codeplay

Enabling AI to be Open, Safe & Accessible to All

Accelerating Neural Networks using Open Standard Software on RISC-V

Mehdi Goli, VP R&D

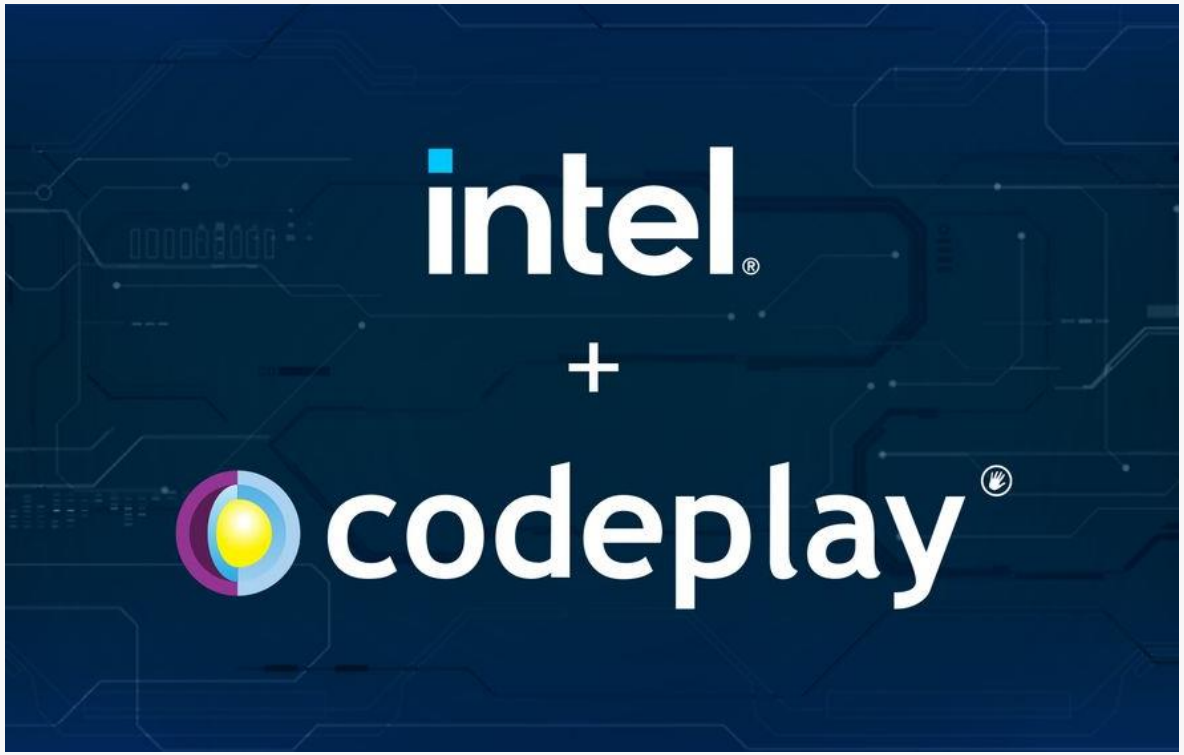
Kumudha Narasinham, Senior SE

Presenter: Peter Žužek, Principal SE

ISC 2023, First International workshop on RISC-V for HPC

Who we are

- After years of collaboration and contribution to open standards alongside **intel**, **Codeplay Software** is a subsidiary of **Intel** after an acquisition made this year.
- We will continue to operate as Codeplay Software and will work extensively with **all relevant industries** to **advance the SYCL ecosystem**, especially around **oneAPI**
- Codeplay is now working jointly with intel to further advance the **SYCL standard** and the **oneAPI** open ecosystem.



© 2023 Codeplay Software Ltd.

Neural networks and 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

© 2023 Codeplay Software Ltd.

SYCL

- Open standard API introduced by Khronos
  - Uses ISO standard C++ code
- Provides single-source programming model for accelerator processors
- Allow accessing both high-level and low-level code
- Suitable for graph model programming by tracking kernel dependency
- Multiple implementations
  - ComputeCPP
  - DPC++
  - hipSYCL
- Programming model
  - Kernel Scope
  - Command group scope
  - Application Scope

© 2023 Codeplay Software Ltd.

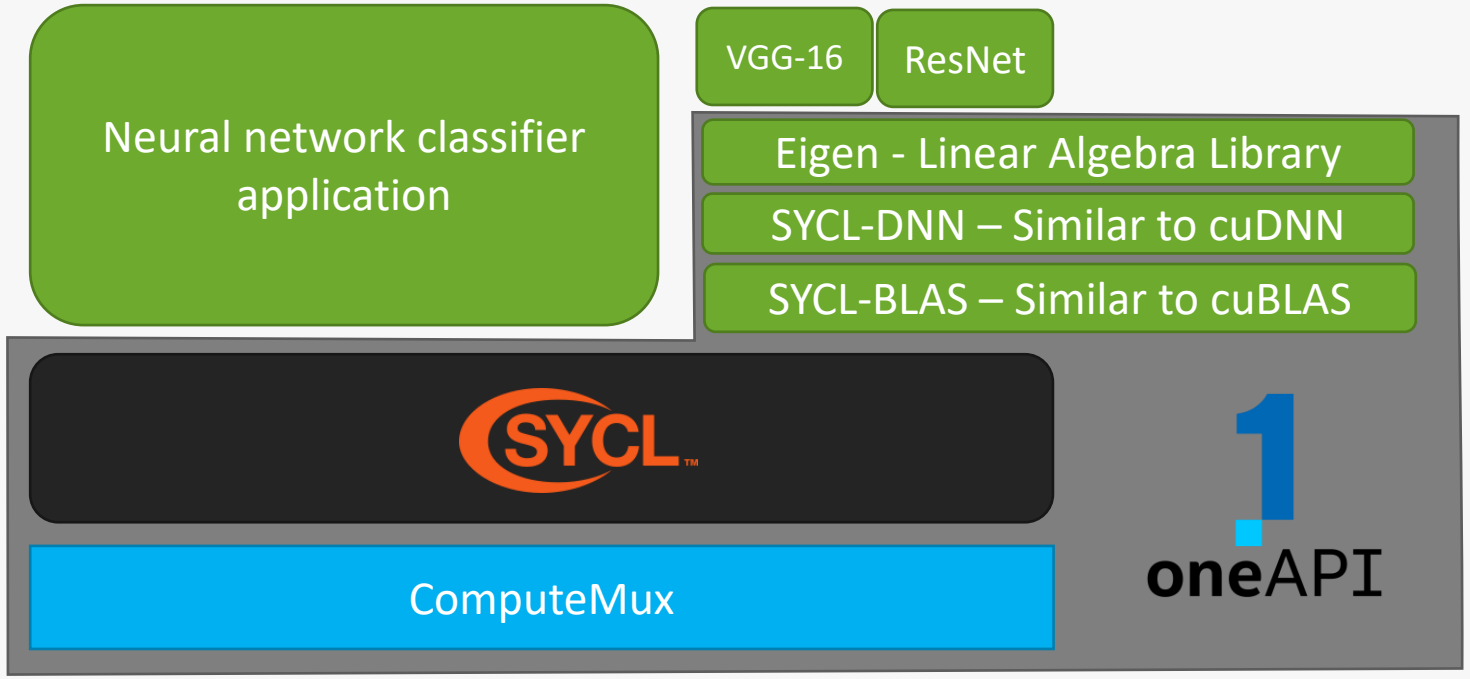
SYCL on RISC-V Architecture

C++ software

Compiler

Driver interface API

RISC-V hardware

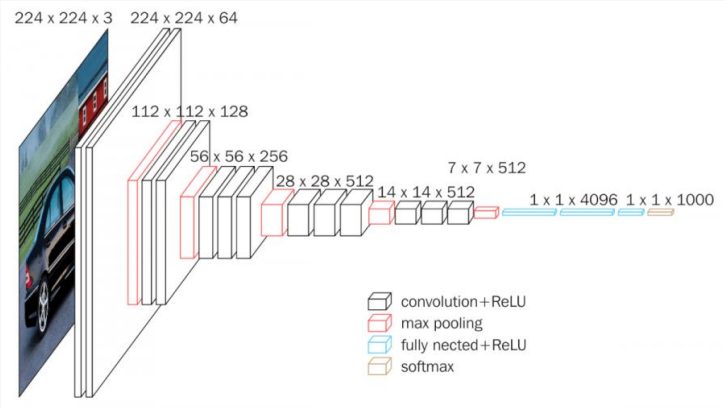


The oneAPI platform provides all the supporting open source libraries and frameworks needed to build this neural network demonstration

© 2023 Codeplay Software Ltd.

Neural network primitives in VGG

VGG




```
sycldnn::conv2d
sycldnn::BiasAdd
sycldnn::Relu
sycldnn::pooling
syclblas::gemm
sycldnn::BiasAdd
sycldnn::Relu
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();
```

```
{   ....
    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;
}
```

Fully Connected layer (FC)

Main Computation

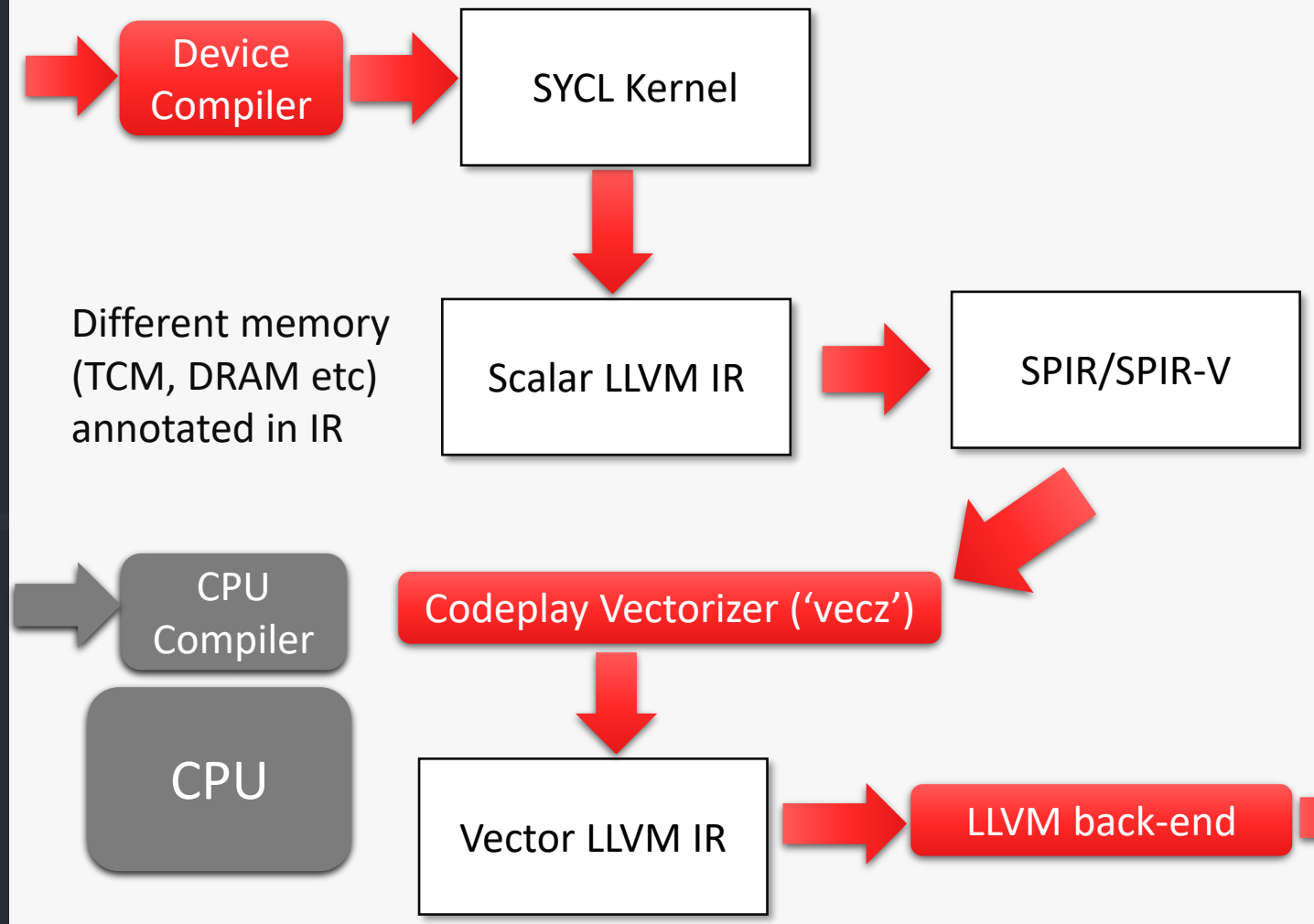
```
...
for (int a = a_start, b = b_start; a <= a_end;
                 a += blockSize, b += (blockSize * matSize)) {
    for (int k = 0; k < blockSize; k++) {
    tmp +=
        pBA[localY * blockSize + k] * pBB[localX * blockSize + k];
    }
    // The barrier ensures that all threads have written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

- Compilation
- Offloading to a Scalar/Vector RISC-V device

© 2023 Codeplay Software Ltd.

```
{   ....
    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;
}
```

RISC-V/RVV Kernel compilation flow FC



© 2023 Codeplay Software Ltd.

FC: Kernel

- SYCL Kernel
- LLVM IR
- SPIR-V
- ComputeMux LLVM IR
- RISC-V Vector Assembly

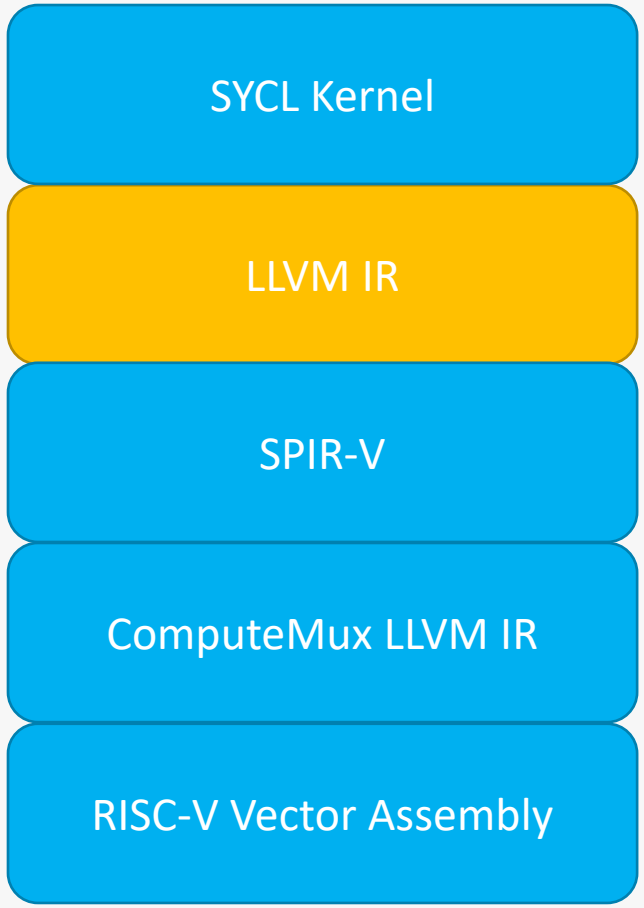
```
cgh.parallel_for<mxm_kernel>(
    nd_range<2>{range<2>(matSize, matSize),
                range<2>(blockSize, blockSize)},
    [=](nd_item<2> it) {
    // Current block
    int blockX = it.get_group(1);
    int blockY = it.get_group(0);

    // 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
    float 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;
    });
```

These are the 2 most inner kernel loops. The compiler *inserts the outer loop* to loop over all the data items to be processed by these cores

```
...
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;
...
```


© 2023 Codeplay Software Ltd.

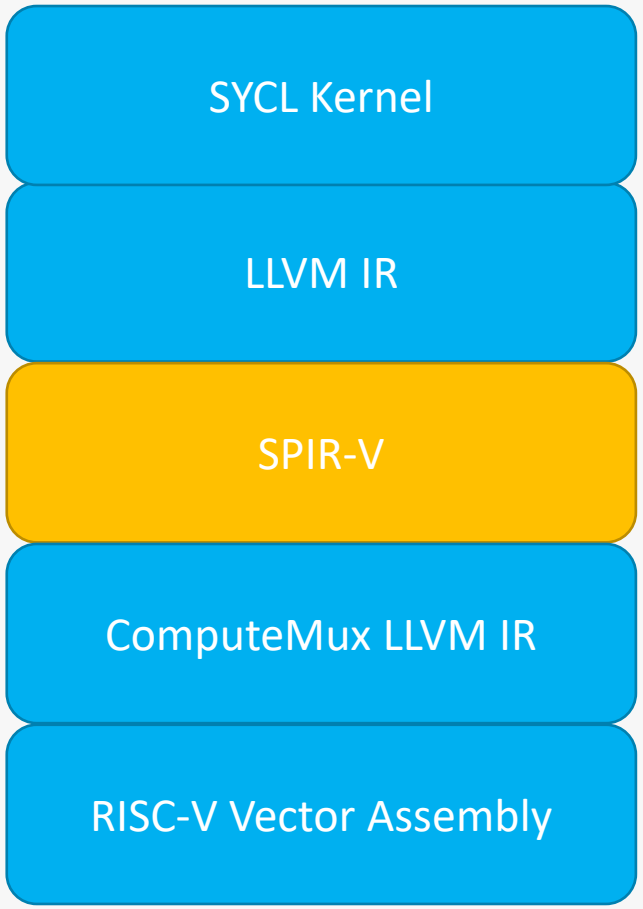
FC : LLVM IR




```
; <label>:50:                                     ; preds = %55, %38
  %51 = phi float [ %39, %38 ], [ %67, %55 ]
  tail call spir_func void @_Z7barrierj(i32 1) #4
  %52 = add nsw i32 %41, %1
  %53 = add nsw i32 %40, %19
  %54 = icmp slt i32 %52, %21
  br i1 %54, label %38, label %70

; <label>:55:                                     ; preds = %55, %38
  %56 = phi float [ %67, %55 ], [ %39, %38 ]
  %57 = phi i32 [ %68, %55 ], [ 0, %38 ]
  %58 = add nsw i32 %57, %29
  %59 = sext i32 %58 to i64
  %60 = getelementptr inbounds float, float addrspace(3)* %2, i64 %59
  %61 = load float, float addrspace(3)* %60, align 4, !tbaa !11
  %62 = add nsw i32 %57, %33
  %63 = sext i32 %62 to i64
  %64 = getelementptr inbounds float, float addrspace(3)* %4, i64 %63
  %65 = load float, float addrspace(3)* %64, align 4, !tbaa !11
  %66 = fmul float %61, %65
  %67 = fadd float %56, %66
  %68 = add nuw nsw i32 %57, 1
  %69 = icmp slt i32 %68, %1
  br i1 %69, label %55, label %50
```


© 2023 Codeplay Software Ltd.

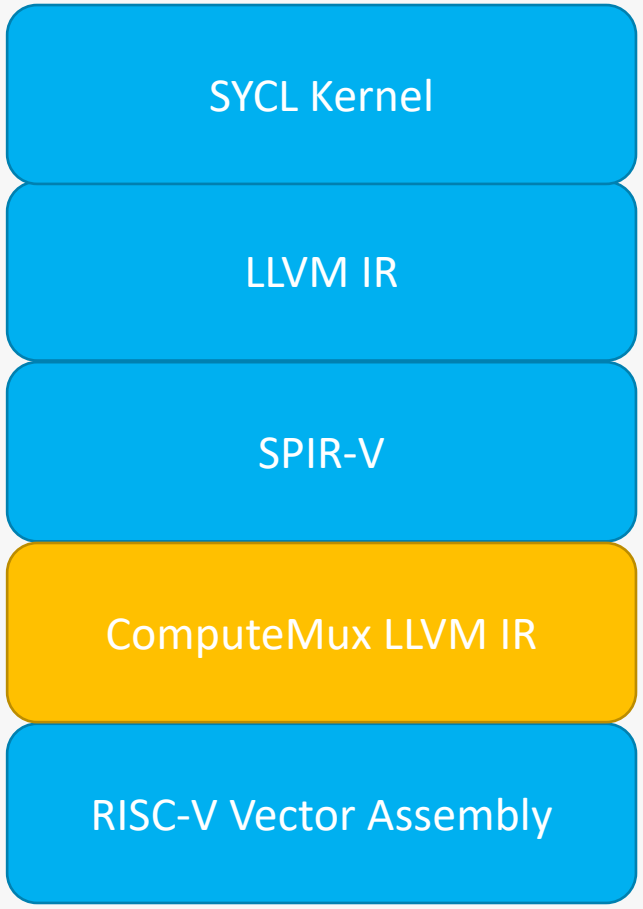
FC: SPIR-V




```
2 Label 26
7 Phi 11 85 84 26 69 25
7 Phi 10 87 86 26 49 25
5 IAdd 10 88 87 58
4 SConvert 2 89 88
5 InBoundsPtrAccessChain 12 90 18 89
6 Load 11 91 90 2 4
5 IAdd 10 92 87 62
4 SConvert 2 93 92
5 InBoundsPtrAccessChain 12 94 20 93
6 Load 11 95 94 2 4
5 FMul 11 96 91 95
5 FAdd 11 84 85 96
5 IAdd 10 86 87 98
5 SLessThan 50 100 86 17
4 BranchConditional 100 26 27

2 Label 27
7 Phi 11 68 69 25 84 26
4 ControlBarrier 82 82 83
5 IAdd 10 72 73 17
5 IAdd 10 70 71 46
5 SLessThan 50 104 72 48
4 BranchConditional 104 25 28
```


© 2023 Codeplay Software Ltd.

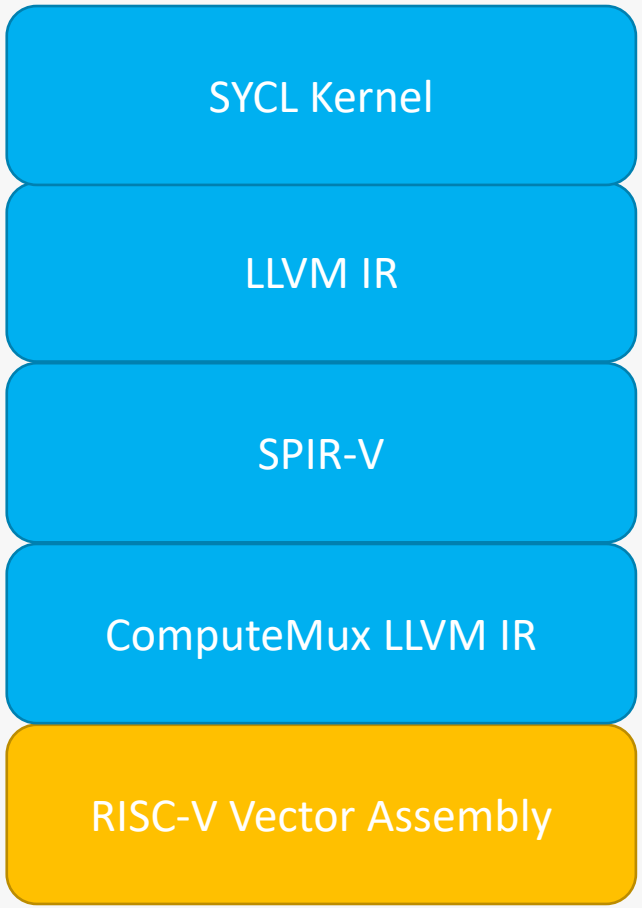
FC: ComputeMux LLVM IR




```
%855 = shufflevector <4 x float> %848, <4 x float> undef, <4 x i32> zeroinitializer
%mul.i.i.i37.i.i = fmul <4 x float> %855, %853
%add.i.i.i38.i.i = fadd <4 x float> %363, %mul.i.i.i37.i.i
%856 = shufflevector <4 x float> %848, <4 x float> undef, <4 x i32> <i32 1, i32 1, i32 1, i32 1>
%mul.i.i.i35.i.i = fmul <4 x float> %856, %852
%add.i.i.i36.i.i = fadd <4 x float> %mul.i.i.i35.i.i, %add.i.i.i38.i.i
%857 = shufflevector <4 x float> %848, <4 x float> undef, <4 x i32> <i32 2, i32 2, i32 2, i32 2>
%mul.i.i.i33.i.i = fmul <4 x float> %857, %851
%add.i.i.i34.i.i = fadd <4 x float> %mul.i.i.i33.i.i, %add.i.i.i36.i.i
%858 = shufflevector <4 x float> %848, <4 x float> undef, <4 x i32> <i32 3, i32 3, i32 3, i32 3>
%mul.i.i.i31.i.i = fmul <4 x float> %858, %854
%add.i.i.i32.i.i = fadd <4 x float> %mul.i.i.i31.i.i, %add.i.i.i34.i.i
%859 = shufflevector <4 x float> %849, <4 x float> undef, <4 x i32> zeroinitializer
%mul.i.i.i29.i.i = fmul <4 x float> %859, %853
%add.i.i.i30.i.i = fadd <4 x float> %362, %mul.i.i.i29.i.i
%860 = shufflevector <4 x float> %849, <4 x float> undef, <4 x i32> <i32 1, i32 1, i32 1, i32 1>
%mul.i.i.i27.i.i = fmul <4 x float> %860, %852
%add.i.i.i28.i.i = fadd <4 x float> %mul.i.i.i27.i.i, %add.i.i.i30.i.i
%861 = shufflevector <4 x float> %849, <4 x float> undef, <4 x i32> <i32 2, i32 2, i32 2, i32 2>
%mul.i.i.i25.i.i = fmul <4 x float> %861, %851
%add.i.i.i26.i.i = fadd <4 x float> %mul.i.i.i25.i.i, %add.i.i.i28.i.i
%862 = shufflevector <4 x float> %849, <4 x float> undef, <4 x i32> <i32 3, i32 3, i32 3, i32 3>
%mul.i.i.i23.i.i = fmul <4 x float> %862, %854
%add.i.i.i24.i.i = fadd <4 x float> %mul.i.i.i23.i.i, %add.i.i.i26.i.i
%863 = shufflevector <4 x float> %850, <4 x float> undef, <4 x i32> zeroinitializer
%mul.i.i.i21.i.i = fmul <4 x float> %863, %853
%add.i.i.i22.i.i = fadd <4 x float> %361, %mul.i.i.i21.i.i
%864 = shufflevector <4 x float> %850, <4 x float> undef, <4 x i32> <i32 1, i32 1, i32 1, i32 1>
%mul.i.i.i19.i.i = fmul <4 x float> %864, %852
%add.i.i.i20.i.i = fadd <4 x float> %mul.i.i.i19.i.i, %add.i.i.i22.i.i
%865 = shufflevector <4 x float> %850, <4 x float> undef, <4 x i32> <i32 2, i32 2, i32 2, i32 2>
%mul.i.i.i17.i.i = fmul <4 x float> %865, %851
%add.i.i.i18.i.i = fadd <4 x float> %mul.i.i.i17.i.i, %add.i.i.i20.i.i
%866 = shufflevector <4 x float> %850, <4 x float> undef, <4 x i32> <i32 3, i32 3, i32 3, i32 3>
%mul.i.i.i15.i.i = fmul <4 x float> %866, %854
%add.i.i.i16.i.i = fadd <4 x float> %mul.i.i.i15.i.i, %add.i.i.i18.i.i
```


© 2023 Codeplay Software Ltd.

FC: RISC-V Vector assembly




```
vsetvli a0, zero, e32, mf2, ta, mu
vfmul.vf    v24, v17, ft2
vfadd.vv    v30, v24, v30
vfmul.vf    v24, v16, ft2
vfadd.vv    v13, v24, v13
vfmul.vf    v24, v15, ft2
vfadd.vv    v22, v24, v22
vfmul.vf    v24, v14, ft2
vfadd.vv    v23, v24, v23
vsetivli    zero, 1, e32, mf2, ta, mu
vslidedown.vi   v24, v8, 2
vfmv.f.s    ft2, v24
vsetvli a0, zero, e32, mf2, ta, mu
vfmul.vf    v24, v12, ft2
vfadd.vv    v30, v24, v30
vfmul.vf    v24, v10, ft2
vfadd.vv    v13, v24, v13
vfmul.vf    v24, v9, ft2
vfadd.vv    v22, v24, v22
vfmul.vf    v24, v31, ft2
vfadd.vv    v23, v24, v23
vsetivli    zero, 1, e32, mf2, ta, mu
vslidedown.vi   v8, v8, 3
vfmv.f.s    ft2, v8
vsetvli a0, zero, e32, mf2, ta, mu
vfmul.vf    v8, v29, ft2
vfadd.vv    v2, v8, v30
vfmul.vf    v30, v28, ft2
vfadd.vv    v1, v30, v13
vfmul.vf    v30, v27, ft2
vfadd.vv    v24, v30, v22
vfmul.vf    v30, v26, ft2
vfadd.vv    v22, v30, v23
```


© 2023 Codeplay Software Ltd.

Benefits of this approach

1. Easy porting of large-scale software from GPUs to new processor designs
2. Can support a very wide range of high-performance architectures
3. Easily tuneable: tile-sizes, memory-sizes, algorithms, memory access patterns, parallelization, scheduling
4. Supports real-world and dynamic workloads

© 2023 Codeplay Software Ltd.

Thank you!

Notices & Disclaimers
Intel technologies may require enabled hardware, software or service activation.
No product or component can be absolutely secure.
Your costs and results may vary.
© Codeplay Software Ltd.. Codeplay, Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.