This repository | Search Pull requests Issues Gist

tensorflow / tensorflow Watch 4,746 Star 52,170 Fork 24,551

Code Issues 929 Pull requests 57 Projects 0 Pulse Graphs

Branch: master tensorflow / tensorflow / tools / benchmark / benchmark_model.cc Find file Copy path

snnn Add cmake build for benchmark_model e10c0b4 19 days ago

8 contributors

537 lines (481 sloc) 20.3 KB Raw Blame History

```
/* Copyright 2016 The TensorFlow Authors. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
==============================================================================*/

// A C++ binary to benchmark a compute graph and its individual operators,
// both on desktop machines and on Android.
//
```

```
19  // See README.md for usage instructions.
20
21  #include "tensorflow/tools/benchmark/benchmark_model.h"
22
23  #include <cstdlib>
24  #include <memory>
25  #include <string>
26  #include <unordered_set>
27  #include <vector>
28
29  #include "tensorflow/core/framework/graph.pb.h"
30  #include "tensorflow/core/framework/tensor.h"
31  #include "tensorflow/core/graph/algorithm.h"
32  #include "tensorflow/core/graph/graph.h"
33  #include "tensorflow/core/graph/graph_constructor.h"
34  #include "tensorflow/core/lib/strings/str_util.h"
35  #include "tensorflow/core/lib/strings/strcat.h"
36  #include "tensorflow/core/platform/env.h"
37  #include "tensorflow/core/platform/init_main.h"
38  #include "tensorflow/core/platform/logging.h"
39  #include "tensorflow/core/platform/platform.h"
40  #include "tensorflow/core/platform/types.h"
41  #include "tensorflow/core/public/session.h"
42  #include "tensorflow/core/util/command_line_flags.h"
43  #include "tensorflow/core/util/reporter.h"
44  #include "tensorflow/core/util/stat_summarizer.h"
45
46  namespace tensorflow {
47  namespace benchmark_model {
48
49  Status InitializeSession(int num_threads, const string& graph,
50                           std::unique_ptr<Session>* session,
51                           std::unique_ptr<GraphDef>* graph_def) {
52    LOG(INFO) << "Loading TensorFlow.";
53
54    tensorflow::SessionOptions options;
```

```
  tensorflow::ConfigProto& config = options.config;
  if (num_threads > 0) {
    config.set_intra_op_parallelism_threads(num_threads);
  }
  LOG(INFO) << "Got config, " << config.device_count_size() << " devices";

  session->reset(tensorflow::NewSession(options));
  graph_def->reset(new GraphDef());
  tensorflow::GraphDef tensorflow_graph;
  Status s = ReadBinaryProto(Env::Default(), graph, graph_def->get());
  if (!s.ok()) {
    LOG(ERROR) << "Could not create TensorFlow Graph: " << s;
    return s;
  }

  s = (*session)->Create(*(graph_def->get()));
  if (!s.ok()) {
    LOG(ERROR) << "Could not create TensorFlow Session: " << s;
    return s;
  }

  return Status::OK();
}

template <class T>
void InitializeTensor(const std::vector<float>& initialization_values,
                      Tensor* input_tensor) {
  auto type_tensor = input_tensor->flat<T>();
  type_tensor = type_tensor.constant(0);
  if (!initialization_values.empty()) {
    for (int i = 0; i < initialization_values.size(); ++i) {
      type_tensor(i) = static_cast<T>(initialization_values[i]);
    }
  }
}

```

```
91  void CreateTensorsFromInputInfo(
92      const std::vector<InputLayerInfo>& inputs,
93      std::vector<std::pair<string, tensorflow::Tensor> >* input_tensors) {
94    for (const InputLayerInfo& input : inputs) {
95      Tensor input_tensor(input.data_type, input.shape);
96      switch (input.data_type) {
97        case DT_INT32: {
98          InitializeTensor<int32>(input.initialization_values, &input_tensor);
99          break;
100       }
101       case DT_FLOAT: {
102         InitializeTensor<float>(input.initialization_values, &input_tensor);
103         break;
104       }
105       case DT_QUINT8: {
106         InitializeTensor<quint8>(input.initialization_values, &input_tensor);
107         break;
108       }
109       case DT_UINT8: {
110         InitializeTensor<uint8>(input.initialization_values, &input_tensor);
111         break;
112       }
113       case DT_BOOL: {
114         InitializeTensor<bool>(input.initialization_values, &input_tensor);
115         break;
116       }
117       case DT_STRING: {
118         if (!input.initialization_values.empty()) {
119           LOG(FATAL) << "Initialization values are not supported for strings";
120         }
121         auto type_tensor = input_tensor.flat<string>();
122         type_tensor = type_tensor.constant("");
123         break;
124       }
125       default:
126         LOG(FATAL) << "Unsupported input type: "
```

```
127                     << DataTypeString(input.data_type);
128       }
129       input_tensors->push_back({input.name, input_tensor});
130     }
131   }
132
133   Status GetOutputShapes(const std::vector<InputLayerInfo>& inputs,
134                          const std::set<string>& wanted_shapes, Session* session,
135                          std::unordered_map<string, TensorShape>* node_shapes) {
136     std::vector<std::pair<string, tensorflow::Tensor> > input_tensors;
137     CreateTensorsFromInputInfo(inputs, &input_tensors);
138     std::vector<tensorflow::Tensor> output_tensors;
139     std::vector<string> output_tensor_names(wanted_shapes.begin(),
140                                             wanted_shapes.end());
141     TF_RETURN_IF_ERROR(
142         session->Run(input_tensors, output_tensor_names, {}, &output_tensors));
143     CHECK_EQ(output_tensors.size(), output_tensor_names.size());
144     for (int i = 0; i < output_tensor_names.size(); ++i) {
145       const string& wanted_shape_name = output_tensor_names[i];
146       const TensorShape& found_shape = output_tensors[i].shape();
147       (*node_shapes)[wanted_shape_name] = found_shape;
148     }
149     return Status::OK();
150   }
151
152   Status CalculateFlops(const GraphDef& graph,
153                         const std::vector<InputLayerInfo>& inputs,
154                         Session* session, int64* total_flops,
155                         std::unordered_map<string, int64>* flops_by_op) {
156     std::unordered_set<string> floppable_ops = {
157         "Conv2D", "MatMul", "QuantizedConv2D", "QuantizedMatMul"};
158
159     std::set<string> wanted_shapes;
160     for (const NodeDef& node : graph.node()) {
161       if (floppable_ops.count(node.op())) {
162         for (const string& input : node.input()) {
```

```
163           wanted_shapes.insert(input);
164         }
165         wanted_shapes.insert(node.name());
166       }
167     }
168     std::unordered_map<string, TensorShape> found_shapes;
169     TF_RETURN_IF_ERROR(
170         GetOutputShapes(inputs, wanted_shapes, session, &found_shapes));
171
172     *total_flops = 0;
173     for (const NodeDef& node : graph.node()) {
174       if (floppable_ops.count(node.op())) {
175         int64 current_flops = 0;
176         // This is a very crude approximation to FLOPs that only looks at a few
177         // op types that commonly form the bulk of the computation for many
178         // models. It's included here because getting even an approximate value
179         // for FLOPs is still very useful for estimating utilization, versus a
180         // device's theoretical maximum FLOPs/second.
181         if ((node.op() == "Conv2D") || (node.op() == "QuantizedConv2D")) {
182           const TensorShape& filter_shape = found_shapes[node.input(1)];
183           const TensorShape& output_shape = found_shapes[node.name()];
184           int64 filter_height = filter_shape.dim_size(0);
185           int64 filter_width = filter_shape.dim_size(1);
186           int64 filter_in_depth = filter_shape.dim_size(2);
187           int64 output_count = output_shape.num_elements();
188           current_flops =
189               output_count * filter_in_depth * filter_height * filter_width * 2;
190         } else if ((node.op() == "MatMul") || (node.op() == "QuantizedMatMul")) {
191           const bool transpose_a = node.attr().at("transpose_a").b();
192           const TensorShape& a_shape = found_shapes[node.input(0)];
193           const TensorShape& output_shape = found_shapes[node.name()];
194           int64 k;
195           if (transpose_a) {
196             k = a_shape.dim_size(0);
197           } else {
198             k = a_shape.dim_size(1);
```

```
199           }
200           int64 output_count = output_shape.num_elements();
201           current_flops = k * output_count * 2;
202         }
203         (*flops_by_op)[node.op()] += current_flops;
204         *total_flops += current_flops;
205       }
206     }
207     return Status::OK();
208   }
209
210   Status RunBenchmark(const std::vector<InputLayerInfo>& inputs,
211                       const std::vector<string>& outputs, Session* session,
212                       StatSummarizer* stats, int64* inference_time_us) {
213     std::vector<std::pair<string, tensorflow::Tensor> > input_tensors;
214     CreateTensorsFromInputInfo(inputs, &input_tensors);
215
216     std::vector<tensorflow::Tensor> output_tensors;
217
218     tensorflow::Status s;
219
220     RunOptions run_options;
221     if (stats != nullptr) {
222       run_options.set_trace_level(RunOptions::FULL_TRACE);
223     }
224
225     RunMetadata run_metadata;
226     const int64 start_time = Env::Default()->NowMicros();
227     s = session->Run(run_options, input_tensors, outputs, {}, &output_tensors,
228                      &run_metadata);
229     const int64 end_time = Env::Default()->NowMicros();
230     *inference_time_us = end_time - start_time;
231
232     if (!s.ok()) {
233       LOG(ERROR) << "Error during inference: " << s;
234       return s;
```

```
235     }
236
237     if (stats != nullptr) {
238       assert(run_metadata.has_step_stats());
239       const StepStats& step_stats = run_metadata.step_stats();
240       stats->ProcessStepStats(step_stats);
241     }
242
243     return s;
244   }
245
246   Status TimeMultipleRuns(double sleep_seconds, int num_runs,
247                           const std::vector<InputLayerInfo>& inputs,
248                           const std::vector<string>& outputs, Session* session,
249                           StatSummarizer* stats, int64* total_time_us) {
250     // Convert the run_delay string into a timespec.
251     timespec req;
252     req.tv_sec = static_cast<time_t>(sleep_seconds);
253     req.tv_nsec = (sleep_seconds - req.tv_sec) * 1000000000;
254
255     *total_time_us = 0;
256
257     LOG(INFO) << "Running benchmark for " << num_runs << " iterations "
258               << (stats != nullptr ? "with" : "without")
259               << " detailed stat logging:";
260
261     Stat<int64> stat;
262     for (int i = 0; i < num_runs; ++i) {
263       int64 time;
264       Status run_status = RunBenchmark(inputs, outputs, session, stats, &time);
265       stat.UpdateStat(time);
266       *total_time_us += time;
267       if (!run_status.ok()) {
268         LOG(INFO) << "Failed on run " << i;
269         return run_status;
270       }
```

```
271
272       // If requested, sleep between runs for an arbitrary amount of time.
273       // This can be helpful to determine the effect of mobile processor
274       // scaling and thermal throttling.
275       if (sleep_seconds > 0.0) {
276   #ifdef PLATFORM_WINDOWS
277         Sleep(sleep_seconds * 1000);
278   #else
279         nanosleep(&req, nullptr);
280   #endif
281       }
282     }
283     std::stringstream stream;
284     stat.OutputToStream(&stream);
285     LOG(INFO) << stream.str() << std::endl;
286
287     return Status::OK();
288   }
289
290   int Main(int argc, char** argv) {
291     string graph = "/data/local/tmp/tensorflow_inception_graph.pb";
292     string input_layer_string = "input:0";
293     string input_layer_shape_string = "1,224,224,3";
294     string input_layer_type_string = "float";
295     string input_layer_values_string = "";
296     string output_layer_string = "output:0";
297     int num_runs = 50;
298     string run_delay = "-1.0";
299     int num_threads = -1;
300     string benchmark_name = "";
301     string output_prefix = "";
302     bool show_sizes = false;
303     bool show_run_order = true;
304     int run_order_limit = 0;
305     bool show_time = true;
306     int time_limit = 10;
```

```
307     bool show_memory = true;
308     int memory_limit = 10;
309     bool show_type = true;
310     bool show_summary = true;
311     bool show_flops = false;
312     int warmup_runs = 2;
313
314     std::vector<Flag> flag_list = {
315         Flag("graph", &graph, "graph file name"),
316         Flag("input_layer", &input_layer_string, "input layer names"),
317         Flag("input_layer_shape", &input_layer_shape_string, "input layer shape"),
318         Flag("input_layer_type", &input_layer_type_string, "input layer type"),
319         Flag("input_layer_values", &input_layer_values_string,
320              "values to initialize the inputs with"),
321         Flag("output_layer", &output_layer_string, "output layer name"),
322         Flag("num_runs", &num_runs, "number of runs"),
323         Flag("run_delay", &run_delay, "delay between runs in seconds"),
324         Flag("num_threads", &num_threads, "number of threads"),
325         Flag("benchmark_name", &benchmark_name, "benchmark name"),
326         Flag("output_prefix", &output_prefix, "benchmark output prefix"),
327         Flag("show_sizes", &show_sizes, "whether to show sizes"),
328         Flag("show_run_order", &show_run_order,
329              "whether to list stats by run order"),
330         Flag("run_order_limit", &run_order_limit,
331              "how many items to show by run order"),
332         Flag("show_time", &show_time, "whether to list stats by time taken"),
333         Flag("time_limit", &time_limit, "how many items to show by time taken"),
334         Flag("show_memory", &show_memory, "whether to list stats by memory used"),
335         Flag("memory_limit", &memory_limit,
336              "how many items to show by memory used"),
337         Flag("show_type", &show_time, "whether to list stats by op type"),
338         Flag("show_summary", &show_time,
339              "whether to show a summary of the stats"),
340         Flag("show_flops", &show_flops, "whether to estimate the model's FLOPs"),
341         Flag("warmup_runs", &warmup_runs, "how many runs to initialize model"),
342     };
```

```
343   string usage = Flags::Usage(argv[0], flag_list);
344   const bool parse_result = Flags::Parse(&argc, argv, flag_list);
345
346   if (!parse_result) {
347     LOG(ERROR) << usage;
348     return -1;
349   }
350
351   std::vector<string> input_layers = str_util::Split(input_layer_string, ',');
352   std::vector<string> input_layer_shapes =
353       str_util::Split(input_layer_shape_string, ':');
354   std::vector<string> input_layer_types =
355       str_util::Split(input_layer_type_string, ',');
356   std::vector<string> input_layer_values =
357       str_util::Split(input_layer_values_string, ':');
358   std::vector<string> output_layers = str_util::Split(output_layer_string, ',');
359   if ((input_layers.size() != input_layer_shapes.size()) ||
360       (input_layers.size() != input_layer_types.size())) {
361     LOG(ERROR) << "There must be the same number of items in --input_layer,"
362                << " --input_layer_shape, and --input_layer_type, for example"
363                << " --input_layer=input1,input2 --input_layer_type=float,float "
364                << " --input_layer_shape=1,224,224,4:1,20";
365     LOG(ERROR) << "--input_layer=" << input_layer_string << " ("
366                << input_layers.size() << " items)";
367     LOG(ERROR) << "--input_layer_type=" << input_layer_type_string << " ("
368                << input_layer_types.size() << " items)";
369     LOG(ERROR) << "--input_layer_shape=" << input_layer_shape_string << " ("
370                << input_layer_shapes.size() << " items)";
371     return -1;
372   }
373   const size_t inputs_count = input_layers.size();
374
375   ::tensorflow::port::InitMain(argv[0], &argc, &argv);
376   if (argc > 1) {
377     LOG(ERROR) << "Unknown argument " << argv[1] << "\n" << usage;
378     return -1;
```

```
379     }
380
381     LOG(INFO) << "Graph: [" << graph << "]";
382     LOG(INFO) << "Input layers: [" << input_layer_string << "]";
383     LOG(INFO) << "Input shapes: [" << input_layer_shape_string << "]";
384     LOG(INFO) << "Input types: [" << input_layer_type_string << "]";
385     LOG(INFO) << "Output layers: [" << output_layer_string << "]";
386     LOG(INFO) << "Num runs: [" << num_runs << "]";
387     LOG(INFO) << "Inter-run delay (seconds): [" << run_delay << "]";
388     LOG(INFO) << "Num threads: [" << num_threads << "]";
389     LOG(INFO) << "Benchmark name: [" << benchmark_name << "]";
390     LOG(INFO) << "Output prefix: [" << output_prefix << "]";
391     LOG(INFO) << "Show sizes: [" << show_sizes << "]";
392     LOG(INFO) << "Warmup runs: [" << warmup_runs << "]";
393
394     std::unique_ptr<Session> session;
395     std::unique_ptr<StatSummarizer> stats;
396     std::unique_ptr<GraphDef> graph_def;
397     Status initialize_status =
398         InitializeSession(num_threads, graph, &session, &graph_def);
399     if (!initialize_status.ok()) {
400       return -1;
401     }
402
403     StatSummarizerOptions stats_options;
404     stats_options.show_run_order = show_run_order;
405     stats_options.run_order_limit = run_order_limit;
406     stats_options.show_time = show_time;
407     stats_options.time_limit = time_limit;
408     stats_options.show_memory = show_memory;
409     stats_options.memory_limit = memory_limit;
410     stats_options.show_type = show_type;
411     stats_options.show_summary = show_summary;
412     stats.reset(new tensorflow::StatSummarizer(stats_options));
413
414     const double sleep_seconds = std::strtod(run_delay.c_str(), nullptr);
```

```
415
416   std::vector<InputLayerInfo> inputs;
417   for (int n = 0; n < inputs_count; ++n) {
418     InputLayerInfo input;
419     CHECK(DataTypeFromString(input_layer_types[n], &input.data_type))
420         << input_layer_types[n] << " was an invalid type";
421     std::vector<int32> sizes;
422     CHECK(str_util::SplitAndParseAsInts(input_layer_shapes[n], ',', &sizes))
423         << "Incorrect size string specified: " << input_layer_shapes[n];
424     for (int i = 0; i < sizes.size(); ++i) {
425       input.shape.AddDim(sizes[i]);
426     }
427     input.name = input_layers[n];
428     if (n < input_layer_values.size()) {
429       CHECK(str_util::SplitAndParseAsFloats(input_layer_values[n], ',',
430                                             &input.initialization_values))
431           << "Incorrect initialization values string specified: "
432           << input_layer_values[n];
433     }
434     inputs.push_back(input);
435   }
436
437   // If requested, run through the graph first to preinitialize everything
438   // before the benchmarking runs.
439   int64 warmup_time_us = 0;
440   if (warmup_runs > 0) {
441     Status warmup_time_status =
442         TimeMultipleRuns(sleep_seconds, warmup_runs, inputs, output_layers,
443                          session.get(), nullptr, &warmup_time_us);
444     if (!warmup_time_status.ok()) {
445       LOG(ERROR) << "Timing failed with " << warmup_time_status;
446       return -1;
447     }
448   }
449
450   // Capture overall inference time without stat logging overhead. This is the
```

```
451     // timing data that can be compared to other libaries.
452     int64 no_stat_time_us = 0;
453     Status no_stat_time_status =
454         TimeMultipleRuns(sleep_seconds, num_runs, inputs, output_layers,
455                          session.get(), nullptr, &no_stat_time_us);
456     const double no_stat_wall_time = no_stat_time_us / 1000000.0;
457     if (!no_stat_time_status.ok()) {
458       LOG(ERROR) << "Timing failed with " << no_stat_time_status;
459       return -1;
460     }
461
462     // Run again to gather detailed log stats to get a better idea of where
463     // relative time is going within the graph.
464     int64 stat_time_us = 0;
465     Status stat_time_status =
466         TimeMultipleRuns(sleep_seconds, num_runs, inputs, output_layers,
467                          session.get(), stats.get(), &stat_time_us);
468     if (!stat_time_status.ok()) {
469       LOG(ERROR) << "Timing failed with " << stat_time_status;
470       return -1;
471     }
472
473     LOG(INFO) << "Average inference timings in us: "
474               << "Warmup: "
475               << (warmup_runs > 0 ? warmup_time_us / warmup_runs : 0) << ", "
476               << "no stats: " << no_stat_time_us / num_runs << ", "
477               << "with stats: " << stat_time_us / num_runs;
478
479     stats->PrintStepStats();
480
481     if (show_sizes) {
482       stats->PrintOutputs();
483     }
484
485     if (show_flops) {
486       int64 total_flops;
```

```
487       std::unordered_map<string, int64> flops_by_op;
488       Status flop_status = CalculateFlops(*graph_def, inputs, session.get(),
489                                           &total_flops, &flops_by_op);
490       if (!flop_status.ok()) {
491         LOG(ERROR) << "FLOPs calculation failed with " << flop_status;
492         return -1;
493       }
494       string pretty_flops;
495       if (total_flops < 1000) {
496         pretty_flops = strings::StrCat(total_flops, " FLOPs");
497       } else if (total_flops < (1000 * 1000)) {
498         const float rounded_flops = (total_flops / 1000.0f);
499         pretty_flops = strings::StrCat(rounded_flops, "k FLOPs");
500       } else if (total_flops < (1000 * 1000 * 1000)) {
501         const float rounded_flops = round(total_flops / 1000.0f) / 1000.0f;
502         pretty_flops = strings::StrCat(rounded_flops, " million FLOPs");
503       } else {
504         const float rounded_flops =
505             round(total_flops / (1000.0f * 1000.0f)) / 1000.0f;
506         pretty_flops = strings::StrCat(rounded_flops, " billion FLOPs");
507       }
508       LOG(INFO) << "FLOPs estimate: " << strings::HumanReadableNum(total_flops);
509       const double mean_run_time = no_stat_wall_time / num_runs;
510       LOG(INFO) << "FLOPs/second: "
511                 << strings::HumanReadableNum(
512                        static_cast<int64>(total_flops / mean_run_time));
513     }
514
515     if (!benchmark_name.empty() && !output_prefix.empty()) {
516       // Compute the total number of values per input.
517       int64 total_size = inputs[0].shape.num_elements();
518
519       // Throughput in MB/s
520       const double throughput =
521           DataTypeSize(inputs[0].data_type) * total_size * num_runs /
522           static_cast<double>(no_stat_wall_time) / (1024 * 1024);
```

```
523
524       // Report the stats.
525       TestReporter reporter(output_prefix, benchmark_name);
526       TF_QCHECK_OK(reporter.Initialize());
527       TF_QCHECK_OK(
528           reporter.Benchmark(num_runs, -1.0, no_stat_wall_time, throughput));
529       TF_QCHECK_OK(reporter.Close());
530     }
531
532     return 0;
533   }
534
535   }  // namespace benchmark_model
536   }  // namespace tensorflow
```

© 2017 GitHub, Inc. Terms Privacy Security Status Help

Contact GitHub API Training Shop Blog About