An introduction to
Open Neural Network Compiler
Connecting ONNX to Proprietary DLAs

Luba Tang
2019/03/18
# ONNC Internals

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-10:30</td>
<td>ONNC Overview</td>
</tr>
<tr>
<td></td>
<td>ONNC Logistic Layers</td>
</tr>
<tr>
<td></td>
<td>ONNC Intermediate Representation</td>
</tr>
<tr>
<td>10:45-12:15</td>
<td>ONNC Pass Management</td>
</tr>
<tr>
<td></td>
<td>ONNC Target Backend</td>
</tr>
</tbody>
</table>

**Luba Tang**<luba@skymizer.com>

CEO & Founder of Skymizer Inc.
Architect of ONNC, MCLinker, and GYM compiler
Compiler and Linker/Electronic System Level Design
ONNC Overview

- Heterogeneous neural network processors
  - The problems that ONNC want to resolve

- How does ONNC resolve this problem?
  - Compiler solutions from ONNX to various neural network processors
One-Year Improvement is only 3%

Computers Stop Getting Faster
Moore’s Law and Dennard Scaling are dying

Based on SPECintCPU. Source: John Hennessy and David Patterson, Computer Architecture: A Quantitative Approach, 6/e. 2018
3.5 monthly doubling demand

Deep Learning (DL) Needs More Compute
The computation demand has increased exponentially

A new golden age in compute architecture

Fast DL Devices Need Tailored IC Design

DL Specific Architecture and Compiler become essential

Neural network models behave diversely

- Different networks have different latency and bandwidth requirement. There is no single architecture to fit all.

<table>
<thead>
<tr>
<th>Core Speed</th>
<th>Latency (ms)</th>
<th>DRAM BW 1GB/s</th>
<th>DRAM BW 2GB/s</th>
<th>DRAM BW 4GB/s</th>
<th>DRAM BW 8GB/s</th>
<th>DRAM BW 10GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>100MHz</td>
<td>75.8</td>
<td>47.2</td>
<td>32.9</td>
<td>25.7</td>
<td>24.3</td>
<td></td>
</tr>
<tr>
<td>200MHz</td>
<td>68.9</td>
<td>37.9</td>
<td>23.6</td>
<td>16.4</td>
<td>15.0</td>
<td></td>
</tr>
<tr>
<td>400MHz</td>
<td>63.4</td>
<td>33.5</td>
<td>19.0</td>
<td>11.8</td>
<td>10.4</td>
<td></td>
</tr>
<tr>
<td>800MHz</td>
<td>61.8</td>
<td>31.7</td>
<td>16.7</td>
<td>9.5</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>1000MHz</td>
<td>61.5</td>
<td>31.4</td>
<td>18.3</td>
<td>9.0</td>
<td>7.6</td>
<td></td>
</tr>
</tbody>
</table>

**Alexnet (~0.73 GOP, 61M weights)**
- Huge fully connected weights
- DRAM speed dominates
- Computation power cannot help

<table>
<thead>
<tr>
<th>Core Speed</th>
<th>Latency (ms)</th>
<th>DRAM BW 1GB/s</th>
<th>DRAM BW 2GB/s</th>
<th>DRAM BW 4GB/s</th>
<th>DRAM BW 8GB/s</th>
<th>DRAM BW 10GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>100MHz</td>
<td>43.8</td>
<td>42.6</td>
<td>42.4</td>
<td>42.3</td>
<td>42.2</td>
<td></td>
</tr>
<tr>
<td>400MHz</td>
<td>14.62</td>
<td>11.89</td>
<td>10.84</td>
<td>10.67</td>
<td>10.65</td>
<td></td>
</tr>
<tr>
<td>800MHz</td>
<td>12.4</td>
<td>7.31</td>
<td>5.79</td>
<td>5.42</td>
<td>5.37</td>
<td></td>
</tr>
<tr>
<td>1000MHz</td>
<td>12.16</td>
<td>6.79</td>
<td>4.86</td>
<td>4.39</td>
<td>4.34</td>
<td></td>
</tr>
</tbody>
</table>

**GoogleNet (~3.2 GOP, 7M weights)**
- Small filter size (1x1)
- Benefit parallelism in CNN operations
- Computation power dominates
- DRAM speed cannot help

<table>
<thead>
<tr>
<th>Core Speed</th>
<th>Latency (ms)</th>
<th>DRAM BW 1GB/s</th>
<th>DRAM BW 2GB/s</th>
<th>DRAM BW 4GB/s</th>
<th>DRAM BW 8GB/s</th>
<th>DRAM BW 10GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>100MHz</td>
<td>193.2</td>
<td>153.1</td>
<td>137.6</td>
<td>132.1</td>
<td>131.9</td>
<td></td>
</tr>
<tr>
<td>200MHz</td>
<td>143.5</td>
<td>96.6</td>
<td>76.5</td>
<td>68.8</td>
<td>67.7</td>
<td></td>
</tr>
<tr>
<td>400MHz</td>
<td>129.8</td>
<td>71.8</td>
<td>48.3</td>
<td>38.3</td>
<td>36.7</td>
<td></td>
</tr>
<tr>
<td>800MHz</td>
<td>126.8</td>
<td>64.9</td>
<td>35.9</td>
<td>24.2</td>
<td>22.1</td>
<td></td>
</tr>
<tr>
<td>1000MHz</td>
<td>126.3</td>
<td>64.3</td>
<td>34.3</td>
<td>21.6</td>
<td>19.3</td>
<td></td>
</tr>
</tbody>
</table>

**ResNet50 (~7.8 GOP, 25M weights)**
- Large CNN operations, large weights
- Residual → directly add two data cubes → DRAM speed dominates
- Computation power and DRAM speed are evenly important
Deep Learning is a kind of Heterogeneous Computing.
Target on Heterogeneous Architecture

- CPU
- GPU
- DSP
- DLA

traditional compiler

<table>
<thead>
<tr>
<th>target</th>
<th>heterogeneous architecture system (HSA)</th>
<th>single architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical feature depends on</td>
<td>operand</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- CPU: flexible
- GPU: effective
Main stream AI compiler
from Cloud to Edge
AI Technologies – Open Source & Community Driven

• **AI Networks**
  – ONNX (Open Neural Network Exchange Format), [https://onnx.ai/](https://onnx.ai/)
  – The new open ecosystem for interchangeable AI models
  – Amazon, Facebook, Microsoft, Baidu, Alibaba, Tencent, ..., etc.

• **Open AI Compiler**
  – Glow: A community-driven approach to AI infrastructure ([Facebook.ai](https://facebook.ai))
  – ONNC: Open Neural Network Compiler ([https://onnc.ai/](https://onnc.ai/)), collection of compiler and AI toolchains for ONNX-based DLA (Skymizer)

• **Open AI Hardware**
  – CHIP Alliance, [https://chipsalliance.org/](https://chipsalliance.org/)
  – Google, Western Digital and SiFive
# ONNC Internals

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-10:30</td>
<td>ONNC Overview</td>
</tr>
<tr>
<td></td>
<td>ONNC Logistic Layers</td>
</tr>
<tr>
<td></td>
<td>ONNC Intermediate Representation</td>
</tr>
<tr>
<td>10:45-12:15</td>
<td>ONNC Pass Management</td>
</tr>
<tr>
<td></td>
<td>ONNC Target Backend</td>
</tr>
</tbody>
</table>

**Luba Tang** <luba@skymizer.com>
CEO & Founder of Skymizer Inc.
Architect of ONNC, MCLinker, and GYM compiler
Compiler and Linker/Electronic System Level Design
High Level Concept of the Architecture Structure

- **Tooling Layer**
  - Compiler Driver
  - Arch Explorer
  - ONNX Reader

- **Compiler Layer**
  - IR
  - Target Machine
  - Pass Mngr

- **Logistic Layer**
  - JSON
  - Diagnostic
  - Support
  - ADT

- **DevOp Layer (Umbrella)**
  - Regression
  - Quick Regression
  - CI
  - Building System
  - Unittest

- **3rd party**
  - LLVM
  - ONNX
ONNC Logistic Layers

- General Routines
- ADT (Abstract Data Type)
  - StringRef
  - Rope
  - IList
  - Flag
  - BinaryTree
- JSON (JSON format read/write/access)
- Support (System, Memory and I/O)
  - IOStream
  - MemoryPool
  - ManagedStatic
  - DataTypes
- Diagnostic (error message handling)
Dev-defined error vs System-defined error

• System-defined error
  – defined by operating system
  – For example, EBUSY, ENOSYS

• An error defined by developer
  – Most defined by compilers
  – For example, Success, NotStartedYet, UnknownError

• All errors that happens in Linux, Mac OS X, FreeBSD and Windows are listed in `<Support/ErrorCodes.h>`
class SystemError

• To encapsulate all errors, we provide SystemError class
• sizeof(SystemError) == sizeof(int)
• You can use it with std::ostream and compare operators

```cpp
#include <onnc/Support/ErrorCode.h>
#include <onnc/Support/IOStream.h>
#include <errno.h>

using namespace onnc;

int fd = open("/", O_WRONLY);
SystemError err(errno); // get a copy from errno
if (!err.isGood())
    errs() << err; // permission deny
```
SystemError is the standard error handler in ONNC

- Most interfaces in Supports use SystemError
  - FileHandle::open
Diagnostic - ONNC specific exception handler

- Most compilers don’t introduce exception and RTTI (runtime type information) in their building system
  - exception slows down the performance of compiler
  - RTTI introduces uncontrollable exceptions
- Most compilers provide her own exception handling system.
- ONNC provides Diagnostic system as its own exception handling system
When to use diagnostics

- If the **end users** should read the message, then we use diagnostics
- FAE follows the lead of the messages returned by ONNC
- FAE can not follow segmentation fault. We should **do our utmost to avoid segmentation fault**.
- You can keep using naïve `printf` only if you are
  - making a debugging message by your own, and
  - you won’t submit it into master
Three modes: Normal/Verbose/Engineering/Noisy

- Most tools in ONNC have three modes: normal, engineering and noisy
- Users can turn on verbose mode by giving one `-v` (verbose)
- Engineering mode by giving more than three `-v`
- Noisy mode by giving more than five `-v`

```
$ onnc -h # normal mode
$ onnc -h -v # verbose mode
$ onnc -h -v -v -v # engineering mode
$ onnc -h -v -v -v -v # noisy mode
```
## Error Levels

- **include `<onnc/Diagnostic/MsgHandling.h>`**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Handler</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal (silent)</td>
<td>unreachable</td>
<td>Impossible error</td>
</tr>
<tr>
<td></td>
<td>fatal</td>
<td>Serious error. ONNC stop immediately</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>Normal error. ONNC will try to run further paces.</td>
</tr>
<tr>
<td>verbose</td>
<td>warning</td>
<td>warning</td>
</tr>
<tr>
<td>engineering</td>
<td>debug</td>
<td>You can see the debugging message</td>
</tr>
<tr>
<td></td>
<td>note</td>
<td>You can see the notes for debugging</td>
</tr>
<tr>
<td>noisy</td>
<td>ignore</td>
<td>Show you every thing</td>
</tr>
</tbody>
</table>
Examples

```cpp
#include <skymizer/Diagnostic/MsgHandling.h>
using namespace skymizer;

if (I got a serious problem)
    fatal(fatal_open_folder) << "this folder" << 2;
```

- Format
  ```cpp
  handler(ID) << msg0 << msg1 << .. << msg9;
  ```

- There are most `10` messages in one print
  - If you put more than 10 messages, you shall get a compilation error when you're building ONNC.

- Messages can be one of the types:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Header File</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool/int32_t/int64_t</td>
<td>Support/DataTypes</td>
</tr>
<tr>
<td>char*/std::string/onnc::StringRef</td>
<td>ADT/StringRef.h</td>
</tr>
<tr>
<td>Path</td>
<td>Support/Path.h</td>
</tr>
<tr>
<td>onnc::SystemError</td>
<td>Support/ErrorCode.h</td>
</tr>
</tbody>
</table>
Add a new error handler

- Add a handler in `include/onnc/Diagnostics/DiagCommonKinds.inc`
  - `DIAG(tag, error level, message)`
- Message format

```
DIAG(fatal_open_folder, Fatal, "cannot open the folder `%0`. (Code: %1)"
```

```
#include <skymizer/Diagnostic/MsgHandling.h>
using namespace skymizer;

fatal(fatal_open_folder) << “this folder” << 2;
```
## ONNC Internals

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-10:30</td>
<td>ONNC Overview</td>
</tr>
<tr>
<td></td>
<td>ONNC Logistic Layers</td>
</tr>
<tr>
<td></td>
<td>ONNC Intermediate Representation</td>
</tr>
<tr>
<td>10:45-12:15</td>
<td>ONNC Pass Management</td>
</tr>
<tr>
<td></td>
<td>ONNC Target Backend</td>
</tr>
</tbody>
</table>

**Luba Tang**  
<luba@skymizer.com>  
CEO & Founder of Skymizer Inc.  
Architect of ONNC, MCLinker, and GYM compiler  
Compiler and Linker/Electronic System Level Design
ONNC Intermediate Representation

- Basic concept of ONNC lowering
- Basic concept of ONNX IR
- Basic concept of ONNC IR
- Tutorial to ONNC IRBuilder
ONNC IR: The heart of ONNC

- Core design thought - *from network domain to compute unit*
- Four phases in the compilation process
  - IRReader - read ONNX prototex and build ONNX IR
  - TensorSel - select corresponding instruction for target devices
  - MemAlloc - turn symbolic operands into memory address
    - instruction scheduling
    - memory partition
    - memory allocation
  - CodeEmit - emit binary code for target devices
Two Levels of IR: Graph IR and Compute IR

- **Graph IR**
  - original ONNX IR
  - used to represent ONNX models

- **Compute IR**
  - ONNC IR
  - used to add hardware information on

- ONNX allows multiple subgraphs in one model
- ONNC allows multiple subgraphs in one model, too
Module - The façade of all IRs

- In ONNC, a module represents a single unit of network that is to be processed together
- A module contains all instances of graph IR and compute IR.
- Pass developers and target backend developers handle with module all the time

```cpp
// include/onnc/IR/Module.h

class Module
{
public:
  xGraph* getRootTensorGraph();
  ComputeGraph* getRootComputeGraph();

  tg_iterator tgBegin(); // used to traverse all tensor graphs
  tg_iterator tgEnd();
  cg_iterator cgBegin(); // used to traverse all compute graphs
  cg_iterator cgEnd();
};
```
Read data from ONNX file

- `onnc::onnx::Reader` can read ONNX file and put data in a module

```cpp
#include <onnc/IRReader/ONNXReader.h>
#include <onnc/IR/Module.h>
#include <onnc/Support/Path.h>
#include <onnc/Support/IOStream.h>
using namespace onnc;

void main()
{
    Module module;
    onnc::onnx::Reader reader;
    reader.parse("my path/bvlc_alexnet.model", module);
    module.print(outs()); // print module information
}
```
The core of compiler - Use-Define Chain

- X1 layer defines a new value of tensor w1
- Conv layer uses three tensors w1, w2, w3
- The relationships between use and define forms a chain - Use-Define Chain
Conceptual data structure of a Use-Define Chain

- Operators are definers. They have a pointer to their output value.
- Operators are also users. They have pointers to their input values.
- Input value and output value is the same object.
Graph IR - defined by ONNX

- Optimizing passes shall change semantics in compute IR, not graph IR.
- Developers read graph IR to know the original semantics of the neural network.
- Because ONNX IR is still changing, ONNC has to re-define all ONNX data structure in onnc namespace with `x` prefix.
  - onnx::Node -> onnc::xNode
- You can see the definitions in onnc/Config/ONNX.h.in
Data structure of a Use-Define Chain in ONNC

- To chain users and definers, ONNC provides a special data structure called `Use` to point out users and its value.
- Since in Neural Network, every value has only one definer, we bookkeep definers in value.
ONNX IR with ONNC IR

```
onnx::Node
  Inputs <*>
  Operator
  Outputs <*>

value

onnx::Node
  inputs
  Operator
  Outputs <*>

ComputeOperand

ComputeOperator
  Inputs <*>
  Operator
  Outputs <*>

value

ComputeOperator
  inputs
  Operator
  Outputs <*>

ComputeOperand
```

36
IRBuilder

IRBuilder encapsulates creating process of a module.

- **Handle with ONNX IR**
  - create a new ONNX graph
  - create/clnoe ONNX layers
  - create ONNX input tensors
  - create ONNX output tensors

- **Handle with compute IR**
  - create a new compute graph
  - create/clone layers
  - create input layers
  - create output layers

- see src/tools/unittests/ComputeIRTest.cpp
Some notes for using ONNX Graph IR

1. Compute IR is smaller
   - ONNX IR keeps data in a tensor even we don’t need it. That is, ONNX IR may be fat.
   - ONNC compute IR will use symbolic representation instead of keeping data.

2. It’s much easier to find input/output tensors in compute IR
   - ONNX IR doesn’t provide initializer operator for the initial inputs, developers must find initial inputs by names.
   - ONNC IR provides *initializer/output operator* and it reduces a lot of works in optimization algorithm.

3. It’s much easier to find subgraph in compute IR
   - ONNX IR uses node attributes to keep subgraphs. If you want to list all subgraphs, you must traverse the whole nodes and edges.
   - ONNC module contains multiple subgraphs of compute IR and ONNX IR.

---

**ONNX IR**

```
   i1   X1      w1       o1
     |         |         |
     |         |         |
     v         v         v
   i2   X2      w2
   |         |         |
   |         |         |
   v         v         v
  S   o2       Y
```

**Compute IR**

```
   i1   X1      w1       O1
     |         |         |
     |         |         |
     v         v         v
   i2   X2      w2
   |         |         |
   |         |         |
   v         v         v
  S   O2
```

**Module**

```
   i1   X1      w1       O1
     |         |         |
     |         |         |
     v         v         v
   i2   X2      w2
   |         |         |
   |         |         |
   v         v         v
  S   Y   O2
```
# ONNC Internals

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-10:30</td>
<td>ONNC Overview</td>
</tr>
<tr>
<td></td>
<td>ONNC Logistic Layers</td>
</tr>
<tr>
<td></td>
<td>ONNC Intermediate Representation</td>
</tr>
<tr>
<td>10:45-12:15</td>
<td>ONNC Pass Management</td>
</tr>
<tr>
<td></td>
<td>ONNC Target Backend</td>
</tr>
</tbody>
</table>

**Luba Tang**<luba@skymizer.com>

CEO & Founder of Skymizer Inc.
Architect of ONNC, MCLinker, and GYM compiler
Compiler and Linker/Electronic System Level Design
ONNC Optimization Flows

- There are two kinds of optimization algorithms for neural network
  - Graph-level optimization
  - Vector-level optimization

- Graph-level optimization handles with matrices
  - Separate a matrix into pieces
  - Merge several matrices into big one
  - Set the order of matrix multiplications

- Vector-level optimization handles with vectors
  - Reorder the cross products of vectors
ONNC Optimization Flows

Graph-level optimization

Liveness analysis

Layer scheduling

Memory allocation

SW decomposition

vector-level optimization

Dead code elimination

Loop optimization
Liveness analysis of tensors

- Find out the live range of every tensor
- Leverage use-define chain of ONNX
- By the help of simple liveness analysis, we can reuse local memory and eliminate $\frac{1}{2}$ memory consumption with greedy allocation

$$W1 = X1()$$
$$W2 = X2()$$
$$W3 = X3()$$
$$W4 = S(W1, W2, W3)$$
$$Y(W4)$$
Layer Scheduling

- If size $W_2 > W_1 > W_3$, then we can reorder $X_1 X_2 X_3$ to reduce the memory consumption.

$$W_3 = X_3()$$
$$W_1 = X_1()$$
$$W_2 = X_2()$$
$$W_4 = S(W_1, W_2, W_3)$$
$$Y(W_4)$$
Memory Allocation

- Memory allocation is used to allocate memory for each layer.
- Layer Scheduling affects the results of memory allocation.
Layer Splitting - Handle the memory limit
Layer Fusion

• Weight stationary and output stationary architectures usually have dedicated element-wise function unit.
• If we can leverage the element-wise function unit, then we can save data movement from outside to the inside core.
Memory runtime consumption

Minimal usage of activation memory. Save 46% than the next best solution

bundle mode with paging system

The details will be published in AICAS’19
http://www.aicas2019.org
Invited paper
Observation: Fragmentation at The Boundaries

At time $t$, both the top and bottom memory space have free space for allocation.
Best-Fit With Heuristics

resnet50
Best-Fit with Heuristic.
Best-Fit With Heuristics & Layer Reordering

resnet50

Best-Fit with Heuristic & Reordering.
Near-optimal results:
ONNC Best-Fit with Heuristic and Reordering
vector level optimization
Four Kinds of Passes in ONNC

- **ModulePass**
  - The most general of all superclasses that you can use
  - Use entire network as an unit
- **TensorPass**
  - Use Tensor Graph as an unit
  - Tensor Graph bases on ONNX IR
- **RegionPass**
  - Use each single-entry-single-exit region in a tensor graph as an unit
  - For example, groups in GoogLeNet
- **ComputePass**
  - Use Compute Graph as an unit

```cpp
// methods in class Pass
bool run(Module& pModule);
virtual bool doInitialization(Module& pModule);
virtual bool doFinalization(Module& pModule);

// methods in class ModulePass
virtual bool runOnModule(Module& pModule) = 0;

// methods in class TensorPass
virtual bool runOnTensor(TensorGraph& pGraph) = 0;
```
AnalysisUsage describes the dependencies between Passes

- PassManager automatically creates all Passes that used by the other Passes.
- Similar to LLVM. Engineers who already familiar to LLVM can understand ONNC in a short time

```cpp
/// in A.cpp
INITIALIZE_PASS(A, “pass_a”)  
GET_ANALYSIS_USAGE(AnalysisUsage& pUsage)
const
{
  pUsage.addRequiredID(A::ID);
  pUsage.addRequiredID(B::ID);
}
```
Write a Simple Dead Code Elimination Pass

• *Dropout layers* are used in training, not inference
• ONNC can safely remove *Dropout* layers
• In this tutorial, we will develop a simple dead code elimination algorithm - remove all dropout in the graph

• See *lib/Transforms/RemoveTrainingNodes.cpp*
# ONNC Internals

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-10:30</td>
<td>ONNC Overview</td>
</tr>
<tr>
<td></td>
<td>ONNC Logistic Layers</td>
</tr>
<tr>
<td></td>
<td>ONNC Intermediate Representation</td>
</tr>
<tr>
<td>10:45-12:15</td>
<td>ONNC Pass Management</td>
</tr>
<tr>
<td></td>
<td>ONNC Target Backend</td>
</tr>
</tbody>
</table>

### Luba Tang  <luba@skymizer.com>

CEO & Founder of Skymizer Inc.
Architect of ONNC, MCLinker, and GYM compiler
Compiler and Linker/Electronic System Level Design
Quadruple - new way to select a NN compute unit

- Quadruple is a string representation that represents
  - Target hardware architecture (micro-architecture, ISA, etc.)
  - Target software environment (ABI, OS, etc.)
  - Target tool (compiler, loader, calibration, etc.)

- for example,
  - LLVM triple: arm-nonono-linux-gnueabi
  - ONNC quadruple: arm-none-linux-gnueabi-calibration-0.2

LLVM triple = HW x SW
ONNC quadruple = HW x SW x Tool
ONNC supports various target devices

- Use LLVM-like triple to select compiler target backend
  - compiler
  - loader
  - calibration
- Every Target instance represents a target device
  - contains cost model
  - contains target-dependent passes
- Target instance registers target-dependent passes into PassManager
TargetBackend Controls The Phases of Lowering

```
// compiler bone
PassManager pm;
TargetBackend* backend = TargetRegistry::Lookup("DLA-A");
backend->addTensorSel(pm);
backend->addMemAlloc(pm);
backend->addCodeEmit(pm);
pm.run();
```

```
// Core/TargetBackend.h
class TargetBackend {
  virtual void addTensorSel(PassManager& pPM) { return; }
  virtual void addMemAlloc (PassManager& pPM) { return; }
  virtual void addCodeEmit (PassManager& pPM) { return; }
};
```

```
// ATargetBackend.cpp
void ABackend::addCodeEmit(PassManager& pPM)
{
  pPM.add(createRemoveUnusedNodePass());
  pPM.add(createUpdateOutputInfoPass());
  pPM.add(createTGMemAllocInfoPass(this));
  pPM.add(createTargetLoweringPass(this));
  pPM.add(createTGCodeEmitPass(this));
}
ONNC System Software
46% less memory consumption
https://onnc.ai

Architecture Design Automation on NVDLA
open source hardware
https://nvdla.org
Open Source Project with Commercial Optimization

### Optimizing pass | Meaning
--- | ---
Layer Splitting | Split one layer into pieces to open opportunities of memory allocation and tensor scheduling
Memory Allocation | To reuse local memory of DLA, to save memory usage
Dead Code Elimination | Remove unused layers
Common Subexpression Elimination | Combine tensors
Layer Fusing | Fuse layers
Tensor Scheduling | Schedule layers to adjust life-range of tensors

Near-optimal memory allocation

4% gap to theoretical optimal result
NVDLA
Reference Design

Original NVDLA opens hardware, but its software is not
ONNC is the first open source compiler supports NVDLA

- **QEMU-based virtual platform**
  - Generic and open source machine emulator and virtualizer

- **Virtual modeling for CPU and DLA**
  - Cortex-A processor
  - RISC-V processor
  - NVDLA with difference configuration

- **Performance Analysis Kit**
  - ONNC-based SDK
  - Running popular ML framework
  - Support debug and performance monitoring
NVDLA System Architecture

- NVDLA needs an embedded CPU running Linux to leverage the NVDLA software stack.
- The headed implementation has an additional micro-controller aside the embedded CPU to offload the NVDLA control task.
ONNC-based software development platform is built to support various hardware design tradeoffs. Explore the hardware/software co-design and optimize at the system level.
ONNC Compiler Porting (1/2)

- Official NVDLA compiler is released in the **binary form** and it only supports limited operators and models
- NVDLA **Linux drivers**, UMD and KMD, are released with source code and exist as defined APIs.
- Successfully demystify the Loadable file format and the NVDLA register specification
- The **NVDLA backend** in ONNC compiler compiles a model into NVDLA Loadable file
- ONNC supports more ONNX models than the official NVDLA compiler
- ONNC can compile **6 models** and run on the NVDLA virtual platform successfully
Compiler Optimization – Performance Evaluation

• Table compares a couple of performance metrics to run an Alexnet inference for two optimization schemes
• The SystemC time shows roughly 25% performance improvement for layer fusion
• Without hardware calibration, those metrics only provide relative referencing for discussion
• Regarding the first-layer convolution mode is that nv_small outperforms nv_full in the direct mode because more than 80% of zero-padding in the first layer is required for a 3-channel image input

Table 2: Performance Evaluation of bvlc-alexnet.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Time</th>
<th>nv_full</th>
<th>nv_small</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer fusion</td>
<td>Off</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>SystemC time (ms)</td>
<td>4900</td>
<td>3800</td>
<td>5100</td>
</tr>
<tr>
<td>Guest OS time (sec)</td>
<td>4164</td>
<td>4079</td>
<td>4796</td>
</tr>
<tr>
<td>Hardware layers</td>
<td>42</td>
<td>36</td>
<td>44</td>
</tr>
</tbody>
</table>

| First-layer convolution       | Mode                      | Direct  | Image    | Direct   | Image   |
| SystemC time (ms)             | 4100                      | 3400    | 3900     | 3700     |
| Guest OS time (sec)           | 3996                      | 1786    | 4503     | 3971     |
More information to get starts

${src}/docs/ONNC-Utilities.md

https://onnc.ai

Thanks You!