SOL: Transparent Neural Network Acceleration on NEC SX-Aurora TSUBASA

Dr. Nicolas Weber (NEC Labs Europe)
Where to start?
Integration into existing frameworks is expensive

Each framework has its own internal and external APIs

- No common code base
- Approaches such as MLIR, ONNX, DLPack, ... not widely adopted or very limited
Integration into existing frameworks is expensive

MLIR - a common intermediate representation (IR)

PeterCDMcLean

MLIR: https://github.com/tensorflow/mlir

MLIR's intention seems to be an IR lowering framework. In my opinion, this has great synergy with the multiple levels of IR that Glow currently provides.

Does Glow have any intention / interest of integration or use of MLIR?

Jfzx Jordan Fix

If Peter, we don't have any plans for MLIR for now. It could make sense to load MLIR into Glow (converting MLIR into Glow IR), which would allow us to use Glow's optimisation stack and target any of our backends. Did you have something in particular in mind for Glow + MLIR?
Integration into existing frameworks is expensive

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226

Any plans to support MLIR #1226
Integration into existing frameworks is expensive

Any plan...
Integration into existing frameworks is expensive

3 line bugfix took two months to be released!
The SOL-Project

**SOL is a full stack AI acceleration middleware**
- Add-on to AI frameworks that does not require any code changes to the framework
- Optimizations range from mathematical/algorithmic down to actual implementations/code generation
What data scientists see:

\[ x = \text{Conv}(x, \text{kernel}=1\times1, \text{bias}=\text{True}) \]
\[ x = \text{ReLU}(x) \]
\[ x = \text{AvgPooling}(x, \text{kernel}=13\times13) \]
SOL in a nutshell

**What data scientists see:**

\[ x = \text{Conv}(x, \text{kernel}=1\times1, \text{bias}=\text{True}) \]
\[ x = \text{ReLU}(x) \]
\[ x = \text{AvgPooling}(x, \text{kernel}=13\times13) \]
SOL in a nutshell

**What data scientists see:**
\[ x = \text{Conv}(x, \text{kernel}=1\times1, \text{bias}=\text{True}) \]
\[ x = \text{ReLU}(x) \]
\[ x = \text{AvgPooling}(x, \text{kernel}=13\times13) \]

**What HPC people see:**

```python
function(Conv):
    for (Batch, OutChannel, Y, X):
        for (InChannel, KernelY, KernelX):
            output[...] += input[...] * weight[...]
            output[...] += bias[...]

function(ReLU):
    for (Batch, OutChannel, Y, X):
        output[...] = \text{max}(0, input[...])

function(AvgPooling):
    for (Batch, OutChannel, Y, X):
        for (KernelY, KernelX):
            output[...] += input[...] / (13\times13)
```

© NEC Corporation 2019
SOL in a nutshell (continued)

What we actually want:

function(FusedNetwork):
    for(Batch, OutChannel):
        float N[…]
        for(Y, X):
            for(InChannel, KernelY, KernelX):
                N[…] += input[…] * weight[…]  
                N[…] += bias[…]
                N[…] = max(0, X)
            for(Y, X):
                for(KernelY, KernelX):
                    output[…] += N[…] / (13*13)
__global__ void F64486B08(...) {
    const int O0idx = omp_get_thread_num();
    const int O0 = O0idx / 256;
    const int O1 = O0idx % 256;
    float T64[169];
    #pragma _NEC ivdep
    for(int O2idx = 0; O2Idx < 169; O2Idx++) {
        float T63 = 0.0f;
        for(int I1 = 0; I1 < 512; I1++) { // #1 Convolution: 1x1 Pooling
            T63 += T61[O0 * 86528 + I1 * 169 + O2idx] * P63_weight[O1 * 512 + I1];
            T63 = (T63 + P63_bias[O1]); // #1 Convolution: Bias
        }
        T64[O2Idx] = sol_ncc_max(T63, 0.0f); // #2 ReLU
    }
    T66[O1] = sol_ncc_reduce_add(T64); // #3 AvgPooling: 13x13 Pooling
    T66[O1] = (T66[O1] / 169.0f); // #3 AvgPooling: Normalization
}
import torch
from torchvision import models

py_model = models.__dict__["..."]()
input = torch.rand(1, 32, 224, 224)
output = py_model(input)
import torch
def classify_image(model, image):
    output = model(image)
    return output

classify_image(py_model, input)
How SOL integrates into the frameworks?

| SOL injects its optimized code as custom model into the framework |

```python
class SolLayer(torch.nn.Module):
    def __init__(self):
        self.ParamA = ...
        self.ParamB = ...

    def forward(self, X):
        return sol.run(X, self.ParamA, self.ParamB)
```

**framework handles model parameters!**

**SOL handles execution**
Training Performance (CNN BS=16, MLP BS=64, FP32)

Xeon 6126 - PyTorch 1.4
Titan V - PyTorch 1.4
SX-Aurora - SOL
Training Performance (CNN BS=16, MLP BS=64, FP32)

<table>
<thead>
<tr>
<th>Model</th>
<th>Training Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Densenet</td>
<td>161</td>
</tr>
<tr>
<td>Resnet</td>
<td>50</td>
</tr>
<tr>
<td>v2 2.0</td>
<td>8192x3</td>
</tr>
<tr>
<td>ShuffleNet</td>
<td>1.0</td>
</tr>
<tr>
<td>Squeezenet</td>
<td>8192x3</td>
</tr>
<tr>
<td>BN 11</td>
<td>8192x3</td>
</tr>
<tr>
<td>VGG</td>
<td>8192x3</td>
</tr>
<tr>
<td>MLP</td>
<td>8192x3</td>
</tr>
</tbody>
</table>

lower is better

Xeon 6126 - PyTorch 1.4
Titan V - PyTorch 1.4
SX-Aurora - SOL
Training Performance (CNN BS=16, MLP BS=64, FP32)

The graph shows the training time (ms) for various models and hardware configurations. The lower the time, the better the performance.

- **Densenet**: 161 ms
- **Resnet**: 50 ms
- **v2 2.0**: 11 ms
- **ShuffleNet**: 8192x3 ms
- **Squeezenet**: 1.0 ms
- **BN 11**: 1.0 ms
- **VGG**: 1.0 ms
- **8192x3**: 1.0 ms

- **Xeon 6126 - PyTorch 1.4**
- **Titan V - PyTorch 1.4**
- **SX-Aurora - SOL**

The bar chart compares the training times for different models and hardware configurations, with SX-Aurora being the most efficient in terms of training time.
Training Performance (CNN BS=16, MLP BS=64, FP32)

<table>
<thead>
<tr>
<th>Model</th>
<th>TFLOP/s</th>
<th>Ratio</th>
<th>GB/s</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Titan V</td>
<td>14.9</td>
<td>3.47</td>
<td>651.3</td>
<td>0.54</td>
</tr>
<tr>
<td>SX-Aurora</td>
<td>4.3</td>
<td></td>
<td>1200.0</td>
<td></td>
</tr>
</tbody>
</table>

Titan V: 14.9 TFLOP/s, 3.47 Ratio, 651.3 GB/s, 0.54 Ratio
SX-Aurora: 4.3 TFLOP/s, 1200.0 GB/s
Inference Performance (BS=1, FP32)

- **SX-Aurora**
- **Inference Time (ms)**

Xeon 6126 - PyTorch 1.4
Titan V - PyTorch 1.4
SX-Aurora - SOL

161
50
v2 2.0
1.0
BN 11
8192x3

lower is better
Inference Performance (BS=1, FP32)

lower is better

0 20 40 60 80 100 120
Inference Time (ms)

SX-Aurora

161
Densenet

50
Resnet

v2 2.0
ShuffleNet

1.0
Squeezenet

BN 11
VGG

8192x3
MLP

Xeon 6126 - PyTorch 1.4
Titan V - PyTorch 1.4
SX-Aurora - SOL
Inference Performance (BS=1, FP32)

- **Densenet**: 161 ms
- **Resnet**: 50 ms
- **v2 2.0**: 20 ms
- **ShuffleNet**: 18 ms
- **Squeezenet**: 16 ms
- **BN 11**: 8 ms
- **8192x3**: 4 ms

*Lower is better.*
Inference Performance (BS=1, FP32)

<table>
<thead>
<tr>
<th>Model</th>
<th>TFLOP/s</th>
<th>Ratio</th>
<th>GB/s</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Titan V</td>
<td>14.9</td>
<td>3.47</td>
<td>651.3</td>
<td>0.54</td>
</tr>
<tr>
<td>SX-Aurora</td>
<td>4.3</td>
<td></td>
<td>1200.0</td>
<td></td>
</tr>
</tbody>
</table>

# Lower is better
How to use DNN in my own software?

Again, dozen of available tools...

- TF-Lite
- LibTorch
- ONNXRuntime
- OpenVino (only Intel)
- NGraph
- TVM
- TensorRT (only NVIDIA)
- SOL
- ...
How to use DNN in my own software?

```c
sol.deploy(trained_model, [input],
    target=sol.deployment.shared_lib, device=sol.device.ve,
    lib_name="MyNetwork", func_name="predict", ...)

#ifndef __MyNetwork__
define __MyNetwork__

#ifndef __cplusplus
extern "C" {
#endif
def predict_init(const int deviceIdx);
int predict_seed(const int seed);
def predict (void* ctx, const float* input, float** output);

#ifndef __cplusplus
}
#endif
define __cplusplus
#endif
```

© NEC Corporation 2019
**Status Quo:**
- PyTorch and ONNX
- CNN, MLP, Transformer, ...
- Training, Inference, Deployment
- ...

SOL RoadMap
SOL RoadMap: Tested Neural Networks

### Convolutional Neural Networks
- Alexnet
- SqueezeNet (1.0, 1.1)
- VGG + BN (11, 13, 16, 19)
- Resnet (18, 34, 50, 101, 152)
- Densenet (121, 161, 169, 201)
- Inception V3
- GoogleNet
- MobileNet (v1, v2)
- MNASNet (0.5, 0.75, 1.0, 1.3)
- ShuffleNet V2 (0.5, 1.0, 1.5, 2.0)
- ResNext (50, 101)
- WideResNet (50, 101)

### Multi Layer Perceptron (MLP)

### Linear/Logistic Regression

### Natural Language Processing
- BERT (PyTorchic + HuggingFace implementations)
- GPT-2 (in upcoming v0.3.0 release)
- LSTM+GRU (coming in Q4 2020)
SOL RoadMap

**Status Quo:**
- PyTorch and ONNX
- CNN, MLP, Transformer, ...
- Training, Inference, Deployment
  - ...

**2020:**
- DL4J (October)
- TensorFlow v2 (December)
- Recurrent Neural Networks (LSTM, GRU)
- torch.nn.DataParallel support for PyTorch

**2021:**
- Adjustable memory consumption during training (trading memory vs performance)
- User defined Custom Layers
- Algorithmic and internal code optimizations to improve performance
- NumPY support
Frovedis

presented by Dr. Erich Focht, NEC-D
Basics on SOL
How to install

- pip3 install sol-0.2.7.2-py3-none-any.whl
  - enforces installation of dependencies

Coming in v0.3.0

- pip3 install sol-0.3.0-py3-none-any.whl[torch, onnx]
  - optional installation of dependencies (i.e. if you do not need support for all frameworks, etc.)
## SOL Vocabular

<table>
<thead>
<tr>
<th>Rest of the World</th>
<th>SOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer</td>
<td>Layer</td>
</tr>
<tr>
<td>Tensor</td>
<td>Tensor</td>
</tr>
<tr>
<td>Model/Neural Network</td>
<td>Model</td>
</tr>
<tr>
<td>Fused Layers</td>
<td>Cluster</td>
</tr>
<tr>
<td>Framework</td>
<td>Frontend</td>
</tr>
<tr>
<td>Device</td>
<td>Device</td>
</tr>
<tr>
<td>Compute Library/Compiler</td>
<td>Backend</td>
</tr>
</tbody>
</table>
SOL Interface

Importing SOL:

- import sol.pytorch as sol
SOL Devices

```python
sol.devices()
```

SOL Device Dump:
X86 CPUs
 *[x86:0] Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz, 12 cores
NEC SX-Aurora Vector Engine
 *[ve:0] NEC SX-Aurora Tsubasa VE101, Firmware: 5399, 8 cores

star indicates default device
activated device
currently used memory
SOL Versions

sol.versions()

SOL Version Dump:

- AVEO: 0.9.12
- DNNL: 1.6.0
- GraphVIZ: 2.30.1
- ISPC: 1.14.1
- Linux: CentOS Linux 7 (Core), 3.10.0-1127.13.1.el7.x86_64
- MKL: 2020.0.1
- NEC NAR: 2.26.20160125
- NEC NC++: 3.0.28
- NEC NLD: 2.26.20160125
- NNPACK: bundled
- OneTBB: 2020_U3
- PyTorch: 1.6.0
- Python: 3.6.9
- SOL: 0.3.0, Betelgeuse
- SQLite: 3.32.3
- VEASL: 2.1.0
- VEILAS: 2.1.0
- VEDA: linked: 0.9.3, loaded: 0.9.3
- VEDNN: bundled
- VEOS: 2.5.0
- X86 GCC AR: 2.30
- X86 GCC G++: 8.3.1
- X86 GCC GCC: 8.3.1
- X86 GCC LD: 2.30
SOL Seed

Print Seeds:
- `sol.seeds()`

3 Types of Seeds:
- Global (all devices)
- DeviceType (all devices of same type)
- Device (a specific device)

Get seed:
- `sol.seed(deviceType=None, deviceIdx=None)`
- `sol.seed(deviceType=sol.device.ve, deviceIdx=None)`
- `sol.seed(deviceType=sol.device.ve, deviceIdx=0)`

Set seed:
- `sol.set_seed(seed, deviceType=None, deviceIdx=None)`
- `sol.set_seed(seed, deviceType=sol.device.ve, deviceIdx=None)`
- `sol.set_seed(seed, deviceType=sol.device.ve, deviceIdx=0)"
Debugging

- `sol.config["compiler::name"] = "Prefix Used for Debugging Output"

- C/C++ device code generated in .sol/ve/source
  - Might not be obvious to read

- `sol.config["compiler::debug"] = True`
  - Compiles with debug symbols
  - Prints execution times of fused layers
  - Outputs visualized NN in .sol/debug/ subfolder
  - Requires: GraphViz (Dot)
Debugging

sol.config[“compiler::debug”] = True

- Prints execution times of fused layers

Index of /v0.2.7.2/.sol/ve/src/

```plaintext
ve_0EAE7ED7_ve4n FI 385 76.912 µs
ve_0EAE7ED7_ncc FI 38B 0.500 µs
ve_0EAE7ED7_ve4n FI 38E 0.029 µs
ve_0EAE7ED7_ncc FI 394 0.200 µs
ve_0EAE7ED7_ve4n FI 3A0 0.039 µs
ve_0EAE7ED7_ve4n FI 4B1 0.036 µs
ve_0EAE7ED7_ve4n FI 47E 0.087 µs
ve_0EAE7ED7_ncc FI 9A9 0.019 µs
ve_0EAE7ED7_ve4n FI 3BB 0.036 µs
ve_0EAE7ED7_ve4n FI 407 0.232 µs
ve_0EAE7ED7_ncc FI 3B2 0.023 µs
ve_0EAE7ED7_ncc FI 404 0.037 µs
ve_0EAE7ED7_ncc FI 3C4 0.022 µs
ve_0EAE7ED7_FI.o 0.017 µs
ve_0EAE7ED7_ve4n FI 3CD 0.017 µs
ve_0EAE7ED7_ve4n FI 3D9 0.037 µs
ve_0EAE7ED7_ve4n FI 48D 0.025 µs
ve_0EAE7ED7_ve4n FI 3D0 0.022 µs
ve_0EAE7ED7_ncc FI 48A 0.025 µs
ve_0EAE7ED7_ncc FI 3F2 0.017 µs
ve_0EAE7ED7_ve4n FI 493 0.025 µs
ve_0EAE7ED7_ncc FI 490 0.023 µs
ve_0EAE7ED7_ve4n FI 400 0.024 µs
ve_0EAE7ED7_ve4n FI 406 0.015 µs
ve_0EAE7ED7_ve4n FI 412 0.018 µs
ve_0EAE7ED7_ve4n FI 456 0.022 µs
ve_0EAE7ED7_ve4n FI 41B 0.015 µs
ve_0EAE7ED7_ve4n FI 42D 0.032 µs
```
sol.config["compiler::debug"] = True

- Outputs visualized NN in .sol/debug/ subfolder
sol.config["compiler::debug_memory_consumption"] = True

- Outputs memory consumption plots
- Requires: matplotlib
Debugging

1. `sol.config[“compiler::name”] = “Prefix Used for Debugging Output”`

2. C/C++ device code generated in .sol/ve/source
   - Might not be obvious to read

3. `sol.config[“compiler::debug”] = True`
   - Compiles with debug symbols
   - Prints execution times of fused layers
   - Outputs visualized NN in .sol/debug/ subfolder
   - Outputs memory consumption plots
   - Requires: matplotlib, GraphViz (Dot)

4. Activate tracing:
   - `sol.config[“log::level”] = sol.log.[error, info, warn, debug, trace]`
   - `SOL_LOG=TRACE python3 mySolScript.py`
SOL’s VE integration into PyTorch
SOL’s VE integration into PyTorch

- PyTorch does not come with support for storing data on VE devices.

- SOL adds this support into PyTorch automatically when loaded.

- We misuse the HIP-device for the VE’s as we can’t add new device types without recompiling PyTorch:
SOL’s VE integration into PyTorch

Identical to how CUDA is used in PyTorch, just with ‘hip’

Copy data to VE: tensor_ve = tensor_cpu.to('hip:0')
Copy data to CPU: tensor_cpu = tensor_ve.cpu() or .to('cpu')
Copy model to VE: model.to('hip:0')
Unfortunately tensor.hip() does not work :(

Synchronize VE execution:
- torch.hip.synchronize()

Selection of VE’s in Server
- export VEDA_VISIBLE_DEVICES=0,1,2
- export VEDA_VISIBLE_DEVICES=${VE_NODE_NUMBER}
Known Issues

- **torch.concat() on CPU can produce wrong results when SOL4VE is loaded**
  - Submitted bugfix to PyTorch, was released in PyTorch v1.6.0. SOL v0.3.0 will support PyTorch v1.6.0.

- **Only minimal number of functions implemented**
  - A + B, A – B, print(A), ...
  - Otherwise you will get a message like: “Function X not implemented for HipTensorId”.
  - Workaround:
    - A.cpu().notImplemented().to('hip:0')
  - CAN ONLY OCCUR OUTSIDE OF YOUR NEURAL NETWORK!!

- **print(tensor) always shows scientific notation.**
We finally want to use it!!!
PyTorch supports four execution modes, SOL only two:

<table>
<thead>
<tr>
<th>Function</th>
<th>SOL Inference</th>
<th>N/A</th>
<th>SOL Training</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>model.eval()</code></td>
<td>SOL Inference</td>
<td>N/A</td>
<td>SOL Training</td>
</tr>
<tr>
<td><code>torch.no_grad()</code></td>
<td>SOL Inference</td>
<td>N/A</td>
<td>SOL Training</td>
</tr>
<tr>
<td><code>model.training()</code></td>
<td>SOL Inference</td>
<td>N/A</td>
<td>SOL Training</td>
</tr>
</tbody>
</table>
Optimizing a model

```python
sol_model = sol.optimize(model, input0, input1, input2, ..., batch_size=32)
model = any torch.nn.Module
inputX
- torch.Tensor
- any primitive datatype (int, float, ...)
- sol.input([0, 3, 224, 224], requires_grad=False, dtype=torch.float)
  **Size of 0 is a wildcard (only in first dimension!)**
batch_size → needs to be set if wildcard is used, otherwise ignored. Is used by SOL in its heuristics.
```
import torch
import sol.pytorch as sol

class Model(torch.nn.Module):
    def forward(self, A, B):
        return A + B

py_model = Model()
sol.config[...] = ... # always set BEFORE sol.optimize
sol_model = sol.optimize(py_model, sol.input([0, 50]), sol.input([0, 50]), batch_size=32)
sol_model.load_state_dict(py_model.state_dict())
sol_model.to('hip:0')
Inference

# generate random input
A_cpu = torch.rand(5, 50)
B_cpu = torch.rand(5, 50)

# copy to VE
A_ve, B_ve = A_cpu.to('hip:0'), B_cpu.to('hip:0')

# activate inference mode
sol_model.eval()

with torch.no_grad():
    # run model
    C_ve = sol_model(A_ve, B_ve)
    # print result
    print(C_ve)
Training

sol_model.training()
for epoch in range(epochs):
    for batch in train_dataloader:
        # get batch and copy to VE
        A_cpu, B_cpu = *batch
        A_ve, B_ve = A_cpu.to('hip:0'), B_cpu.to('hip:0')

        # run forward pass
        C_ve = sol_model(A_ve, B_ve)

        # compute loss on CPU
        C_cpu = C_ve.cpu()
        loss = loss_function(C_cpu)

        # run backward pass
        loss.backward()

        # Optional: wait for VE to complete this iteration
        torch.hip.synchronize()
Known Issues/Pitfalls

“SQLITE Error UNIQUE CONSTRAINT …”

- SOL cache got corrupted. Either:
  - run: `rm -r .sol`
  - or call `sol.cache.clear()` before `sol.optimize(...)`

SOL does not complain when the model and the input data are not located on the same device:

- fixed in v0.3.0

`sol.deploy(...)` not fully working in v0.2.7.2. Would need some manual fixing in generated code.

- fixed in v0.3.0
More information in the SOL docs

This example requires the torchvision package: https://github.com/pytorch/vision/. Please note that SOL does not support the use of `model.eval()` or `model.train()`. SOL always assumes `model.eval()` for running inference, and `model.train()` when running training.

```python
import torch
import sol.pytorch as sol
import torchvision.models as models

class TrainingModel(torch.nn.Module):
    def __init__(self, model):
        super().__init__()
        self.n_model = model
        self.n_loss = torch.nn.L1Loss()

    def forward(self, x, y, z, target):
        output = self.n_model(x, y, z)
        loss = self.n_loss(output, target)
        return(output, loss)
```

---

© NEC Corporation 2019
# login to server
ssh hpc.icm.edu.pl
...

# install and activate virtualenv
pip3 install --user virtualenv
virtualenv sol
source sol/bin/activate

# install sol
pip3 install /apps/nec/sol/sol-0.2.7.2-py3-none-any.whl
pip3 install torchvision==0.6.1

# test sol
mkdir tmp
cd tmp
VEDA_VISIBLE_DEVICES=0 python3 /apps/nec/sol/test.py
How to get started on ICM