Orchestrating a brighter world

NEC

Integration of NEC SX-Aurora into AI Frameworks

Nicolas Weber - NEC Labs Europe

Erich Focht - NEC Deutschland

Motivation

Obvious: Everyone does AI today!

- AI-optimized Fridges, Microwaves, Toasters, T-800 Terminators, …

But where to start?

TensorFlow

Chainer

theano

Caffe2

mxnet

PyTorch

Integration into existing frameworks is expensive

Each framework has its own APIs

- Approaches such as MLIR, ONNX, DLPack, … not widely adopted or very limited

Device support tightly integrated into frameworks

- not portable between frameworks
- PyTorch alone has over 60.000 lines of code solely dedicated to NVIDIA GPUs!

1-2 major releases per framework per year

Upstreaming code is a time consuming and tedious task

Available options?

- The “Google”-way: hire 200 engineers
- **Be Smart™!**

The SOL-Project

SOL is a full stack AI acceleration framework

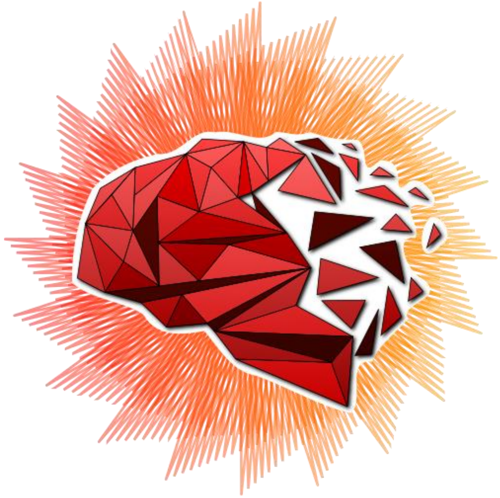
- Optimizations range from mathematical/algorithmic down to actual implementations/code generation
- Add-on to AI frameworks that does not require any code changes

Tightly integrates into existing frameworks

- TensorFlow
- PyTorch
- MxNet (in development)

Broad support for hardware architectures

- X86 CPUs
- NVIDIA GPUs
- ARM CPUs
- ARM GPUs (in development)
- AMD GPUs (in development)



SOL in a nutshell

What data scientists see:

```
x = Conv(x, kernel=1x1, bias=True)
x = ReLU(x)
x = AvgPooling(x, kernel=13x13)
```

What HPC people see:

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

function(ReLU):
    for(Batch, OutChannel, Y, X):
        output[…] = max(0, input[…])

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

SOL in a nutshell (continued)

- **All layers merged into a single kernel function, using specialized hardware features**

CUDA blocks

shared memory

CUDA cores

```
__global__ void F64486B08(...) {
  const int O0idx = blockIdx.x;
  const int O0 = O0idx / 256;
  const int O1 = O0idx % 256;
  __shared__ float T64[169];
  for(int O2idx = threadIdx.x; O2Idx < 169; O2Idx += 128) {
    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] = fmaxf(T63, 0.0f);            // #2 ReLU
  }
  T66[O1] = REDUCE_ADD(T64);                  // #3 AvgPooling: 13x13 Pooling
  T66[O1] = (T66[O1] / 169.0f);               // #3 AvgPooling: Normalization
}
```

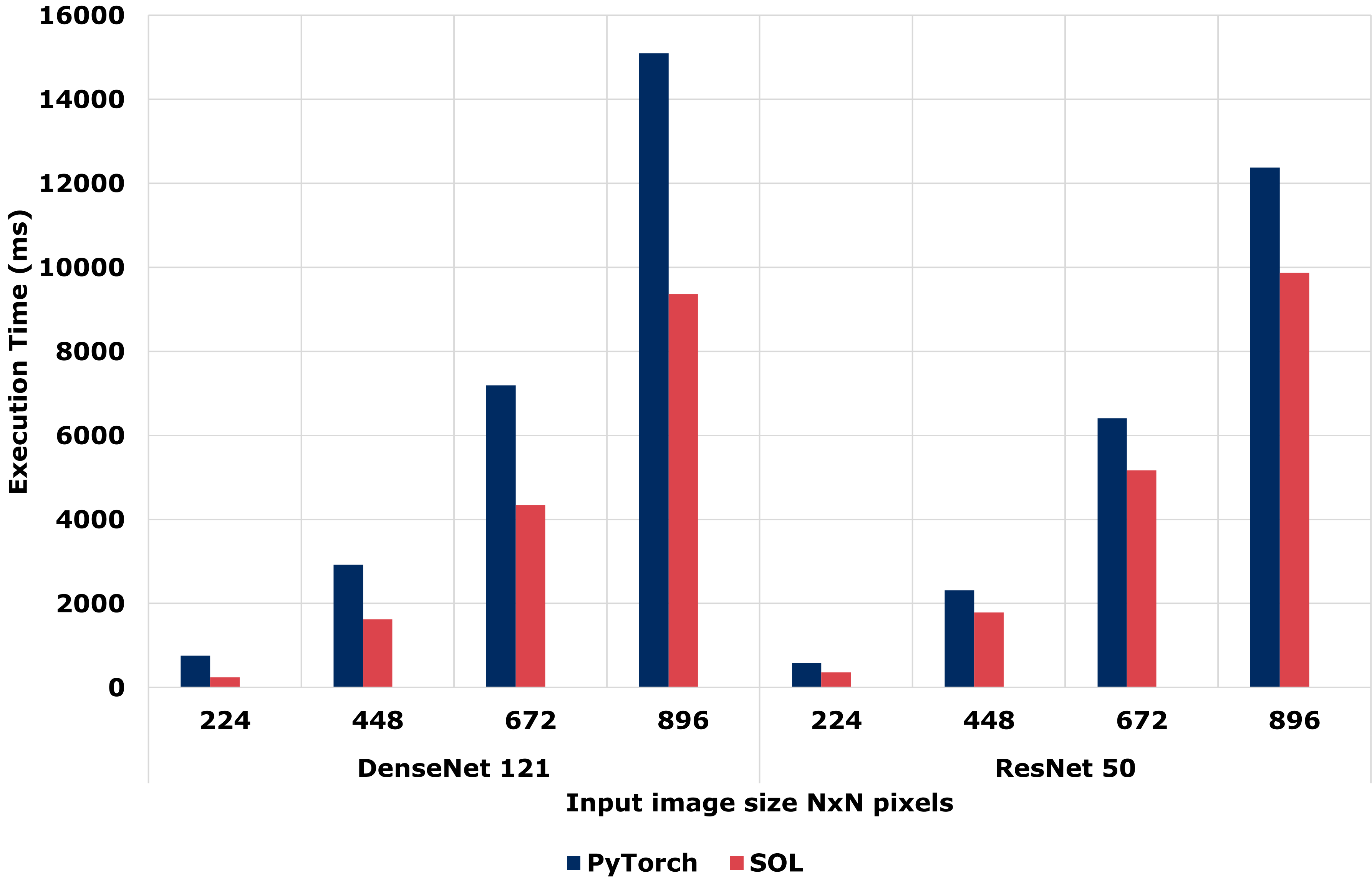
Reduction

SOL Usage (Pytorch)

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

py_model  = models.__dict__["…"]()
input     = torch.rand(32, 32, 224, 224)
sol_model = sol.optimize(py_model, input.size())
sol_model.load_state_dict(py_model.state_dict())
output = sol_model(input)
```

Performance Improvements on Xeon Gold 6126 (BS=8)



How to integrate SX-Aurora into the frameworks?

- SOL injects its optimized code as Custom Layer into the framework

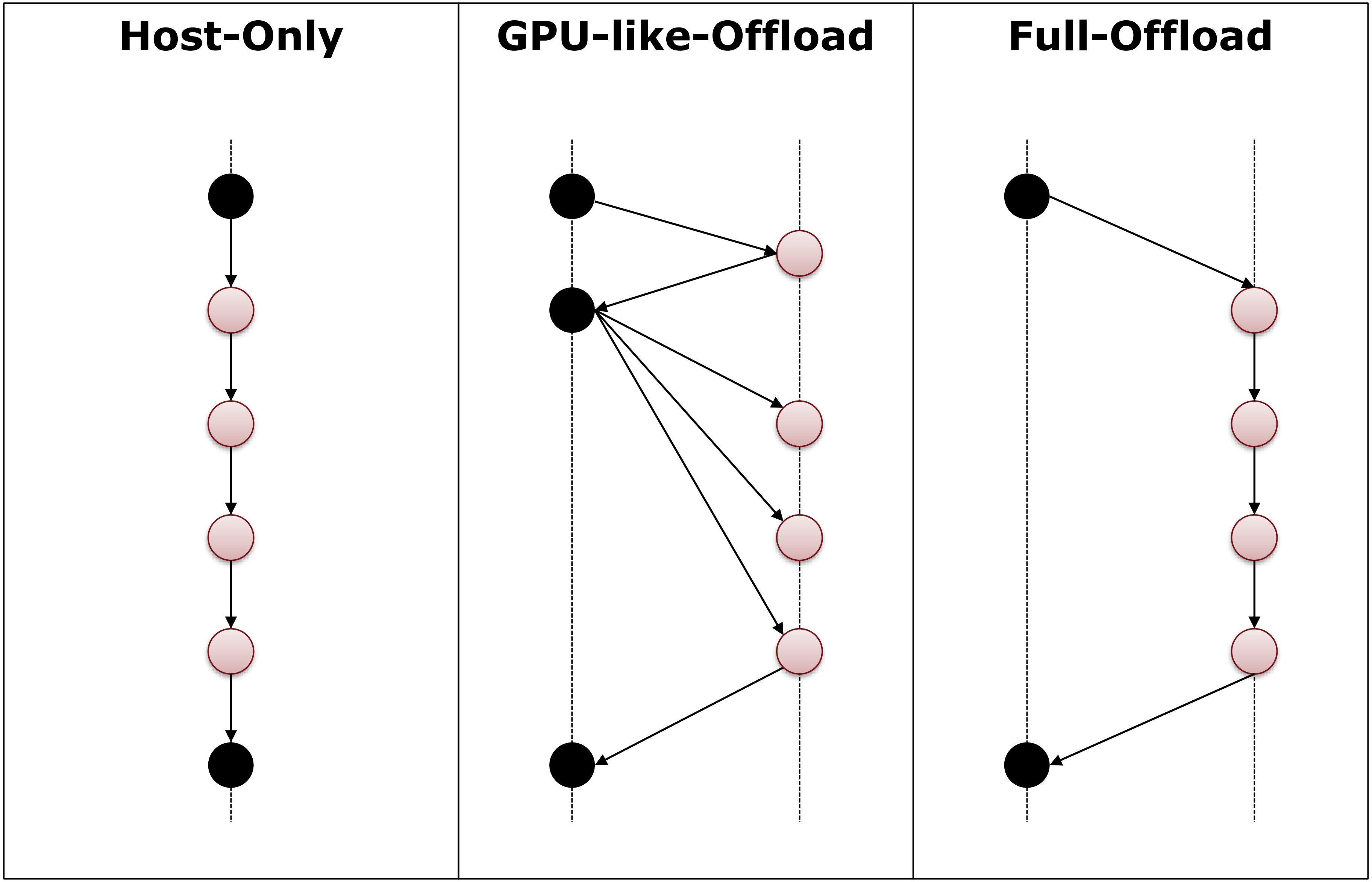
```
class SolLayer(torch.nn.Module):
      def  __init__(self):
            self.ParamA = …
            self.ParamB = …

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

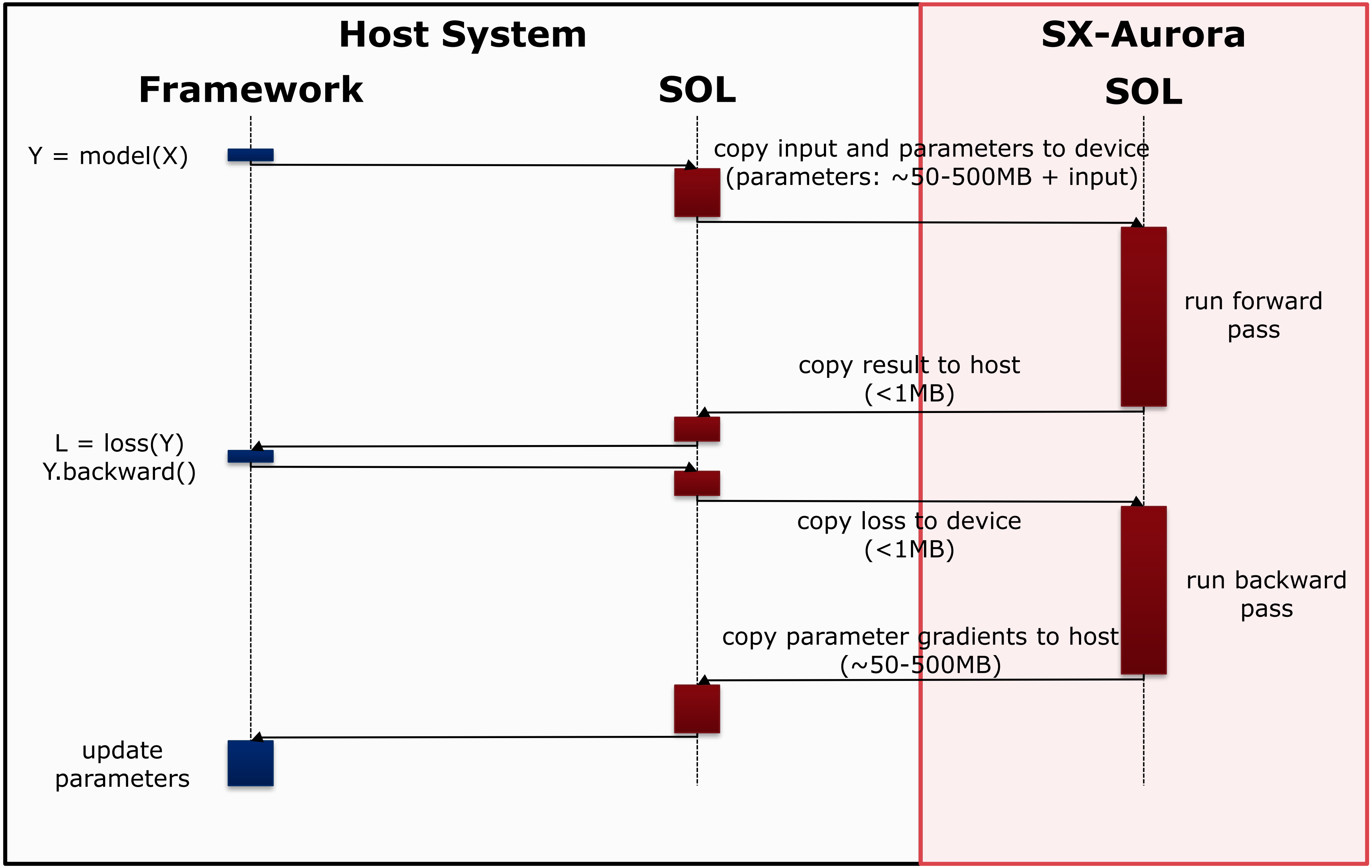
framework handles model parameters!

SOL handles execution

Execution Modes



SOL SX-Aurora Training



SOL VH-VE Coupling: VEO

Vector Engine Offloading

- Talked about it on 2 past WSSPs. Why not again?

API: OpenCL alike, but different, and yes, you can do CUDA style things

SOL-VE integration:

Language	files	blank	comment	code
C++	8	83	73	**348**
C/C++ Header	14	28	2	155
CMake	2	0	0	12
SUM:	24	111	75	515

- ~500 lines of code
- Good device abstraction in SOL!

AI frameworks need no VE adaptation at all!

- Easy!

But...

VEO is still … fresh

- Little issues here and there
- and:

Call latency

- System call penalty (~50us)
- One big kernel better than many small ones
- SOL generated code now has very few VEO function calls containing many other
  - Hint: the compiler is not always happy with passing more than 200 arguments

Host to device (VH-VE) data transfer

- The usual suspect for accelerator programming

VEO Data Transfer Speedup

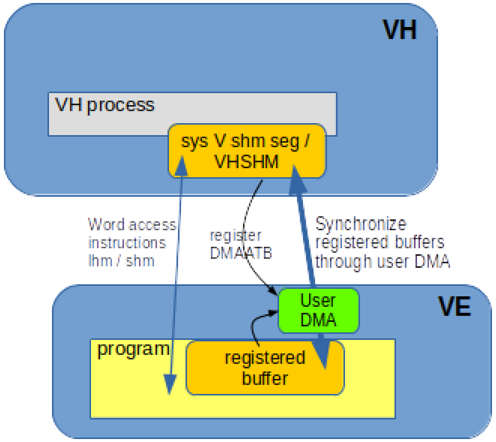
VEO default uses "system DMA" descriptors

- One set of descriptors per VE
- Only physical addressing: virtual-physical translation must be done on the fly
- Controlled by VEOS
- Initiated by VH

User DMA descriptors

- Each core has 2 sets
- Works with registered buffers
- No virtual-to-physical translation needed
- Controlled and initiated by VE user process
- Low level API exists:

```
#include <vhshm.h>
#include <vedma.h>
```



VEO-UDMA for SOL and others

VH and VE library for VEO programs

- Hides complexity of User DMA
- https://github.com/sx-aurora/veo-udma
- … work in progress...

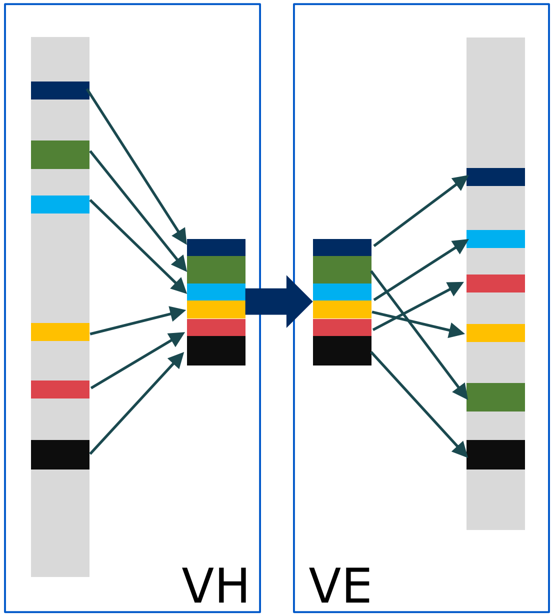
```
int veo_udma_peer_init(int ve_node_id, struct veo_proc_handle *proc,
                        struct veo_thr_ctxt *ctx, uint64_t lib_handle);
int veo_udma_peer_fini(int peer_id);
size_t veo_udma_send(struct veo_thr_ctxt *ctx, void *src, uint64_t dst, size_t len);
size_t veo_udma_recv(struct veo_thr_ctxt *ctx, uint64_t src, void *dst, size_t len);
```

on VH

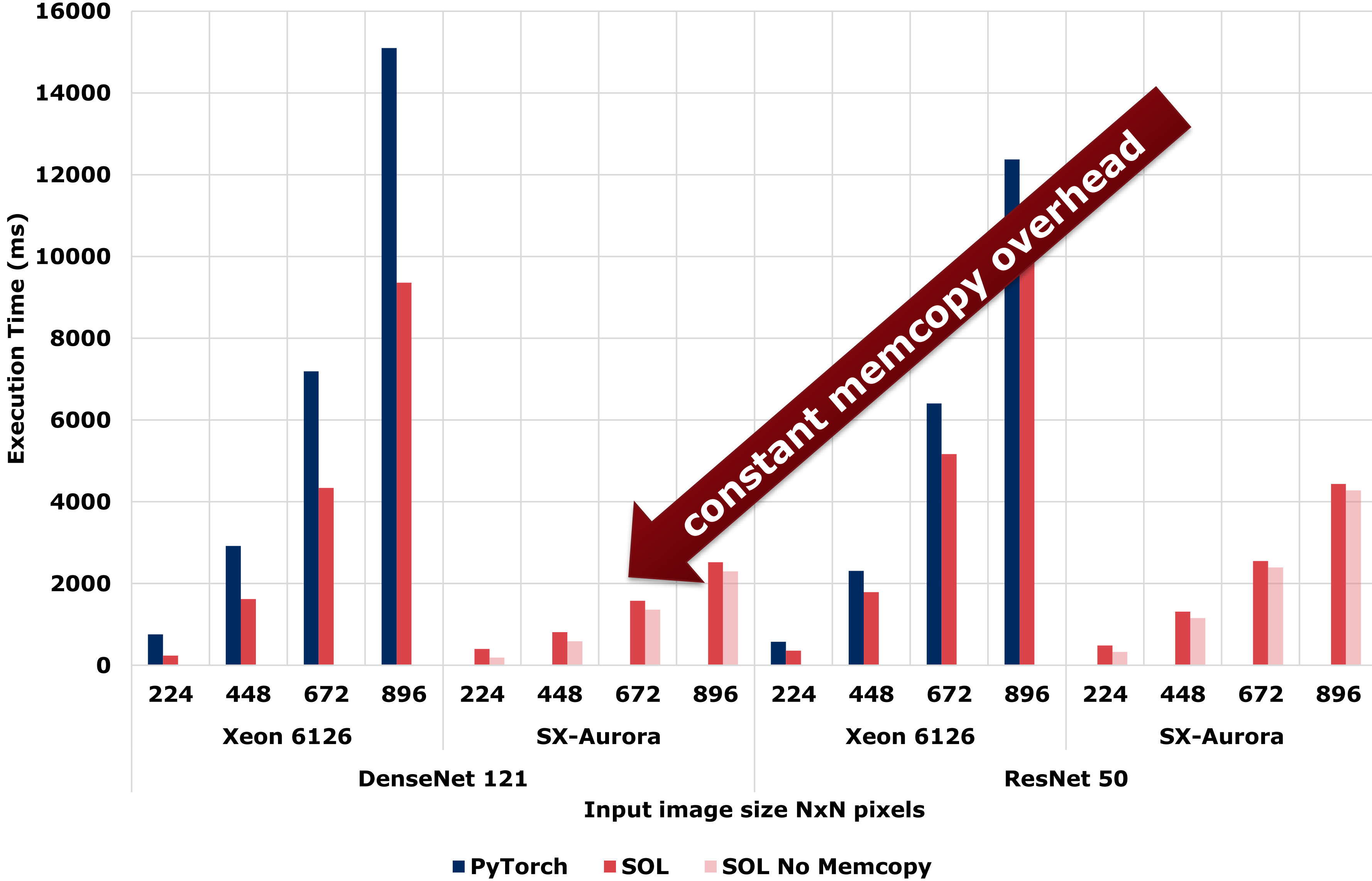
Gather/Scatter packed transfers

- Crucial for parameters transfer in SOL

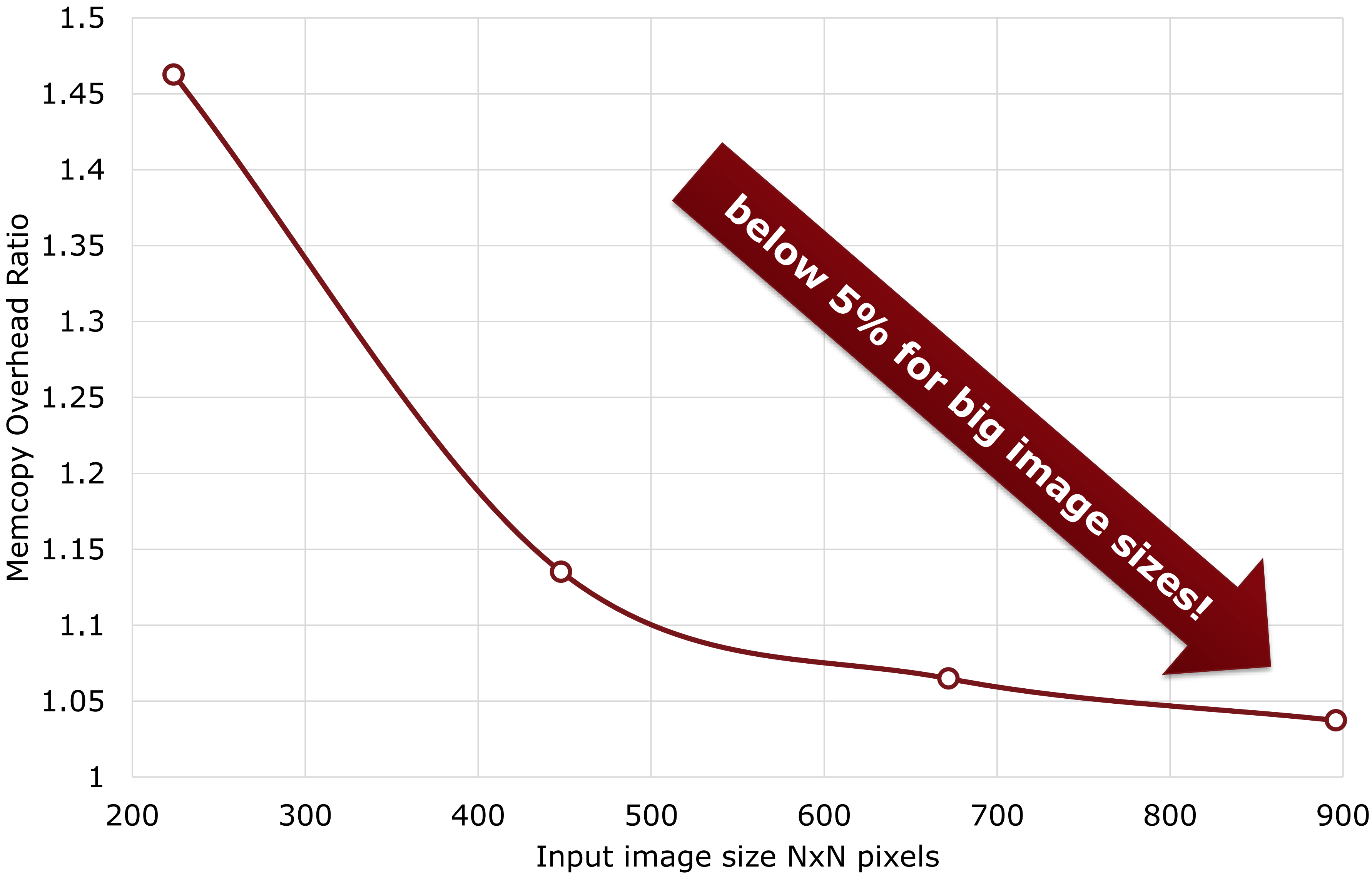
```
int veo_udma_send_pack(int peer, void *src, uint64_t dst, size_t len);
int veo_udma_send_pack_commit(int peer);
int veo_udma_recv_pack(int peer, uint64_t src, void *dst, size_t len);
int veo_udma_recv_pack_commit(int peer);
```



Performance Xeon Gold 6126 vs SX-Aurora (BS=8)



Memcopy Overhead Ratio



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)

Multi Layer Perceptron (MLP)

Linear/Logistic Regression

“How can the neural network be used in an application?”

SOL supports model deployment!

```
sol.deploy(trained_model, [1, 3, 224, 224],
target=sol.deployment.shared_lib, device=sol.device.ve,
func_name="predictMyStuff", …)
```

Generates library with a trained neural network model for inference

Native VE function call to integrate in your application

```
void predictMyStuff(const float* input, float** output);
```

Coming soon: Custom Layer Support

- Adding new functionality to the framework

```
class MyLayer(torch.nn.Layer):
  def __init__(self, …):
    super().__init__()




    self.ParamA = torch.nn.Parameter(…)
    self.ParamB = torch.nn.Parameter(…)

  def forward(self, X):
    # … code that executes when PyTorch
    # executes the layer …
```

Coming soon: Custom Layer Support

- Adding new functionality to the framework

```
class MyLayer(sol.nn.CustomLayer):
   def __init__(self, …):
      super().__init__({
      sol.device.nvidia: [“libMyCUDA.so”, “FwdCUDA”,
“BwdCUDA”],
      sol.device.ve:     [“libMyVE.so”, “FwdVE”, “BwdVE”]
      })
      self.ParamA = torch.nn.Parameter(…)
      self.ParamB = torch.nn.Parameter(…)

   def forward(self, X):
      # … code that executes when PyTorch
      # executes the layer …
```

Coming soon: Custom Layer Support

```
void FwdVE(void* ctx, const float* X, const float*
ParamA, const float* ParamB, float* Y) {
      /* YOUR CODE HERE */
}

void BwdVE(void* ctx, const float* dY, const float*
ParamA, const float* ParamB, float* dX, float* dParamA,
float* dParamB) {
      /* YOUR CODE HERE */
}
```

Orchestrating a brighter world

NEC

May the VECTOR be with you!

Nicolas Weber < **nicolas.weber@neclab.eu** >
Erich Focht < **erich.focht@emea.nec.com** >