Optimizing OpenCL Kernels and Runtime for DNN Inference on FPGAs

Seung-Hun Chung and Tarek S. Abdelrahman

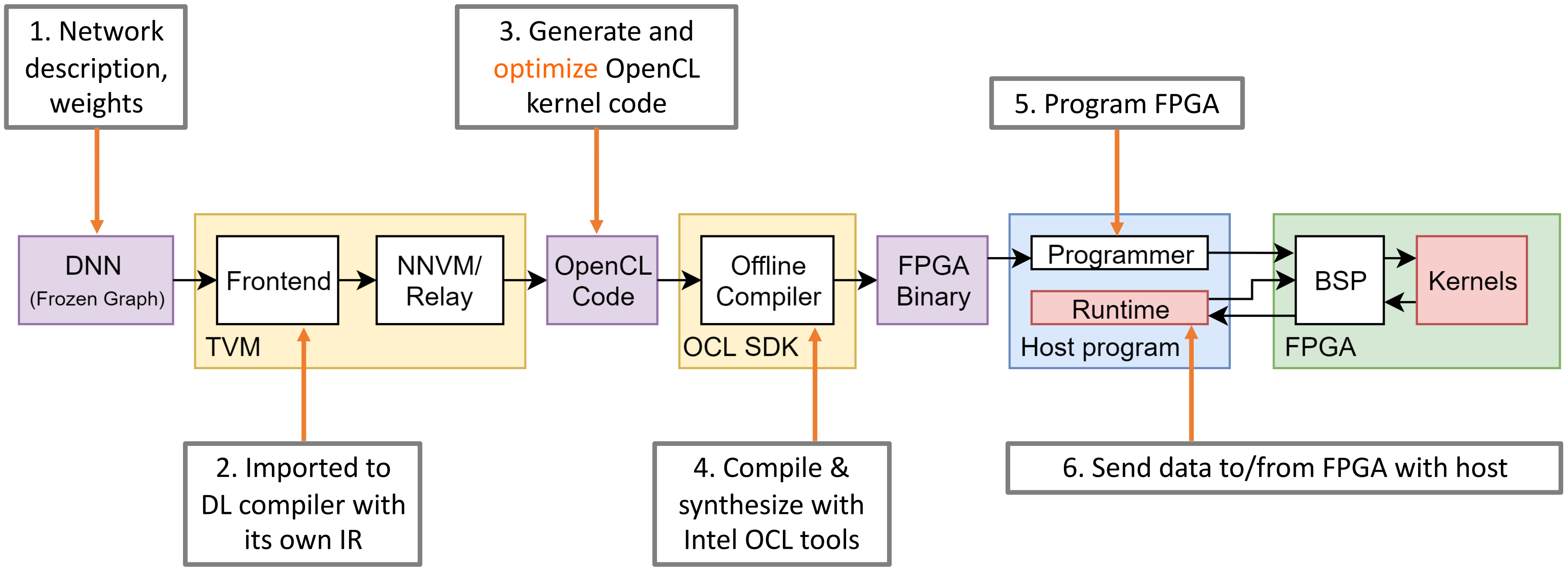
The Edward S. Rogers Dept. of Elec. & Comp. Engineering, University of Toronto

sh.chung@mail.utoronto.ca, tsa@ece.utoronto.ca

Overview

- Goal: enable fast hardware support for prototype DNN architectures by directly translating a frozen DNN model into FPGA hardware
- Approach: generate OpenCL kernels with a ML compiler (TVM) and then use HLS tools (aocl) to generate RTL descriptions
- Challenge: quality of generated hardware is very poor
- Opportunity: optimize the generated OpenCL code and host runtime to improve performance

Compilation Flow


1. Network description, weights
3. Generate and optimize OpenCL kernel code
5. Program FPGA
DNN (Frozen Graph)
Frontend
NNVM/ Relay
TVM
OpenCL Code
Offline Compiler
OCL SDK
FPGA Binary
Programmer
Runtime
Host program
BSP
Kernels
FPGA
2. Imported to DL compiler with its own IR
4. Compile & synthesize with Intel OCL tools
6. Send data to/from FPGA with host

Challenges

- Generated code from TVM designed for execution on GPUs
  - Overly relies on global memory with inefficient memory accesses leading to poor performance
- Granularity in the transfer of buffers is kernel/layer-level
- Host program can be improved for increased concurrency

Optimizations

1. Loop Unrolling (Kernel): increase the number of parallel operations by replicating hardware
2. Channels (Kernel): use FIFOs between kernel instances to conserve global memory bandwidth and reduce contention
3. Autorun Kernels (Kernel/Host): allow a kernel to execute independently of the host
4. Concurrent Execution (Host): allow kernels to execute concurrently
5. Kernel Reuse (Kernel): allow the reuse of kernels for similar layers

Loop Unrolling (Kernel Optimization)

- Increase parallelism by replicating hardware
- Can do more operations per cycle at the expense of increased resource usage (DSPs for MACs, logic/M20K for load-store units)
- Unroll factor is crucial

```
#pragma unroll M
for (int ax1 = 0; ax1 < 84; ++ax1) {
  float accum = 0.0f;
  #pragma unroll N
  for (int k = 0; k < 120; ++k) {
    accum += input0[k] * input1[ax1][k];
  }
}
```

```
N controls parallelism in inner loop
M controls parallelism in outer loop
```

Channels (Kernel Optimization)

- Reduce global memory BW traffic by moving kernel-to-kernel data via FIFOs

- Conserves memory bandwidth for filter weight and input array reads

```
channel float ch __attribute__((depth(32)));

kernel void layer1(global float *input) {
  // do something with input\
  for (int x = 0; x < N; x++) {
    float tmp += ...
    write_channel_intel(ch, tmp);
  }
}

kernel void layer2(...) {
  float local_memory_buf[N];
  for (int x = 0; x < N; x++) {
    local_memory_buf[x] = read_channel_intel(ch);
  }
}
```

Autorun & Concurrent Execution (Host Opts)

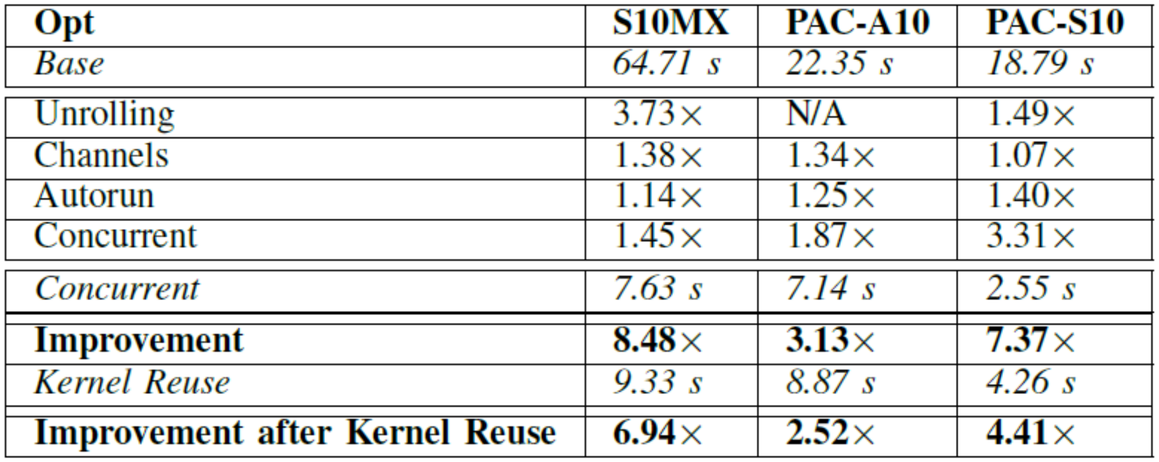
- Increase performance by removing communication overhead between host and kernel
- Autorun:
  - Kernel autonomously executes
  - Limitation: can only be used with kernels without arguments
  - Can use kernel-to-kernel channels to communicate
- Concurrent Execution
  - Maintain multiple command queues so that multiple kernels may execute concurrently
  - Useful for fully-pipelined implementations of CNNs

Kernel Reuse (Kernel Optimization)

- CNN operations are repetitive—can group multiple layers into single kernels to save resources
- Freed up resources can be put towards unrolling computation
- E.g. combine conv2d into single kernel, FC into another, pooling to another, etc.

Evaluation

Manual application of the optimization



Opt	**S10MX**	**PAC-A10**	**PAC-S10**
Base	*64.71 s*	*22.35 s*	*18.79 s*
Unrolling	3.73×	N/A	1.49×
Channels	1.38×	1.34×	1.07×
Autorun	1.14×	1.25×	1.40×
Concurrent	1.45×	1.87×	3.31×
Concurrent	*7.63 s*	*7.14 s*	*2.55 s*
Improvement	**8.48**×	**3.13**×	**7.37**×
Kernel Reuse	*9.33 s*	*8.87 s*	*4.26 s*
Improvement after Kernel Reuse	**6.94**×	**2.52**×	**4.41**×

TABLE I: Cumulative improvements of optimizations.

Optimizations impact is promising

Summary

- Optimizations increase parallelism, mitigate latency by reducing global memory accesses and reducing OpenCL control overhead
- Improvements seen over 3 different FPGA platforms, and up to 8.48x over the unoptimized bitstream
- Accelerator can perform up to 4.79x faster than CPU/TensorFlow
- Results encourage us to automate these optimizations and explore application to larger networks