Tracing and profiling dataflow applications

Pierre Zins, Michel Dagenais
December, 2017

Polytechnique Montréal
Laboratoire DORSAL
Agenda

- Introduction
- Existing tools for profiling
- Available platforms
- Current results
- Conclusion and future work
Machine learning and deep learning

- Extremely popular in many domains
- Quite expensive in terms of computation
- Always a need for better performance
- Should be able to train a model well on CPU + GPU platform

- Focus on Tensorflow
- Developed by Google
- Open-source since November 2015
- In October Tensorflow 1.4 has arrived

- Training multilayer perceptron (50 epochs):
  - CPU only : 240 s
  - CPU and GPU : 17 s
Introduction

- Dataflow model
- Computation graph

Research goals:

- Better understand the execution of the graph and its performance
- Insure that the GPU is used efficiently
- Detect problems or suggest optimizations
Introduction

Tensorflow program: two phases

1) Definition of the computation graph
2) Execution of the graph with the provided input data
   • Session: Links the python client and the core of Tensorflow

Tensorflow program pattern

```python
with tf.Session() as sess:
    for epoch in range(training_epochs):
        total_batch = int(mnist.train.num_examples/batch_size)
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            _, cost = sess.run([train_op, loss_op], feed_dict={X: batch_x, Y: batch_y})
```

Parameters: number of epochs, batch size, learning rate, ...

Network parameters: number of hidden layers, number of neurons, number of output classes, ...

Graph definition: neurons, layers, activation functions, loss functions, optimizer, ...
Existing tools for profiling

- Tensorflow internally collects metadata
- Limited to one session run
- Timeline module exports traces in Chrome Trace Format
- Visualization with the Chromium Viewer

- CPU only
Existing tools for profiling

- Tensorboard: suite of visualizing tools
- Based on metadata collected during the execution

Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, Abadi et al, November 2015
GPU backend

- Initially: CUDA
  - Nvidia libraries: cuDNN, CUPTI
  - Requires Nvidia hardware
  - Additional information is collected

- Alternatives:
  - SYCL
  - HipTensorflow
Alternatives - SYCL

- SYCL: C++ Single-source Heterogeneous Programming for OpenCL
- Tensorflow support of SYCL developed by Codeplay and Google
- Eigen support of SYCL
- Implementations: Codeplay LibComputeCpp, TriSYCL, SYCL-GTX
- Require proprietary OpenCL driver: AMD fglrx or AMDGPUPRO

```c++
#include <sycl(sycl)

template <typename T, size_t N>
void simple_vadd(const std::array<T, N> &A, const std::array<T, N> &B,
                 std::array<T, N> &C)
{
    cl::sycl::queue deviceQueue;
    cl::sycl::range<1> numOfItems(N);
    cl::sycl::buffer<T, 1> bufferA(A.data(), numOfItems);
    cl::sycl::buffer<T, 1> bufferB(B.data(), numOfItems);
    cl::sycl::buffer<T, 1> bufferC(C.data(), numOfItems);

    deviceQueue.submit([&](cl::sycl::handler &gh) {
        auto accessorA = bufferA.template get_access<cl<sycl_read>(gh);
        auto accessorB = bufferB.template get_access<cl<sycl_read>(gh);
        auto accessorC = bufferC.template get_access<cl<sycl_write>(gh);

        gh.parallel_for<class SimpleVadd<T>>(numOfItems,
                                             [=](cl::sycl::id<1> wID) {
                                                accessorC[wID] = accessorA[wID] + accessorB[wID];
                                            });
    });
    deviceQueue.wait_and_throw();
}
```
Alternatives - HSA

- HipTensorflow from AMD
- Work on HSA platform
- Three main advantages:
  - Single address space accessible to both CPU and GPU
  - User-space queues
  - Preemptive context switching
Alternatives - HSA

HipTensorflow

- hipTensorflow
- hipEigen

HIP

HC

ROCM runtime (HSA)  
ROCM Kernel Driver

Other libraries from AMD

- MIOpen
- rocBLAS
- hcFFT
- hipBLAS
- hcRNG

Hipify

clang-based tool to automatically translate CUDA source code into HIP C++

Compiler

HCC
Alternatives - HSA

- Profiling:
  - ROCm Profiler (RCP): HSA API, kernel execution time, data transfers and performance counters
  - HIP Profiling: AMD markers for function entry and exit
  - HC Profiling: kernel execution, barrier, asynchronous copy

```c
#define HIP_BEGIN_MARKER(markerName, group) amdBeginMarker(markerName, group, nullptr);
#define HIP_END_MARKER(markerName, group) amdEndMarkerEx(markerName, group, nullptr);
```

- Instrumentation of Tensorflow with markers

```c
HIP_BEGIN_MARKER(op_kernel->name().c_str(), marker_name.c_str());
device->Compute(CHECK_NOTNULL(op_kernel), &ctx);
HIP_END_MARKER(op_kernel->name().c_str(), marker_name.c_str());
```
### Alternatives - HSA

<table>
<thead>
<tr>
<th>Host</th>
<th>Host Thread 10250</th>
<th>Host Thread 10275</th>
<th>Host Thread 10283</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HIP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>session.run()</td>
<td>DirectSession::Run</td>
<td>DirectSession::Run</td>
<td>DirectSession::Run</td>
</tr>
<tr>
<td>HIP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TF_kernel_sync/cpu0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| HSA  | Host Thread 10283 |                  |                  |
| HIP  |                  |                  |                  |
| TF_kernel_sync/job:localhost/replica:0/task:0/cpu0 |                  |                  |                  |
| HIP  |                  |                  |                  |
| TF_kernel_sync/job:localhost/replica:0/task:0/cpu0 |                  |                  |                  |
| HIP  |                  |                  |                  |
| HSA  | Queue 2 - Device 0 (gfx803) | Queue 3 - Device 0 (gfx803) |                  |

#### Data Transfers
- 140335123926176 (gfx803) -> 140335123926176 (gfx803)
- 140335123926176 (gfx803) -> 35845552 (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz)
- 35845552 (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz) -> 140335123926176 (gfx803)
Alternatives - HSA

Performance counters
- ROCm-Profiler: flag -C
  - Get the counters for every kernel launched by the application
  - Some kernels require multiple passes
  - Not suitable for all applications

- GPUPerf API library (GPA)
  - Used by the profiler
  - More complex to use
  - More flexibility
Example

autoencoder.py

- Training loop:

```python
for i in range(10):
    for start, end in zip(range(0, len(trX), batch_size), range(batch_size, len(trX)+1, batch_size)):
        input_ = trX[start:end]
        mask_np = np.random.binomial(1, 1 - corruption_level, input_.shape)
        sess.run(train_op, feed_dict={X: input_, mask: mask_np})
```

Duration: 13.1 s

*Can we improve it?*

*Existing tools based on the collected metadata in Tensorflow are not useful in this case*

Mean interval duration (530 session runs): 19 ms
Example

autoencoder.py

- Training loop:

```python
maskQ = Queue()
def worker():
    while not is_end:
        mask_np = np.random.binomial(1, 1 - corruption_level, [1024, 784])
        maskQ.put(mask_np)

threads = []
for i in range(2):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()

for i in range(10):
    for start, end in zip(range(0, len(trX), batch_size), range(batch_size, len(trX)+1, batch_size)):
        input_ = trX[start:end]
        sess.run(train_op, feed_dict={X: input_, mask: maskQ.get()})
```

Duration : 8.7 s

Mean interval duration (530 session runs) : 10.2 ms
Autoencoder: GPU usage

- Change batch size to increase the GPU usage
- Limitation: can affect the learning, because bigger batches mean less weights updates
- The difference is really obvious with the traces

<table>
<thead>
<tr>
<th>Batch size</th>
<th>Duration</th>
<th>GPU usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>144 s</td>
<td>18 %</td>
</tr>
<tr>
<td>64</td>
<td>104 s</td>
<td>21 %</td>
</tr>
<tr>
<td>80</td>
<td>93 s</td>
<td>23 %</td>
</tr>
</tbody>
</table>

Learn and stop when accuracy is higher than a threshold (0.9)

**autoencoder**

<table>
<thead>
<tr>
<th>Batch size</th>
<th>Duration</th>
<th>gpu_usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>267 s</td>
<td>17 %</td>
</tr>
<tr>
<td>512</td>
<td>144 s</td>
<td>20 %</td>
</tr>
<tr>
<td>1024</td>
<td>75 s</td>
<td>22 %</td>
</tr>
<tr>
<td>2048</td>
<td>51 s</td>
<td>30 %</td>
</tr>
</tbody>
</table>

**multilayer perceptron**
Input pipeline
- Existing tools are not suitable for analyzing the input pipeline
- The graph should not be waiting for new data

Using feed_dict
- Input data preparation: 1.5 ms
- Python feed_dict: 0.6 ms
- Execution time: 220s (50000 iterations)

Using Tensorflow Queue
- Dequeue data: 0.3 ms
- Execution time: 182 s (50000 iterations)
Conversion to CTF

- ROCm-profiler: outputs ATP format
- ATP: text format with 4 sections
  - API calls
  - API timestamps
  - Kernel execution timestamps
  - Performance markers

**CodeXL:**
- Direct visualization of ATP traces
- Difficulties with large traces (> 200-300 Mb)

**TraceCompass:**
- Manages well large traces
- Combination with kernel traces

### Elements

<table>
<thead>
<tr>
<th>Elements</th>
<th>Method / instrumentation</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tensorflow</td>
<td>AMD markers</td>
<td>ROCm-Profiler</td>
</tr>
<tr>
<td>HIP</td>
<td>AMD markers</td>
<td>ROCm-Profiler</td>
</tr>
<tr>
<td>ROCm Runtime (HSA)</td>
<td>API Interception</td>
<td>ROCm-Profiler</td>
</tr>
<tr>
<td>Kernel execution</td>
<td>AMD special profiling functions</td>
<td>ROCm-Profiler</td>
</tr>
<tr>
<td>Data transfers</td>
<td>Deduced from HSA API calls</td>
<td>CodeXL</td>
</tr>
<tr>
<td>Python code</td>
<td>Lttng python logging</td>
<td>Lttng</td>
</tr>
</tbody>
</table>

- Scripts to convert ATP into CTF using Babeltrace python bindings
- XML Callstack View for Tracecompass
View TraceCompass
View TraceCompass
Conclusion

- Existing tools provide information for each node of the graph: execution time, memory usage
- Combining several profiling tools provide more detailed information
- Help to better understand the execution and the performance

Future work

- Investigate more the combination of kernel and userspace traces and try it on a non-HSA platform like CUDA or SYCL
- Integrate performance counters informations
- Compute more statistics from the trace
- Find more use-cases
Thank you!
Questions?