatch size
  - \( 512 \times 512 \times 64 \times 2 = 0.034 \) GFLOP
- The GPU I’m using can perform ~6,000 GFLOP per second
  - Best GEMM runtime is therefore: 5.7us

<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>93.93%</td>
<td>575.72us</td>
<td>8</td>
<td>71.964us</td>
<td>70.241us</td>
<td>78.945us</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
</tbody>
</table>

- 72us => ~500 GFLOP/s
SGEMM Performance

What’s wrong?

• GPUs are highly parallel machines
• Our problem does not have much parallelism
• How can we get more parallelism?
  • Change the hyperparameters
  • Look for algorithmic parallelism
SGEMM Performance

What’s wrong?

• GPUs are highly parallel machines
• Our problem does not have much parallelism
• How can we get more parallelism?
  • Change the hyperparameters
  • Look for algorithmic parallelism
Revisit the problem

### Computational Requirements
- Four matrix products with input h
- Four matrix products with input i
- Element-wise operations on the results of these matrix products
SGEMM Performance

Improvement #1

\[
\begin{align*}
[A_1][h] &= [x_1] \\
[A_2][h] &= [x_2] \\
[A_3][h] &= [x_3] \\
[A_4][h] &= [x_4]
\end{align*}
\]

- As our matrix operations share inputs we can combine them
## Combined Matrices

### 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>93.93%</td>
<td>575.72us</td>
<td>8</td>
<td>71.964us</td>
<td>70.241us</td>
<td>78.945us</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>84.40%</td>
<td>198.11us</td>
<td>2</td>
<td>99.057us</td>
<td>98.177us</td>
<td>99.937us</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
</tbody>
</table>

- From ~500 GFLOP/s to ~1350 GFLOP/s
SGEMM Performance
Improvement #2

• We are still doing two independent matrix products
• These can be performed in parallel with each other using library functionality

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

**Improvement #2**

<table>
<thead>
<tr>
<th>Name</th>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxwell_sgemm_128x64_tn</td>
<td>84.40%</td>
<td>198.11us</td>
<td>2</td>
<td>99.057us</td>
<td>98.177us</td>
<td>99.937us</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxwell_sgemm_128x64_tn</td>
<td>84.52%</td>
<td>197.92us</td>
<td>2</td>
<td>98.960us</td>
<td>87.744us</td>
<td>110.18us</td>
</tr>
</tbody>
</table>

- Because the two calls are overlapped the tool cannot show improvement
- Each took ~99us, but one started before the other finished
Overlapping Computation
Improvement #2

- **Different output mode:** `--print-gpu-trace`
- **Prints start time and duration of each GPU kernel called**
- **Before:** $0.1\text{ms} + 100\text{us} = 0.20\text{ms}$

<table>
<thead>
<tr>
<th>Start</th>
<th>Duration</th>
<th>Stream</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>773.98ms</td>
<td>97.857us</td>
<td>7</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>774.08ms</td>
<td>100.64us</td>
<td>7</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
</tbody>
</table>

- **After:** $0.06\text{ms} + 90\text{us} = 0.15\text{ms}$ (~1.35x speedup - ~1800 GFLOPs)

<table>
<thead>
<tr>
<th>Start</th>
<th>Duration</th>
<th>Stream</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>774.35ms</td>
<td>114.02us</td>
<td>13</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>774.41ms</td>
<td>89.569us</td>
<td>14</td>
<td>maxwell_sgemm_128x64_tn</td>
</tr>
</tbody>
</table>
SGEMM Performance
Improvement #3?

• 1800 GFLOPs is only ~30% of peak
• Problem is the same: not enough parallelism
  • Increase size of hyperparameters? (eg. Minibatch 128 => ~3100 GFLOPs)
  • More work that can be overlapped (eg. More layers)?
• 1800 GFLOPs is still very fast!
Visual Profiler
Visual Profiler

**nvvp**

- Either import the output of `nvprof -o ...`
- Or run application through GUI
LSTM Timeline

- Element-wise operations are taking half our time!
- A lot of time for these operations is on the CPU
  - Launch overhead
  - Typically $O(10\text{us})$
Elementwise Operations

Optimisation

• We pay launch overhead on every GPU launch
• Optimisation:
  • Reduce number of GPU launches
  • Each element can be operated on completely independently
  • Should only need 1 GPU launch
LSTM Timeline

- Process "LSTM" (10179)
- Thread 3173799712
  - Runtime API
  - Driver API
  - Profiling Overhead
- Quadro M6000 Context 1 (CUDA)
  - Compute
    - 96.0% maxwell_sgemm_128x64 tn
    - 40.0% elementWise(float*, f...)
  - Streams
    - Stream 13
    - Stream 14
cudaProfilerStart();

// LSTM Forward pass
float alpha = 1.f;
float beta = 0.f;

// SGEMM operations
cublasSetStream(handle, s1);
cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, 4 * hiddenSize, miniBatch, hiddenSize,
&alpha, T, hiddenSize,
in_h, hiddenSize,
&beta, tmp, 4 * hiddenSize);
cublasSetStream(handle, s2);
cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, 4 * hiddenSize, miniBatch, hiddenSize,
&alpha, &T[4 * hiddenSize * hiddenSize], hiddenSize,
in_i, hiddenSize,
&beta, &tmp[4 * hiddenSize * miniBatch], 4 * hiddenSize);

...
LSTM
From naïve to optimised

<table>
<thead>
<tr>
<th>Optimisation</th>
<th>Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>~670us</td>
<td>(1.0x)</td>
</tr>
<tr>
<td>Combined matrices</td>
<td>~320us</td>
<td>2.1x</td>
</tr>
<tr>
<td>Matrix overlapping</td>
<td>~300us</td>
<td>2.2x</td>
</tr>
<tr>
<td>Fused element-wise ops</td>
<td>~160us</td>
<td>4.2x</td>
</tr>
</tbody>
</table>

- Without profiling, this performance would have been hard to achieve
Further optimisations possible when considering LSTM as a whole
  - Eg. A single GEMM for all inputs

- Overlapping computation from multiple layers?
- Transposition?
Digging Deeper
nvprof
More advanced options

• nvprof -h

• There are quite a few options!

• Some useful ones:
  • -o: creates an output file which can be imported into nvvp
  • -m and -e: collect metrics or events
  • --analysis-metrics: collect all metrics for import into nvvp
  • --query-metrics and --query-events: query which metrics/events are available
Most are quite in-depth, however some useful ones for quick analysis

In general, events are only for the expert. Rarely useful

(A few) useful metrics:

- `dram_read_throughput`: Main GPU memory read throughput
- `dram_write_throughput`: Main GPU memory write throughput
- `flop_count_sp`: Number of single precision floating point operations
- `flop_count_dp`: Number of double precision floating point operations
More details
Deep dive into a kernel

- The profilers let us dig much deeper into individual kernels
- Moving from “it is going slow?” to “why is it going slow?”
- Let’s dive into the elementwise operation
- Requires interactive nvvp session, or output from --analysis-metrics
Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>[2 kernel instances] maxwell_sgemm_128x64_tn</td>
</tr>
<tr>
<td>1</td>
<td>[1 kernel instances] elementWise(float*, float*, float*, float*, float*, float*)</td>
</tr>
</tbody>
</table>
Kernel Performance Is Bound By Memory Bandwidth

For device "Quadro M6000" the kernel’s compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L2 Cache memory.

![Chart showing utilization of compute and memory (L2 Cache) operations](image-url)
More information
Bound by L2 bandwidth

- If we wanted to optimise further we might be able to do something with this information
  - Increase caching at a lower level?
  - Reduce the amount of data moved?
Conclusions
Profiling tips

- The nvprof output is a very good place to start
- The timeline is a good place to go next
- Only dig deep into a kernel if it’s taking a significant amount of your time.
- Where possible, try to match profiler output with theory
  - For example, if I know I’m moving 1GB, and my kernel takes 10ms, I expect the profiler to report 100GB/s.
  - Discrepancies are likely to mean your application isn’t doing what you thought it was
- Know your hardware:
  - If your GPU can do 6 TFLOPs, and you’re already doing 5.5 TFLOPs, you won’t go much faster!
Conclusions

- Profiling is a key step in optimisation
- Sometimes quite simple changes can lead to big improvements in performance
- Your mileage may vary!
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

The world’s most important event for GPU developers