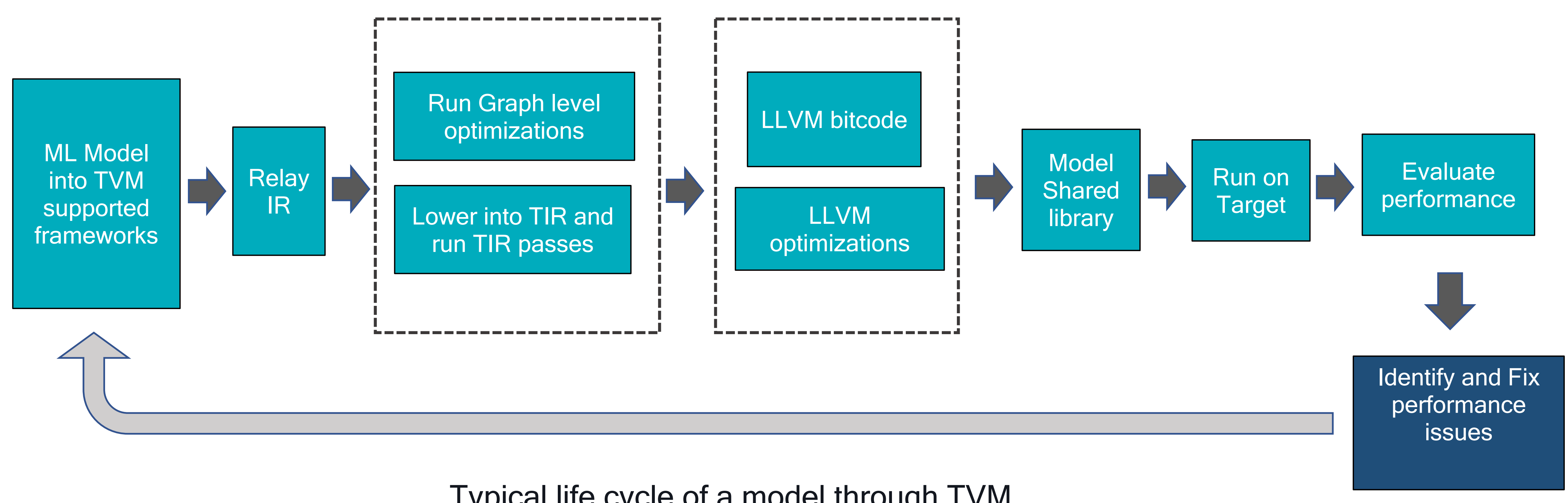
December, 2021 TVM Conference

Qualcomm

Lightweight Profiling for TVM

Jyotsna Verma
Qualcomm Innovation Center, Inc.

Motivation



Typical life cycle of a model through TVM

Profiling

- First step towards improving performance of any application
- Helps determine performance bottlenecks
- Provide feedback on performance optimizations
  - Not uncommon for a compiler optimization to work in some cases but not always, and may require adjustments

Current profiling capability in TVM

- Debug graph executor
  - Provides time elapsed for each layer in the graph, and reports as a JSON or CSV file
  - Can be very useful in the absence of other profiling tools
  - But limited to only coarse-grained function level information

Sample Output:

* Some of the columns are excluded

Name	Duration (us)	Percent
tvmgen_default_fused_nn_conv2d_add_multiply_add_multiply_add_nn_relu	10224300	23.94
tvmgen_default_fused_nn_conv2d_add_multiply_add_add_nn_relu_5	1837470	4.30238
tvmgen_default_fused_nn_conv2d_add_add_nn_relu_7	1268090	2.9692
tvmgen_default_fused_nn_conv2d_add_add_nn_relu_5	1152130	2.69767
tvmgen_default_fused_nn_conv2d_add_add_nn_relu_5	1151580	2.69638
tvmgen_default_fused_nn_conv2d_add_add_nn_relu_5	1150080	2.69289
tvmgen_default_fused_nn_conv2d_add_add_nn_relu_7	1126730	2.63821
tvmgen_default_fused_nn_conv2d_add_add_nn_relu_6	818770	1.91713
tvmgen_default_fused_nn_conv2d_add_add_nn_relu_6	813481	1.90474
tvmgen_default_fused_nn_conv2d_add_add_nn_relu_6	808125	1.8922

Additional profiling options

- On-device profiling capability
  - Best way to get real time performance metrics for an application
  - Often involves tedious process and may require specialized knowledge
- Simulator
  - Models on-target behavior and may provide detailed instruction level profiling info
  - Often simulators are slow and may not be feasible when in need for a quick feedback

Lightweight Profiling (LWP) in TVM

- Attempts to fill the gap between various profiling options
  - Provide loop level profiling in terms of processor cycles or any other target specific metric
    - ✓ More detailed information than debug graph executor
    - ✓ Not as detailed as hardware/simulator but a lot more efficient
  - Can be used to map TIR with LLVM bitcode and assembly

- A simple idea motivated by the profiling instrumentation in LLVM
  - LLVM's profiling instrumentation occurs in the front end
  - Also, it only tracks execution count and not actual cycles spent on a code segment

Basic Design

Involves 3 main components:

- TIR pass
    - Instrument functions/loops with TVM profile builtin
    - During codegen, builtin is replaced with a call to target specific handler
- LWP Handler
    - Records profiling data into a buffer
    - For Qualcomm® Hexagon ™, the handler is implemented in assembly and is optimized to minimize overhead.
- Process the profile data to construct a report

Qualcomm Hexagon is a product of Qualcomm Technologies, Inc. and/or it's subsidiaries.

TIR Pass Implementation Details

Instruments IRModule with profiling builtin

- Find depth of every loop in the function
- Assign unique ID to every function and loop in the module
- The maximum loop depth for instrumentation can be controlled using a flag during model compilation
- Instrumented loops and functions are encapsulated within the profiling builtin

With LWP instrumentation

```
, GlobalVar(tvmgen_main_fused_nn_conv2d_expand_dims_expand_dims_a
  allocate A[float32 * 1685440], storage_scope = global
  allocate B[float32 * 114688], storage_scope = global
  allocate C[float32 * 802816], storage_scope = global
  for (yy, 0, 229) {
    for (xx, 0, 230) {
      for (cc, 0, 32) {
        A[(((yy*7360) + (xx*32)) + cc)] = tir.if_then_else(((((c
      }
    }
  }
  for (ff, 0, 64) {
    for (yy, 0, 7) {
      for (xx, 0, 8) {
        for (cc, 0, 32) {
          B[((((ff*1792) + (yy*256)) + (xx*32)) + cc)] = tir.if_t
        }
      }
    }
  }
  for (i1, 0, 64) {
    for (i2, 0, 112) {
      for (i3, 0, 112) {
        C[(((i1*12544) + (i2*112)) + i3)] = 0f
        for (ry, 0, 7) {
          for (rc.rx.fused.outer, 0, 2) {
            for (rc.rx.fused.inner, 0, 128) {
              C[(((i1*12544) + (i2*112)) + i3)] = (C[(((i1*12544)
            }
          }
        }
      }
    }
  }
  for (ax1, 0, 64) {
    for (ax2, 0, 112) {
```

```
, GlobalVar(tvmgen_main_fused_nn_conv2d_expa
  tir.profile_intrinsic(0)
  allocate A[float32 * 1685440], storage_sco
  allocate B[float32 * 114688], storage_scop
  allocate C[float32 * 802816], storage_scop
  tir.profile_intrinsic(1)
  for (yy, 0, 229) {
    tir.profile_intrinsic(2)
    for (xx, 0, 230) {
      tir.profile_intrinsic(3)
      for (cc, 0, 32) {
        A[(((yy*7360) + (xx*32)) + cc)] = ti
      }
      tir.profile_intrinsic(3)
    }
    tir.profile_intrinsic(2)
  }
  tir.profile_intrinsic(1)
  tir.profile_intrinsic(4)
  for (ff, 0, 64) {
    tir.profile_intrinsic(5)
    for (yy, 0, 7) {
      tir.profile_intrinsic(6)
      for (xx, 0, 8) {
        for (cc, 0, 32) {
          B[((((ff*1792) + (yy*256)) + (xx*3
        }
      }
      tir.profile_intrinsic(6)
    }
    tir.profile_intrinsic(5)
  }
  tir.profile_intrinsic(4)
```

LWP Handler

- By default, profiling builtins are ignored during codegen
- Target specific codegen needs to lower builtin to their handler
- Target needs to provide their own handler implementation

Example Handler:

```
 2 extern unsigned int* __lwp_buffer_ptr;
 3 extern unsigned int __lwp_buffer_size;
 4 extern unsigned int __lwp_buffer_count;
 5
 6 void lwp_handler(int id, int ra){
 7   if (__lwp_buffer_count + 4 > __lwp_buffer_size)
 8     return;
 9   __lwp_buffer_ptr[__lwp_buffer_count++] = id; //Unique ID
10   __lwp_buffer_ptr[__lwp_buffer_count++] = ra; //Return Address
11
12   // Record processor cycles
13   uint64_t cycles = perf_get_pcycles();
14   uint32_t cycle_hi = (uint32_t) (cycles>>32);
15   uint32_t cycle_lo = (uint32_t) cycles;
16   __lwp_buffer_ptr[__lwp_buffer_count++] = cycle_hi;
17   __lwp_buffer_ptr[__lwp_buffer_count++] = cycle_lo;
18 }
```

Challenges

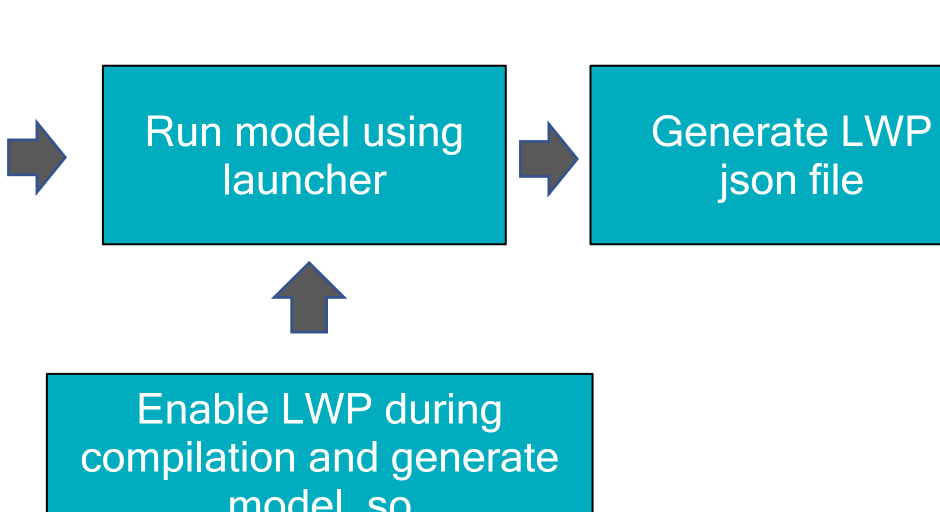
The simple design resulted in LWP buffer overflowing

- Added a secondary buffer which is indexed using the unique ID and is used to keep track of the execution count
- Reduced logging at a specific call site to first 100 instances
- Information from both buffers is combined offline to calculate total processor cycles for each loop

Basic workflow for LWP

Hexagon Launcher

- Uses TVM runtime to run a model on Hexagon backend
- Contains LWP handler implementation
- Writes runtime profiling data into a json file




```
{
  "entries":[
  {
    "ret":4156747296,
    "id":275,
    "cyc":1535811665485
  },
  {
    "ret":4156747472,
    "id":276,
    "cyc":1535812797161
  },
  {
    "ret":4156747544,
    "id":277,
    "cyc":1535812801820
  },
  {
    "ret":4156747736,
    "id":278,
    "cyc":1535812802188
  },
  {
    "ret":4156747760,
    "id":278,
    "cyc":1535812802248
  },
```


JSON file

LWP Report

Function Name	Loop/Function ID	Loop Depth	Start Offset	End Offset	Processor Cycles
tvmgen_default_fused_func_1	143	-	0x27bf4	0x284a8	1974627519
tvmgen_default_fused_func_1	144	0	0x27c94	0x27e88	880392
tvmgen_default_fused_func_1	145	1	0x27cb0	0x27e74	847455
tvmgen_default_fused_func_1	147	0	0x27e90	0x27f08	436511
tvmgen_default_fused_func_1	148	1	0x27ea8	0x27ef0	328120
tvmgen_default_fused_func_1	150	0	0x27f10	0x28168	193345969
tvmgen_default_fused_func_1	151	1	0x27f38	0x28150	193224582
tvmgen_default_fused_func_1	155	0	0x28170	0x28454	3976574
tvmgen_default_fused_func_1	156	1	0x281b0	0x28418	3973134
tvmgen_default_fused_func_2	335	-	0x177d0	0x181b4	651978932
tvmgen_default_fused_func_2	336	0	0x17870	0x17a58	21522853
tvmgen_default_fused_func_2	337	1	0x17880	0x17a3c	21489046
tvmgen_default_fused_func_2	339	0	0x17a60	0x17af8	49057
tvmgen_default_fused_func_2	340	1	0x17a7c	0x17adc	43719
tvmgen_default_fused_func_2	342	0	0x17b00	0x17e84	628936479
tvmgen_default_fused_func_2	343	1	0x17b30	0x17e5c	628843016
tvmgen_default_fused_func_2	347	0	0x17e90	0x18164	1444969
tvmgen_default_fused_func_2	348	1	0x17eb0	0x18134	1401827

Limitations

- Call to the handler has a potential to affect LLVM optimizations and therefore alter codegen
- Can affect performance of the instrumented code
  - Instrumentation of the inner loops should be avoided
  - For the outer loops, performance overhead should be minimal, and we should be able to adjust it during processing of the LWP data if needed.
- Doesn’t work for loops that are parallelized

Future work

- Replace handler calls with LLVM intrinsic to minimize any impact on codegen
- Provide additional compile time config flags
  - Better instrumentation control. For example:
    - Instrument loops with siblings
    - Use loop height as a criteria to limit instrumentation

Qualcomm

Thank you

Follow us on:

For more information, visit us at:

www.qualcomm.com & www.qualcomm.com/blog

Nothing in these materials is an offer to sell any of the components or devices referenced herein.

©2018-2021 Qualcomm Technologies, Inc. and/or its affiliated companies. All Rights Reserved.

Qualcomm and Hexagon are trademarks or registered trademarks of Qualcomm Incorporated. Other products and brand names may be trademarks or registered trademarks of their respective owners.

References in this presentation to “Qualcomm” may mean Qualcomm Incorporated, Qualcomm Technologies, Inc., and/or other subsidiaries or business units within the Qualcomm corporate structure, as applicable. Qualcomm Incorporated includes our licensing business, QTL, and the vast majority of our patent portfolio. Qualcomm Technologies, Inc., a subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of our engineering, research and development functions, and substantially all of our products and services businesses, including our QCT semiconductor business.