codeplay

Enabling AI to be Open, Safe & Accessible to All

# Towards performance portability of AI models using SYCL-DNN

Muhammad Tanvir – Staff Software Engineer – ML libraries

IWOCL – 10-12 May 2022

# Outline

- Deep Learning Challenges
- Problem Statement – performance portability
- How does SYCL help?
- Performance Evaluations
- Deep Learning on RISC-V
- Conclusions

# Deep Learning Challenges

- **Diversity** of Technologies and Techniques
- **Migrating** between Deep Learning frameworks
- Multiple implementations of same **NN Optimization algorithms**
- **Maintainability** of various version of low-level backend/libraries integrated in existing high-level frameworks
- **Hardcoded** implementation of inference engines for a restricted set of hardware
- Multiple implementations of same **NN algorithms** across various target **HW**

# Deep Learning Challenges

- **Diversity** of Technologies and Techniques.
- **Migrating** between DL framework.

ONNX

- Multiple implementations of Same **NN Optimization algorithms.**
- **Maintainability** of various version of low-level libraries integrated in existing high-level frameworks.
- **Hardcoded** implementation of Inference engines for a restricted set of hardware.

ONNX Runtime

- Multiple implementations of same **NN algorithms** across various target **HW**

# Problem Statement

- Multiple implementations of same **NN algorithms** across various target **HW** – **<u>is there a way around this</u>**?

- An ideal solution(s) should:
  - Be **portable** across different platforms
  - Require **little (to no) changes** to the actual kernel code
  - Yield **acceptable performance** as compared to vendor optimized code
  - Have **backward compatibility (maintainability)**

# SYCL

- Supports cross-platform portability
  - Different implementations of SYCL compilers provide a variety of targets
- Is maintainable
  - Open-source implementations
- Kernel Code modifications?
- Performance Portability?

# SYCL

- Kernel Code Modification?

- **Possible Solution**: Use C++ template meta programming to write highly parametrized kernels (**for the most compute intensive operations**)

- **Benefits**:
  - Reuse the same kernel code
  - Modify (tune) the template parameters to maximize performance on target hardware

# SYCL

- Performance Portability?

- Possible Solution:
  - Tune the template parameters of the kernel to best match the underlying hardware*
  - Expose maximum performance out of the tuned kernel

- Acceptable Performance:
  - Within 70%-80% range of vendor optimized code's performance

*e.g. https://github.com/codeplaysoftware/sycl-blas/blob/master/tools/auto_tuner/gen/intel_gpu.json

# SYCL

## SYCL-DNN

- Conv
- Batchnorm
- Pool
- Softmax etc

https://github.com/codeplaysoftware/SYCL-DNN

## SYCL-BLAS

- GEMM
- GEMV
- Reduction etc

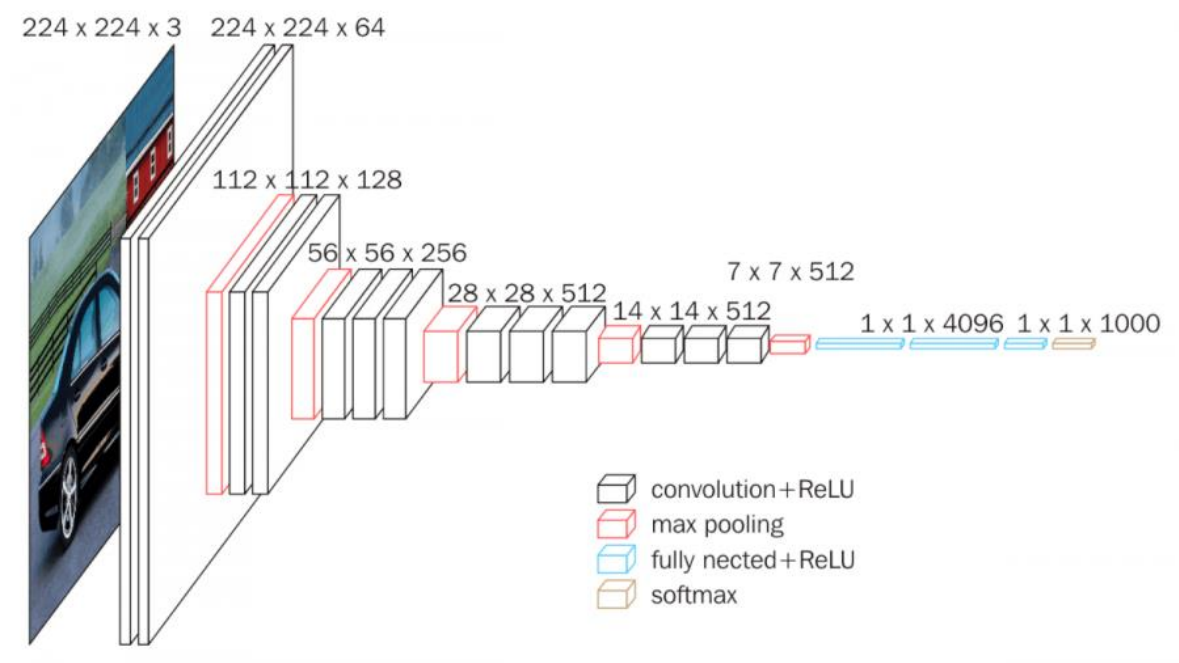
https://github.com/codeplaysoftware/sycl-blas

# Performance Evaluations

1. Identify the most compute intensive operation(s)
2. Choose the target hardware
3. Tune SYCL code based on the target hardware
4. Evaluate performance
5. Repeat steps 2-4 for remaining target hardware

# Performance Evaluations

**VGG**



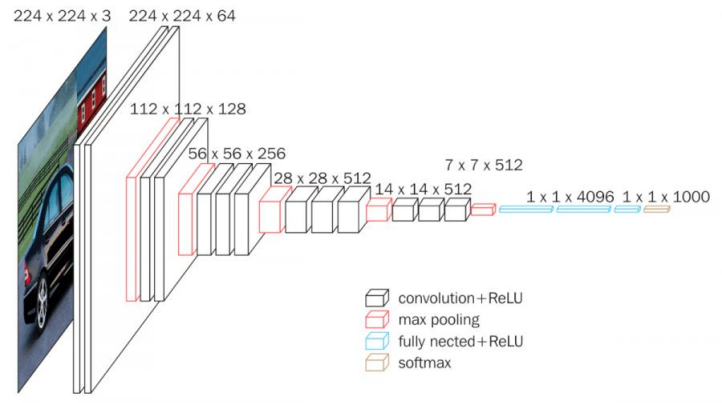


## VGG Demo

# Performance Evaluations

1. Identify the most compute intensive operation

**VGG**




```
Conv      ->    sycldnn::conv2d
BiasAdd   ->    sycldnn::biasAdd
Relu      ->    sycldnn::relu
Pooling   ->    sycldnn::pooling
Gemm      ->    syclblas::gemm
Softmax   ->    sycldnn::softmax
```

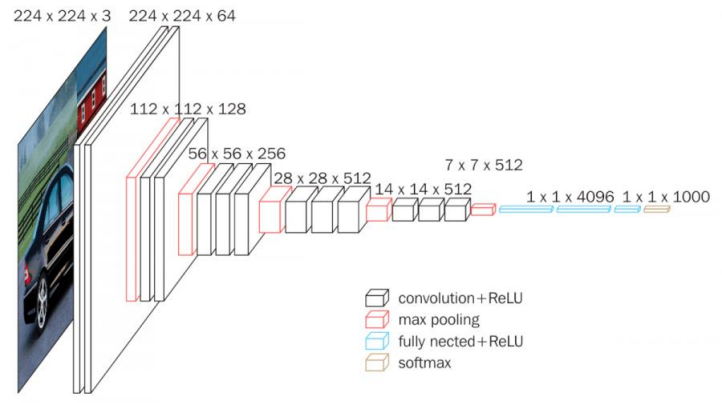
```
std::vector<float> output;
 std::string data_dir{argv[1]};
 auto input = read_image_data(argv[2], backend);
 Network network(input, output, data_dir, backend, *selector);
 network.add_layer<ConvolutionLayer, 3>({3, 64, 224});
 network.add_layer<BiasAddLayer, 2>({64, 224});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({64, 64, 224});
 network.add_layer<BiasAddLayer, 2>({64, 224});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({64, 224});
 network.add_layer<ConvolutionLayer, 3>({64, 128, 112});
 network.add_layer<BiasAddLayer, 2>({128, 112});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({128, 128, 112});
 network.add_layer<BiasAddLayer, 2>({128, 112});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({128, 112});
 network.add_layer<ConvolutionLayer, 3>({128, 256, 56});
 network.add_layer<BiasAddLayer, 2>({256, 56});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({256, 256, 56});
 network.add_layer<BiasAddLayer, 2>({256, 56});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({256, 256, 56});
 network.add_layer<BiasAddLayer, 2>({256, 56});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({256, 56});
 network.add_layer<ConvolutionLayer, 3>({256, 512, 28});
 network.add_layer<BiasAddLayer, 2>({512, 28});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 28});
 network.add_layer<BiasAddLayer, 2>({512, 28});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 28});
 network.add_layer<BiasAddLayer, 2>({512, 28});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({512, 28});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 14});
 network.add_layer<BiasAddLayer, 2>({512, 14});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 14});
 network.add_layer<BiasAddLayer, 2>({512, 14});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 14});
 network.add_layer<BiasAddLayer, 2>({512, 14});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({512, 14});
 network.add_layer<FullyConnectedLayer, 1>({4096});
 network.add_layer<BiasAddLayer, 2>({4096, 1});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<FullyConnectedLayer, 1>({4096});
 network.add_layer<BiasAddLayer, 2>({4096, 1});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<FullyConnectedLayer, 1>({1000});
 network.add_layer<BiasAddLayer, 2>({1000, 1});
 network.add_layer<SoftmaxLayer, 0>({});
 network.run();
```

# Performance Evaluations

1. Identify the most compute intensive operation

**GEMM**

**VGG**




```
Conv      ->    sycldnn::conv2d
BiasAdd   ->    sycldnn::biasAdd
Relu      ->    sycldnn::relu
Pooling   ->    sycldnn::pooling
Gemm      ->    syclblas::gemm
Softmax   ->    sycldnn::softmax
```

```
std::vector<float> output;
 std::string data_dir{argv[1]};
 auto input = read_image_data(argv[2], backend);
 Network network(input, output, data_dir, backend, *selector);
 network.add_layer<ConvolutionLayer, 3>({3, 64, 224});
 network.add_layer<BiasAddLayer, 2>({64, 224});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({64, 64, 224});
 network.add_layer<BiasAddLayer, 2>({64, 224});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({64, 224});
 network.add_layer<ConvolutionLayer, 3>({64, 128, 112});
 network.add_layer<BiasAddLayer, 2>({128, 112});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({128, 128, 112});
 network.add_layer<BiasAddLayer, 2>({128, 112});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({128, 112});
 network.add_layer<ConvolutionLayer, 3>({128, 256, 56});
 network.add_layer<BiasAddLayer, 2>({256, 56});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({256, 256, 56});
 network.add_layer<BiasAddLayer, 2>({256, 56});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({256, 256, 56});
 network.add_layer<BiasAddLayer, 2>({256, 56});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({256, 56});
 network.add_layer<ConvolutionLayer, 3>({256, 512, 28});
 network.add_layer<BiasAddLayer, 2>({512, 28});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 28});
 network.add_layer<BiasAddLayer, 2>({512, 28});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 28});
 network.add_layer<BiasAddLayer, 2>({512, 28});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({512, 28});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 14});
 network.add_layer<BiasAddLayer, 2>({512, 14});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 14});
 network.add_layer<BiasAddLayer, 2>({512, 14});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<ConvolutionLayer, 3>({512, 512, 14});
 network.add_layer<BiasAddLayer, 2>({512, 14});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<PoolingLayer, 2>({512, 14});
 network.add_layer<FullyConnectedLayer, 1>({4096});
 network.add_layer<BiasAddLayer, 2>({4096, 1});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<FullyConnectedLayer, 1>({4096});
 network.add_layer<BiasAddLayer, 2>({4096, 1});
 network.add_layer<ReLULayer, 0>({});
 network.add_layer<FullyConnectedLayer, 1>({1000});
 network.add_layer<BiasAddLayer, 2>({1000, 1});
 network.add_layer<SoftmaxLayer, 0>({});
 network.run();
```

# Performance Evaluations

## 2. Choose the target hardware

- Intel CPU
- Intel GPU
- NVIDIA GPU
- RISC-V

# Performance Evaluations

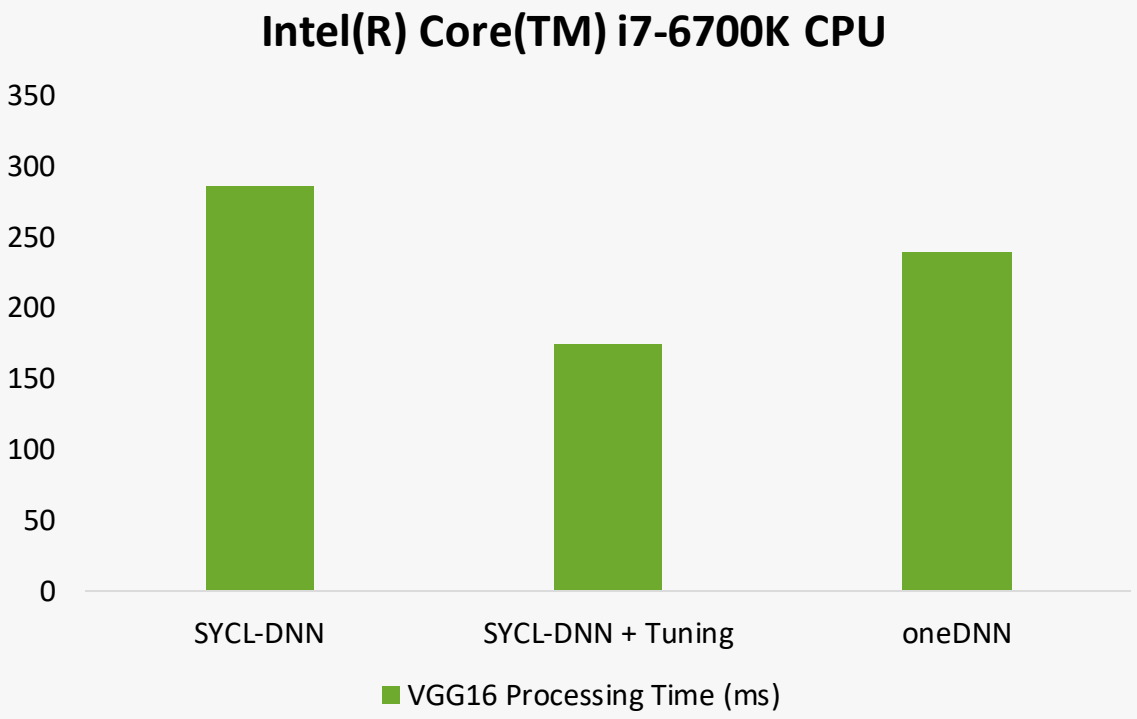
## 3. Tune SYCL code based on the target hardware

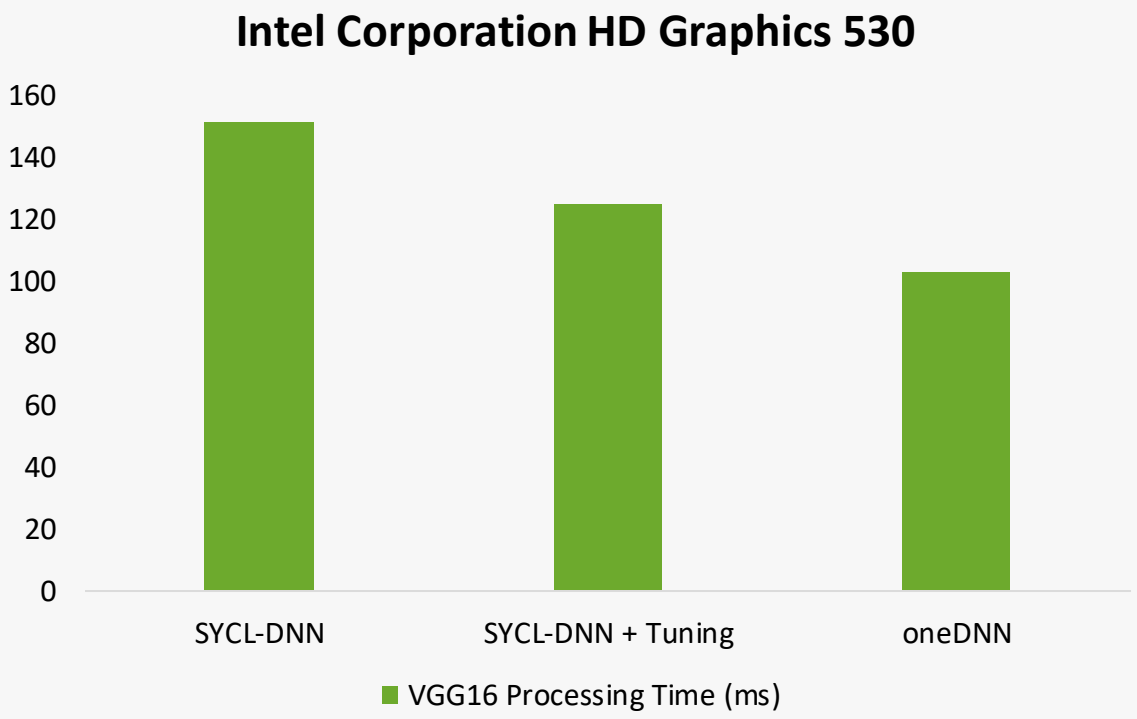
- Target Hardware
  - Intel(R) Core(TM) i7-6700K CPU @ 4 GHz
  - Intel Corporation HD Graphics 530
- Most Compute Intensive Operation
  - GEMM
- Tuning Methodology
  - Lawson, John, and Mehdi Goli. "Performance portability through machine learning guided kernel selection in SYCL libraries." *Parallel Computing* 107 (2021): 102813.

# Tuning paradigm

- Every GEMM operation, in DNNs, has different compute intensity
- Most optimal solution
  - One tuned kernel per GEMM operation
  - Yields the best performance
  - Size of the library increases drastically
- Semi-optimal solution
  - One tuned kernel per DNN model
  - Yields semi-optimal performance
  - Limited no. of kernels in the library

# Performance Evaluations




*With semi-optimal tuning regime

# Performance Evaluations

## 2. Choose the target hardware

- Intel CPU
- Intel GPU
- NVIDIA GPU
- RISC-V

# Performance Evaluations

## 3. Tune SYCL code based on the target hardware

- Target Hardware
  - NVIDIA Titan RTX GPU – 24 GB DDR6
- Most Compute Intensive Operation
  - GEMM
- Tuning tool
  - Modified version of SYCL-BLAS auto-tuner*

*https://github.com/codeplaysoftware/sycl-blas/tree/master/tools/auto_tuner

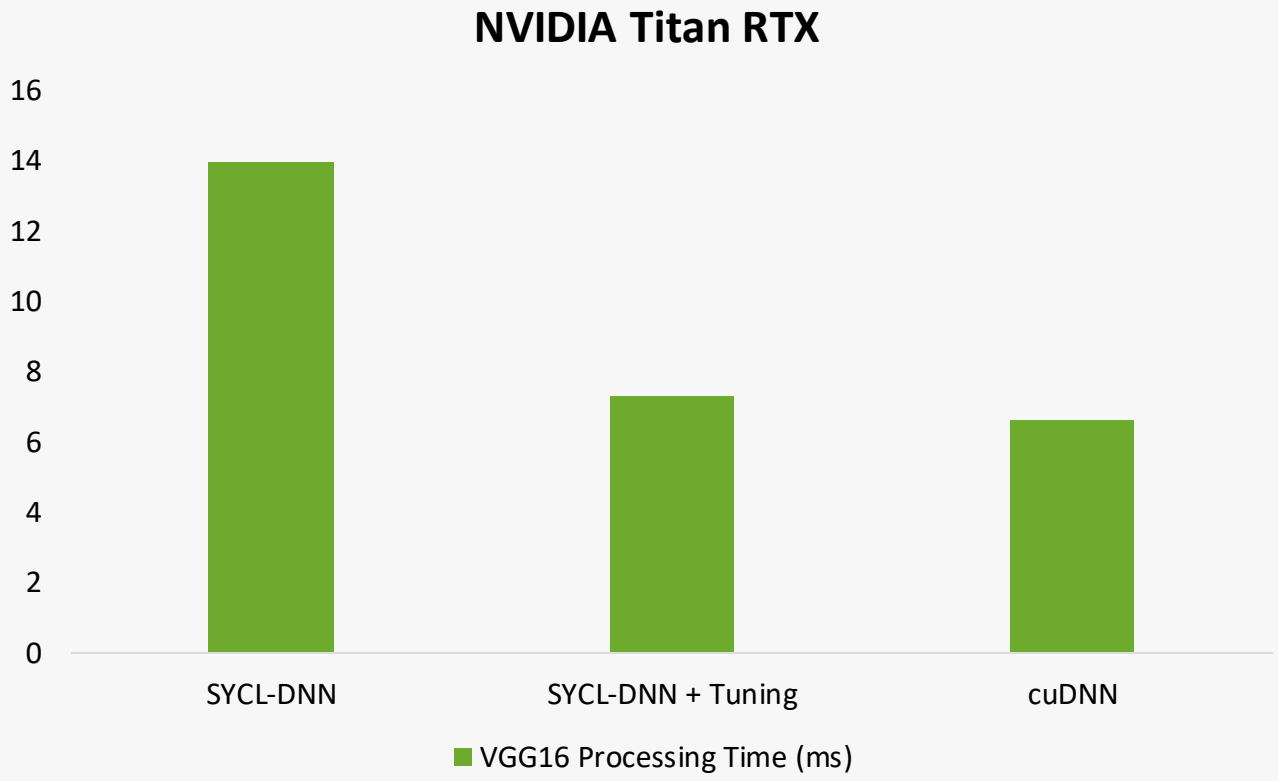
# Performance Evaluations

## 3. Tune SYCL code based on the target hardware

- Search all possible GEMM configs* exhaustively
  - Choose a GEMM config
  - Run the entire VGG16 (DNN) model and measure performance
  - Record results for all possible GEMM configs
  - Choose the GEMM config which yields the best performance

*https://github.com/codeplaysoftware/sycl-blas/blob/master/tools/auto_tuner/gen/nvidia_gpu.json

# Performance Evaluations



# Performance Evaluations

## 2. Choose the target hardware

- Intel CPU
- Intel GPU
- NVIDIA GPU
- RISC-V

# Deep Learning on RISC-V

- Target Hardware
  - RISC-V spike simulator – single core

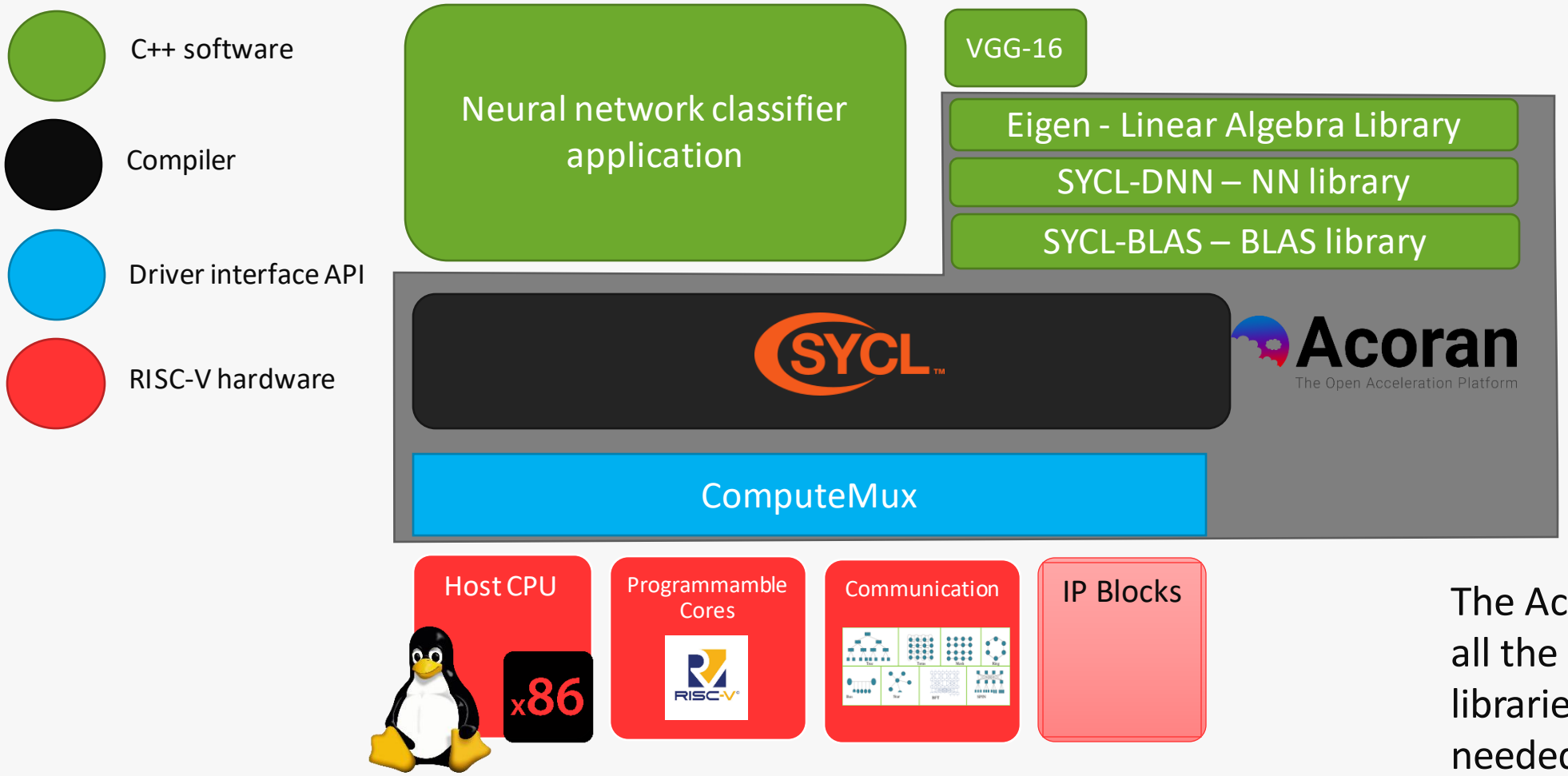
- Software Stack
  - Acoran – The Open Acceleration Platform*

*https://www.codeplay.com/solutions/acoran/

# Why Neural networks on RISC-V?

- Domain specific accelerators are required to achieve **cost-effective performance** on-chip
- Cost effective performance requires **tuning the design** to the needs of the workload required
- RISC-V ISA has a minimalist base integer instruction set and provides custom extensions
  - An ideal starting point for creating special accelerators
- More companies are looking at RISC-V to enable AI software
- Designs can benefit from the RISC-V vector extension
  - Enables vectorization for various application
  - Helps achieve high compute density on chip

# Acoran – The Open Acceleration Platform




The Acoran platform provides all the supporting open-source libraries and frameworks needed to build this neural network demonstration

# GEMM Operator

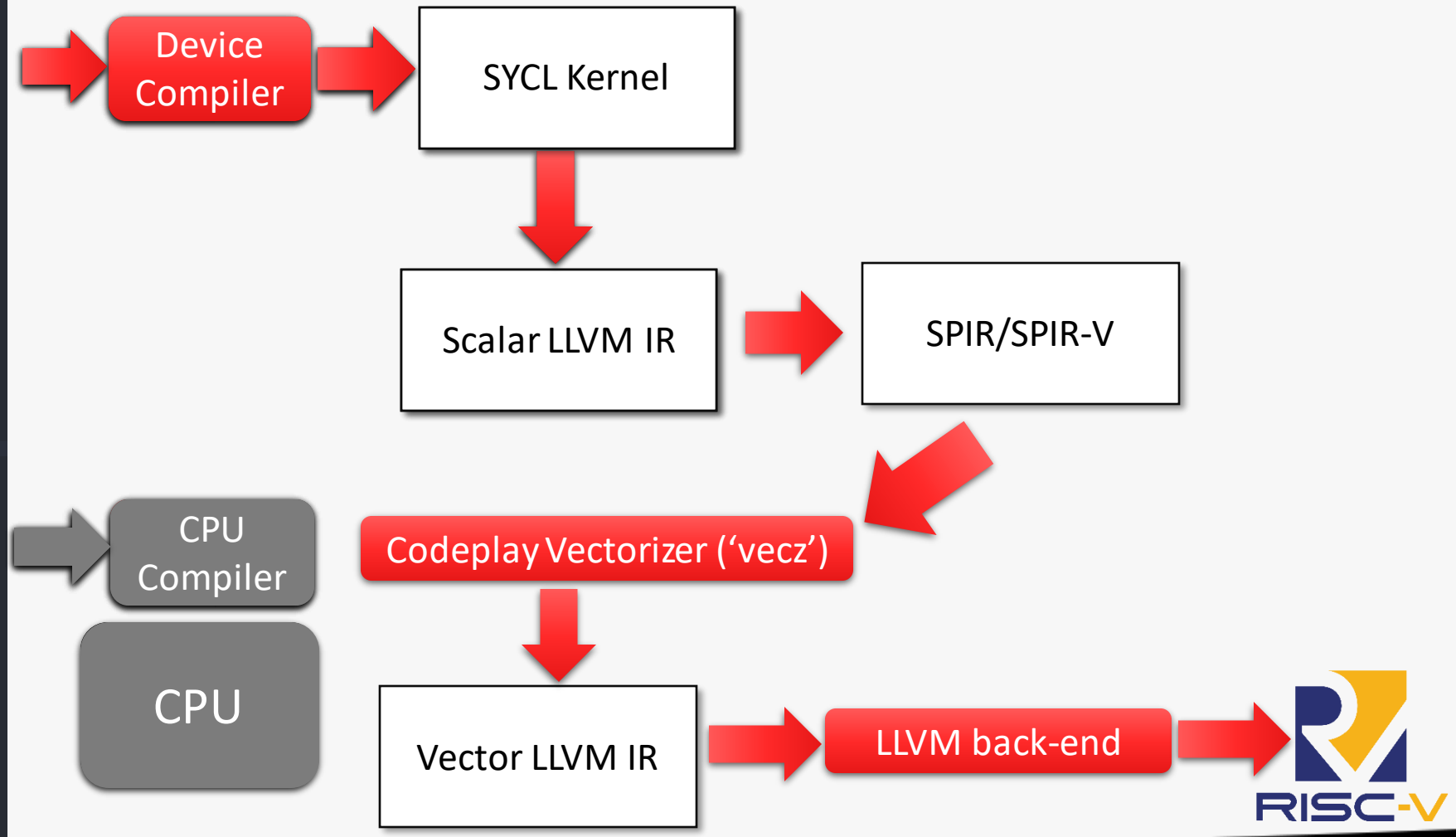
```
{   ....
    range<1> dimensions(matSize * matSize);
    const property_list props = {property::buffer::use_host_ptr()};
    buffer<T> bA(MA, dimensions, props);
    buffer<T> bB(MB, dimensions, props);
    buffer<T> bC(MC, dimensions, props);

    q.submit([&](handler& cgh) {
      auto pA = bA.template get_access<access::mode::read>(cgh);
      auto pB = bB.template get_access<access::mode::read>(cgh);
      auto pC = bC.template get_access<access::mode::write>(cgh);
      auto localRange = range<1>(blockSize * blockSize);

      accessor<T, 1, access::mode::read_write, access::target::local> pBA(
          localRange, cgh);
      accessor<T, 1, access::mode::read_write, access::target::local> pBB(
          localRange, cgh);

      cgh.parallel_for<mxm_kernel>(
          nd_range<2>{range<2>(matSize, matSize),
                      range<2>(blockSize, blockSize)},
          [=](nd_item<2> it) {
            // Current block
            int blockX = it.get_group(1);
            int blockY = it.get_group(0);

            // Current local item
            int localX = it.get_local_id(1);
            int localY = it.get_local_id(0);

            // Start in the A matrix
            int a_start = matSize * blockSize * blockY;
            // End in the b matrix
            int a_end = a_start + matSize - 1;
            // Start in the b matrix
            int b_start = blockSize * blockX;

            // Result for the current C(i,j) element
            T tmp = 0.0f;
            // We go through all a, b blocks
            for (int a = a_start, b = b_start; a <= a_end;
                 a += blockSize, b += (blockSize * matSize)) {
              // Copy the values in shared memory collectively
              pBA[localY * blockSize + localX] =
                  pA[a + matSize * localY + localX];
              // Note the swap of X/Y to maintain contiguous access
              pBB[localX * blockSize + localY] =
                  pB[b + matSize * localY + localX];
              it.barrier(access::fence_space::local_space);
              // Now each thread adds the value of its sum
              for (int k = 0; k < blockSize; k++) {
                tmp +=
                    pBA[localY * blockSize + k] * pBB[localX * blockSize + k];
              }
              // The barrier ensures that all threads have written to local
              // memory before continuing
              it.barrier(access::fence_space::local_space);
            }
            auto elemIndex = it.get_global_id(0) * it.get_global_range()[1] +
                             it.get_global_id(1);
            // Each thread updates its position
            pC[elemIndex] = tmp;
          });
    });
  }
  return false;
}
```

Main Computation

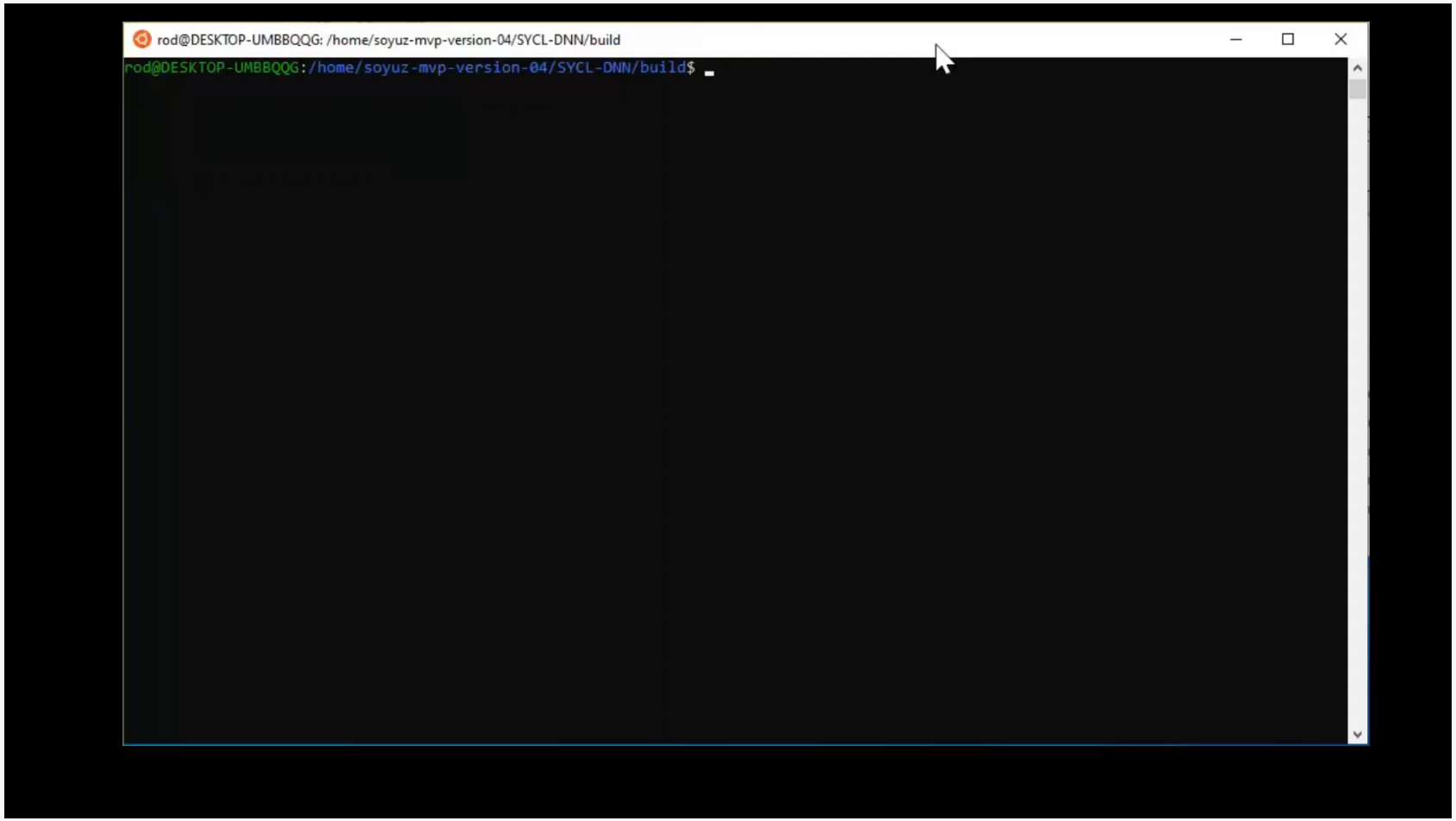
```
...
for (int a = a_start, b = b_start; a <= a_end;
                 a += blockSize, b += (blockSize * matSize)) {
    for (int k = 0; k < blockSize; k++) {
    tmp +=
        pBA[localY * blockSize + k] * pBB[localX * blockSize + k];
    }
    // The barrier ensures that all threads have written to local
    // memory before continuing
    it.barrier(access::fence_space::local_space);
}
...
pC[elemIndex] = tmp;
...
```

# RISC-V/RVV Kernel compilation flow FC

```
    range<1> dimensions(matSize * matSize);
    const property_list props = {property::buffer::use_host_ptr()};
    buffer<T> bA(MA, dimensions, props);
    buffer<T> bB(MB, dimensions, props);
    buffer<T> bC(MC, dimensions, props);

    q.submit([&](handler& cgh) {
      auto pA = bA.template get_access<access::mode::read>(cgh);
      auto pB = bB.template get_access<access::mode::read>(cgh);
      auto pC = bC.template get_access<access::mode::write>(cgh);
      auto localRange = range<1>(blockSize * blockSize);

      accessor<T, 1, access::mode::read_write, access::target::local> pBA(
          localRange, cgh);
      accessor<T, 1, access::mode::read_write, access::target::local> pBB(
          localRange, cgh);

      cgh.parallel_for<mxm_kernel>(
          nd_range<2>{range<2>(matSize, matSize),
                      range<2>(blockSize, blockSize)},
          [=](nd_item<2> it) {
            // Current block
            int blockX = it.get_group(1);
            int blockY = it.get_group(0);

            // Current local item
            int localX = it.get_local_id(1);
            int localY = it.get_local_id(0);

            // Start in the A matrix
            int a_start = matSize * blockSize * blockY;
            // End in the b matrix
            int a_end = a_start + matSize - 1;
            // Start in the b matrix
            int b_start = blockSize * blockX;

            // Result for the current C(i,j) element
            T tmp = 0.0f;
            // We go through all a, b blocks
            for (int a = a_start, b = b_start; a <= a_end;
                 a += blockSize, b += (blockSize * matSize)) {
              // Copy the values in shared memory collectively
              pBA[localY * blockSize + localX] =
                  pA[a + matSize * localY + localX];
              // Note the swap of X/Y to maintain contiguous access
              pBB[localX * blockSize + localY] =
                  pB[b + matSize * localY + localX];
              it.barrier(access::fence_space::local_space);
              // Now each thread adds the value of its sum
              for (int k = 0; k < blockSize; k++) {
                tmp +=
                    pBA[localY * blockSize + k] * pBB[localX * blockSize + k];
              }
              // The barrier ensures that all threads have written to local
              // memory before continuing
              it.barrier(access::fence_space::local_space);
            }
            auto elemIndex = it.get_global_id(0) * it.get_global_range()[1] +
                             it.get_global_id(1);
            // Each thread updates its position
            pC[elemIndex] = tmp;
          });
    });
  }
  return false;
}
```



# Deep Learning on RISC-V




VGG Demo

# Conclusions

- SYCL-DNN / SYCL-BLAS have support for efficient acceleration of popular DNNs
- Acoran platform provides an end-to-end compute stack for accelerating DNNs on RISC-V
  - https://developer.codeplay.com/products/acoran/pre-alpha
- Recent Update: Adding SYCL to upstream ONNX Runtime
  - https://github.com/codeplaysoftware/onnxruntime

We're Hiring!

codeplay.com/careers/

codeplay®

Enable AI & HPC to be Open, Safe and Accessible to All

# Thank you

muhammad.tanvir@codeplay.com

@codeplaysoft

info@codeplay.com

codeplay.com