Scientific Computing Accelerated on FPGA

Unified Shared Memory and External Function Interface with oneAPI

Suleyman Demirsoy

Copyright © 2022 Intel Corporation.

This document is intended for personal use only. Unauthorized distribution, modification, public performance, public display, or copying of this material via any medium is strictly prohibited

Legal Notices and Disclaimers

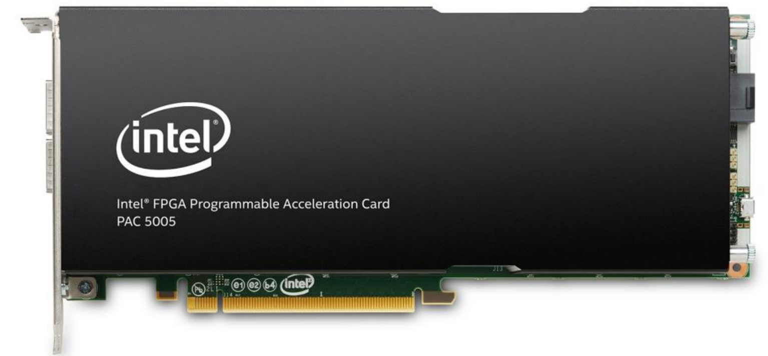
For notices and disclaimers, visit www.intel.com/PerformanceIndex or scan the QR code:

© Intel Corporation. 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.

Agenda

- Motivation and Vision
- Unified Shared Memory
- External Function Interface

oneAPI



Copyright © 2022 Intel Corporation

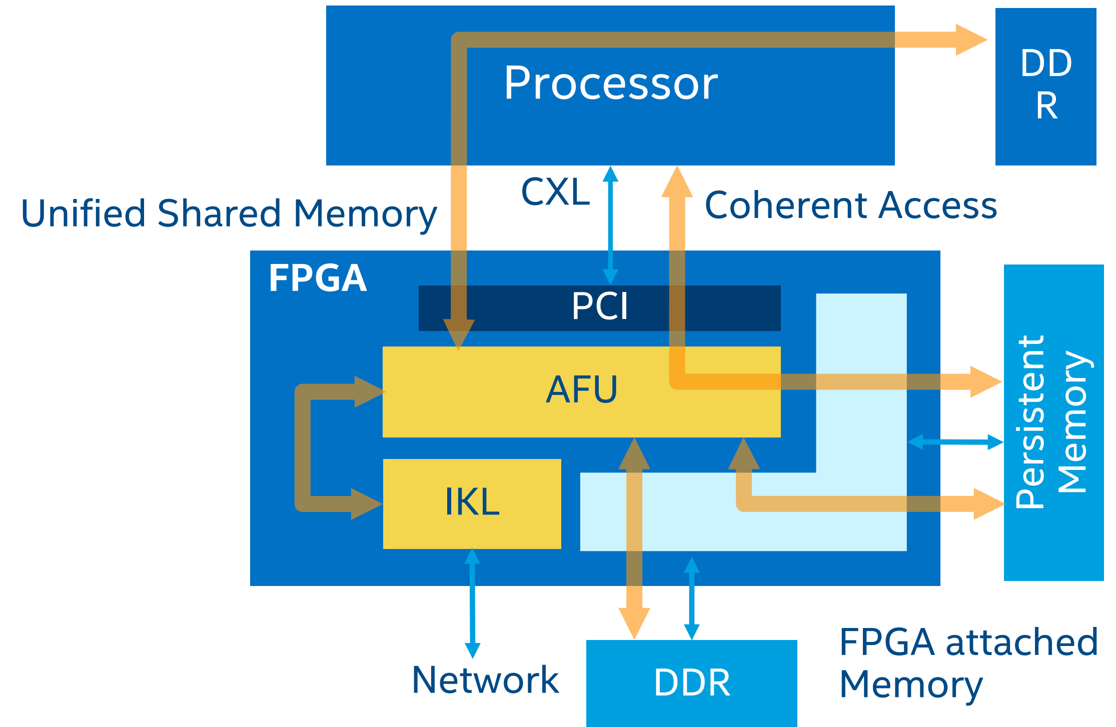
Our Vision

- Revolutionize next generation heterogenous HPC and data management systems with coherent FPGA technologies in order to:
  - accelerate critical system and time-sensitive business tasks
  - get more work done per node for less power
  - better organize memory for quicker access to data in demand
- with familiar and easy to use development flows and C++ access methods.

Copyright © 2022 Intel Corporation

Abstracting the Access with oneAPI and CXL

- Datapaths of importance
  a. AFU ←→ Memory (CPU, FPGA)
  b. AFU ←→ Storage (Local, Remote
  c. CPU → AFU, Memory (FPGA)
  d. AFU ←→ Network



- Shared Memory and CXL protocol capabilities support these accesses.
- Inter Kernel Link (IKL) can be integrated to Open FPGA Stack.

Copyright © 2022 Intel Corporation

Universal Shared Memory

- USM
  - FPGA takes care about data transfers
  - Host Allocations
  - Shared Allocations
  - Device Allocations
    need: malloc_device, memcpy or
    data transfer kernel
- USM ideal for streaming applications at interface speed!
- Without USM: Buffers with accessors - SYCL runtime controlled

```
562 q.submit([&](handler& h) {
563   h.single_task<paralleladdKernel>([=]() [[intel::kernel_args_restrict]] {
564
565     host_ptr<int> in(in_host);
566     host_ptr<int> out(out_host);
567
568     int i_cnt = 0;
569
570         // Init regs to zero
571         [[intel::fpga_register]] std::array<int, 16> raw_data;
572         #pragma unroll
573         for (int i = 0; i < 16; i++) { raw_data[i] = 0; }
574
575         [[intel::fpga_register]] std::array<int, 8> add_data;
576         #pragma unroll
577         for (int i = 0; i < 8; i++) { add_data[i] = 0; }
578
579
580
581       // Run over all CLs
582       while(i_cnt < iterations) {
583
584         // Load complete CL in one clock cycle, (same for PCIe and DDR4)
585         #pragma unroll
586         for (uint idx = 0; idx < 16; idx++) {
587           {
588             raw_data[idx] = in[idx + i_cnt*16];
589           }
590         }
591
592         add_data[0] = raw_data[0] + raw_data[8];
593         add_data[1] = raw_data[1] + raw_data[9];
594         add_data[2] = raw_data[2] + raw_data[10];
595         add_data[3] = raw_data[3] + raw_data[11];
596         add_data[4] = raw_data[4] + raw_data[12];
597         add_data[5] = raw_data[5] + raw_data[13];
598         add_data[6] = raw_data[6] + raw_data[14];
599         add_data[7] = raw_data[7] + raw_data[15];
600
601         // Write results back with half CL, as we can write and read
602         // a CL per clock cycle this will create no bottleneck
603         #pragma unroll
604         for (uint idx = 0; idx < 8; idx++) {
605           {
606             out[idx + i_cnt*8] = add_data[idx];
607           }
608         }
609
610
611         i_cnt++;
612
613       }
614
615   });
616 }).wait();
```

Copyright © 2022 Intel Corporation

PCIe 3.0 – Host

- Calculate number of necessary cache lines
- malloc_host to allocate host buffer
- Initialize input and output buffers
- After kernel finished read output buffer

```
234  ///////////////
235  // parallel_add
236  ///////////////
237
238      printf("\n \n ### parallel_add ### \n\n");
239
240      if(2*size % 16 == 0)
241      {
242          number_CL = 2*size / 16;
243      }
244      else
245      {
246          number_CL = 2*size / 16 + 1;
247      }
248      printf("Number CLs: %zd \n", number_CL);
249
250
251
252  if ((in = malloc_host<Type>(number_CL*16, q)) == nullptr) {
253      std::cerr << "ERROR: could not allocate space for 'in'\n";
254      std::terminate();
255  }
256  if ((out = malloc_host<Type>(number_CL*16/2, q)) == nullptr) {
257      std::cerr << "ERROR: could not allocate space for 'out'\n";
258      std::terminate();
259  }
260
261      // Init input buffer
262      for(int i=0; i< (number_CL*16); ++i)
263      {
264          if(i < 2*size)
265          {
266              if(i % 16 > 7)
267              {
268                  in[i] = 5;
269              }
270              else
271              {
272                  in[i] = 7;
273              }
274          }
275          else
276          {
277              in[i] = 0;
278          }
279      }
280
281      // Init output buffer
282      for(int i=0; i< (number_CL*16/2); ++i)
283      {
284          out[i] = 312;
285      }
```

Copyright © 2022 Intel Corporation

PCIe 3.0 – Kernel

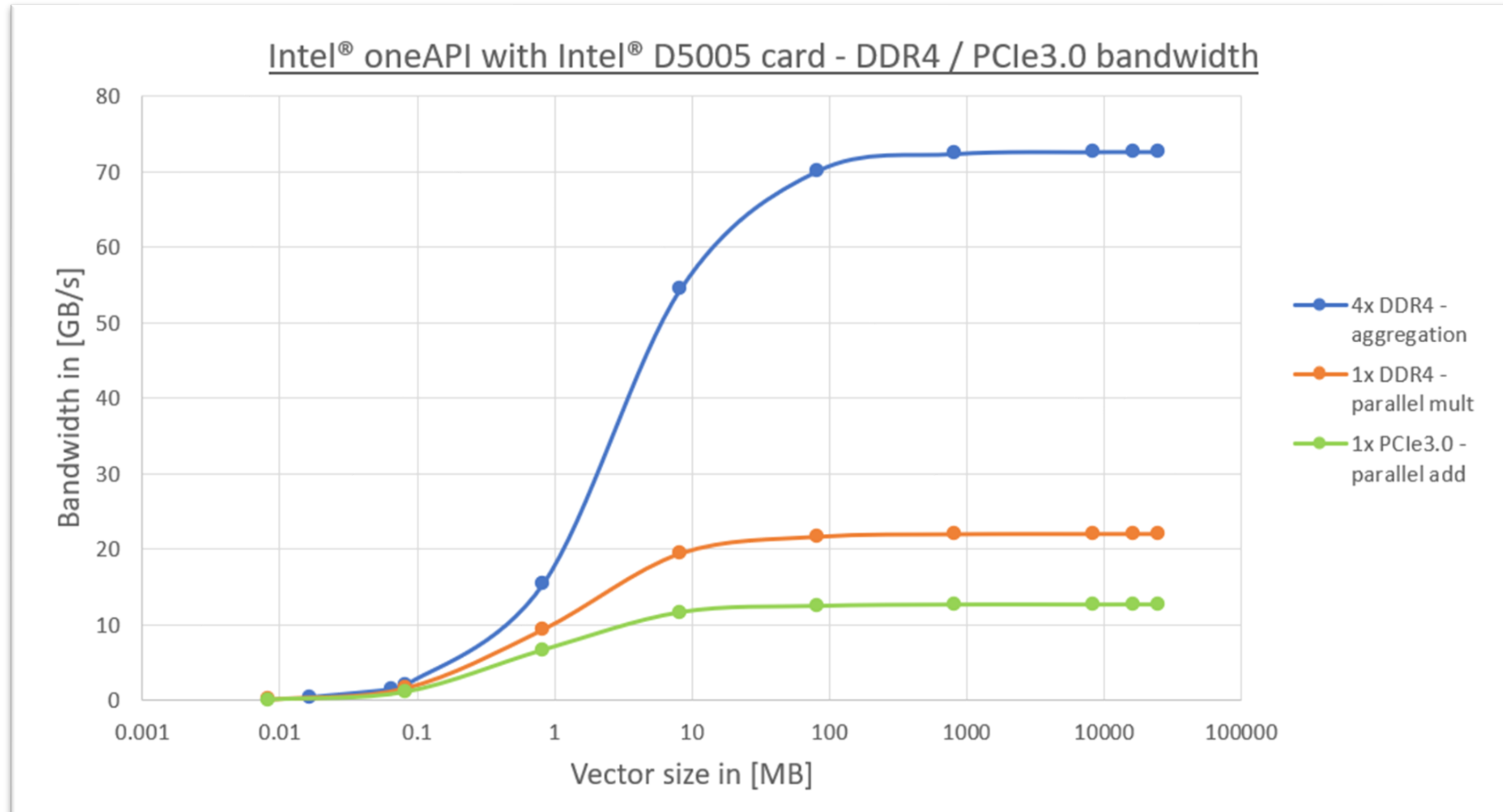
- USM
  - FPGA takes care about data transfers
  - Host pointers
  - Iterate over complete cache lines
  - Make sure you load/store one cache line per clock cycle to use interface BW
    - BW Fmax dependent
- USM ideal for streaming applications at interface speed!

```
562  q.submit([&](handler& h) {
563    h.single_task<paralleladdKernel>([=]() [[intel::kernel_args_restrict]] {
564
565      host_ptr<int> in(in_host);
566      host_ptr<int> out(out_host);
567
568        int i_cnt = 0;
569
570                // Init regs to zero
571                [[intel::fpga_register]] std::array<int, 16> raw_data;
572                #pragma unroll
573                for (int i = 0; i < 16; i++) { raw_data[i] = 0; }
574
575                [[intel::fpga_register]] std::array<int, 8> add_data;
576                #pragma unroll
577                for (int i = 0; i < 8; i++) { add_data[i] = 0; }
578
579
580
581            // Run over all CLs
582            while(i_cnt < iterations) {
583
584                // Load complete CL in one clock cycle, (same for PCIe and DDR4)
585                #pragma unroll
586                for (uint idx = 0; idx < 16; idx++) {
587                    {
588                        raw_data[idx] = in[idx + i_cnt*16];
589                    }
590                }
591
592                add_data[0] = raw_data[0] + raw_data[8];
593                add_data[1] = raw_data[1] + raw_data[9];
594                add_data[2] = raw_data[2] + raw_data[10];
595                add_data[3] = raw_data[3] + raw_data[11];
596                add_data[4] = raw_data[4] + raw_data[12];
597                add_data[5] = raw_data[5] + raw_data[13];
598                add_data[6] = raw_data[6] + raw_data[14];
599                add_data[7] = raw_data[7] + raw_data[15];
600
601                // Write results back with half CL, as we can write and read
602                // a CL per clock cycle this will create no bottleneck
603                #pragma unroll
604                for (uint idx = 0; idx < 8; idx++) {
605                    {
606                        out[idx + i_cnt*8] = add_data[idx];
607                    }
608                }
609
610
611                i_cnt++;
612
613            }
614
615    });
616  }).wait();
```

Copyright © 2022 Intel Corporation

Interface Bandwidth Comparison

PCIe3.0 vs. DDR4



Copyright © 2022 Intel Corporation

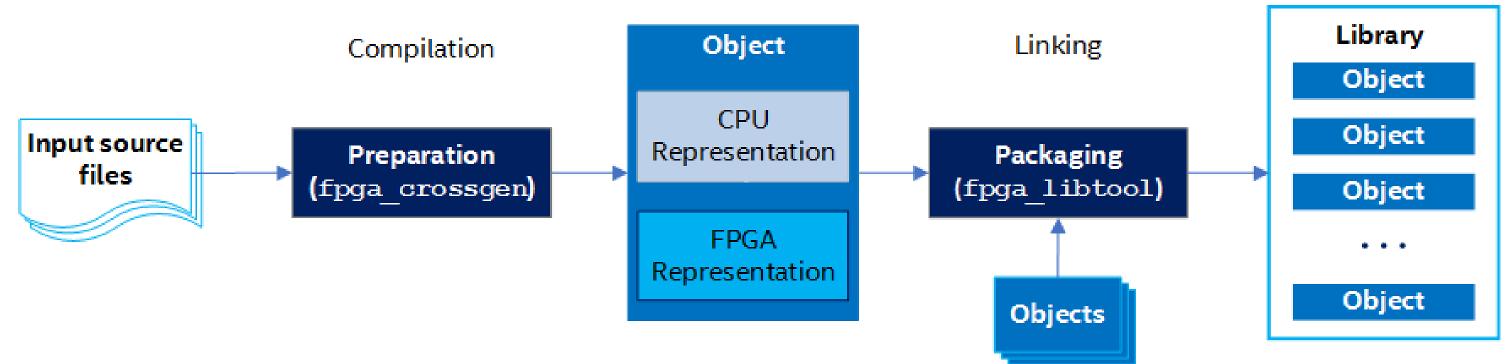
intel.

Use of RTL Libraries for FPGA in oneAPI

- Create a static library file using RTL
  - fpga_crossgen: rtl -> object
  - fpga_libtool: objects -> library

  Files needed:
    - RTL wrapper
    - XML description
    - Emulation model file (SYCL-based)

  RTL design constraints:
    - RTL module must have a clock port, a resetn port, and Avalon® streaming interface input and output ports
    - A single pair of ready and valid logic must control all the inputs
    - Declare the RTL module as stall-free possible



- Include library file to use the functions inside your SYCL* kernels.

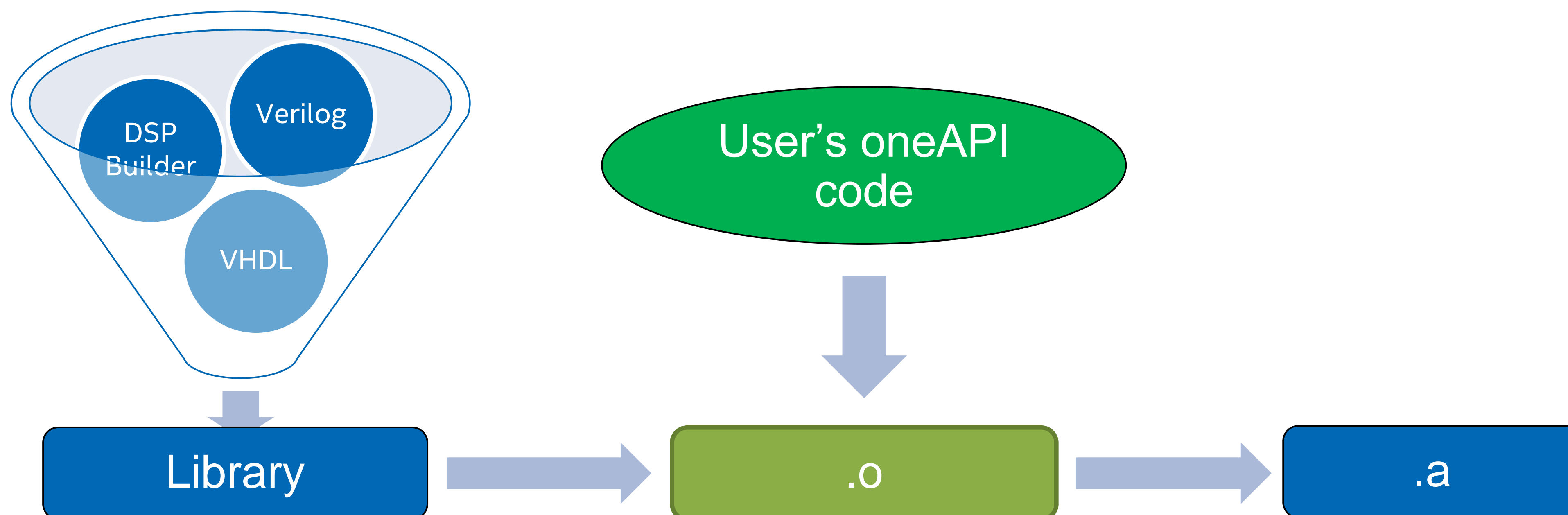
```
dpcpp -fintelfpga main.cpp lib.a
```

Copyright © 2022 Intel Corporation

intel.

Use of Library

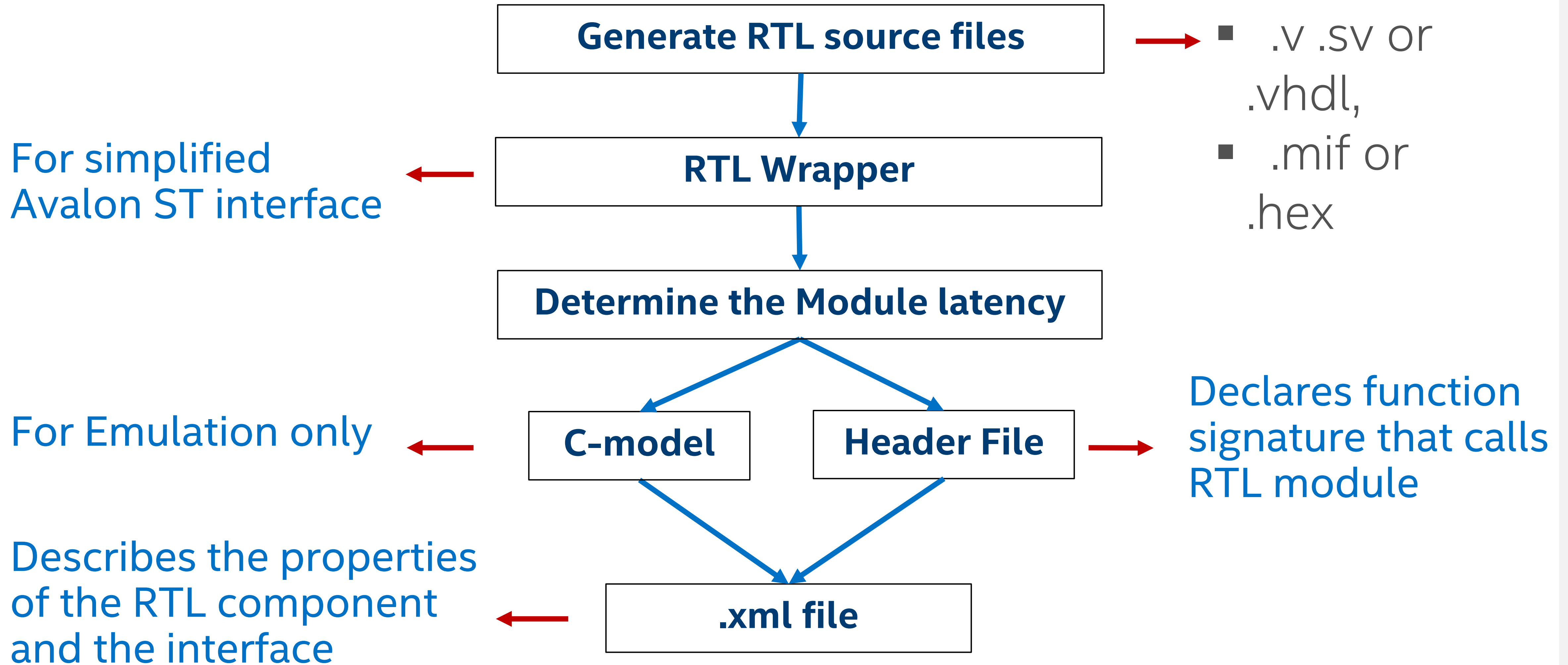
- Create libraries from RTL or oneAPI source and call those library functions from User OpenCL code



Copyright © 2022 Intel Corporation

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos

Development flow of creating RTL based call



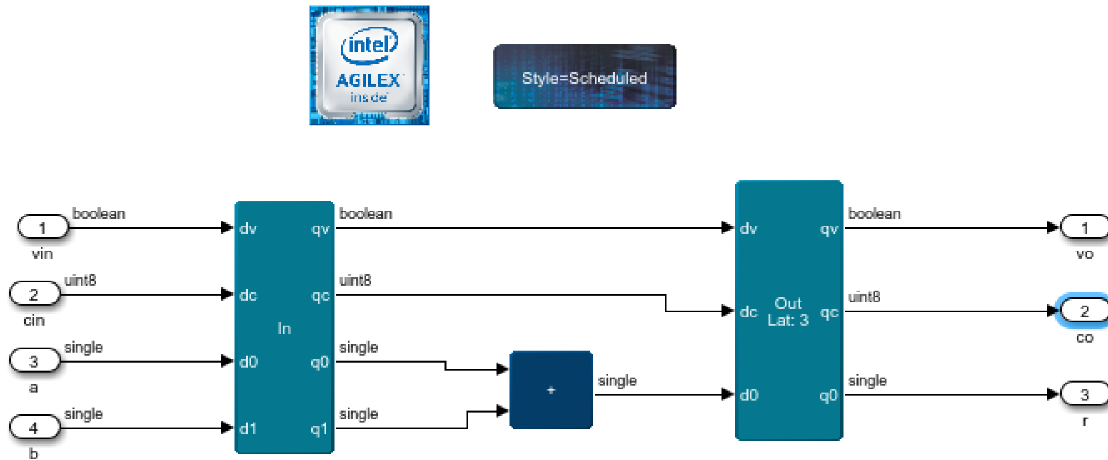
Copyright © 2022 Intel Corporation

.xml File Format

```
<RTL_SPEC>

  <FUNCTION module="myadd_module" name="myadd">
    <ATTRIBUTES>
      <IS_STALL_FREE value="yes"/>
      <IS_FIXED_LATENCY value="yes"/>
      <EXPECTED_LATENCY value="3"/>
      <CAPACITY value="3"/>
      <HAS_SIDE_EFFECTS value="no"/>
      <ALLOW_MERGING value="yes"/>
    </ATTRIBUTES>
    <INTERFACE>
      <AVALON port="clock" type="clock"/>
      <AVALON port="resetn" type="resetn"/>
      <AVALON port="ivalid" type="ivalid"/>
      <AVALON port="iready" type="iready"/>
      <AVALON port="ovalid" type="ovalid"/>
      <AVALON port="oready" type="oready"/>
      <INPUT port="idata" width="64"/>
      <OUTPUT port="odata" width="32"/>
    </INTERFACE>
<REQUIREMENTS>
   <FILE name="dspba_library_package.vhd"/>
   <FILE name="dspba_library.vhd"/>
   <FILE name="dspba_library_ver.sv"/>
   <FILE name="dspba_sim_library_package.vhd"/>
   <FILE name="dspba_s10_optimized_package.vhd"/>
   <FILE name="dspba_s10_optimized.sv"/>
   <FILE name="../efi/efi_dut_safe_path_flat.vhd"/>
   <FILE name="../efi/flt_efi_dut_addBlock_typeSFloatIEEE_23_8_typeSFloatIEEE_23_8_typeSFloatIEEE_23_80000a6354c2463a0c2462a5u.vhd"/>
   <FILE name="../efi/efi_dut.vhd"/>
   <FILE name="myadd_module.sv"/>
  </REQUIREMENTS>
  <RESOURCES>
   <ALUTS value="0"/>
   <FFS value="0"/>
   <RAMS value="0"/>
   <MLABS value="0"/>
   <DSPS value="1"/>
  </RESOURCES>
 </FUNCTION>

</RTL_SPEC>
```

Copyright © 2022 Intel Corporation

DSP Builder for Intel FPGAs



- Define the data types
- Build your algorithm
- Specify target clock frequency for the right amount of pipelining
- Create all the collateral for library development

Software model generation

☑ Enabled

Compiler: None (no automatic build)

Software model generation will also create an automated test bench (ATB) executable in the 'atb' subfolder. This executable tests the generated model by driving it with the input stimulus files and testing it against the output stimulus files created by Simulink.

The ATBs are recommended as a starting point for understanding how the generated software model might be integrated into an existing system.

HLD library function generation

Outputs a C wrapper for the generated software model, a System Verilog wrapper for the generated RTL and scripts for library compilation. Note that capacity will be assumed to be equal to latency, please adjust in the generated XML accordingly if this is not the case.

☑ Enabled

Target: oneAPI

Component type: Stateless

☐ Pack valid in data struct ☐ Variable latency

Instance count: 1

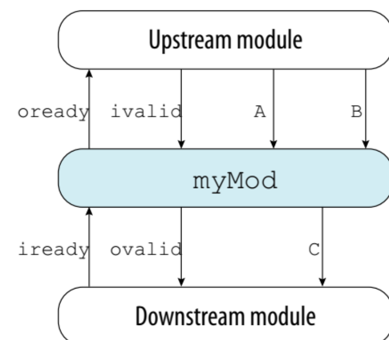
Note: The instance count can be modified after generation by defining the FUNCTION_INSTANCE_COUNT value during compilation.

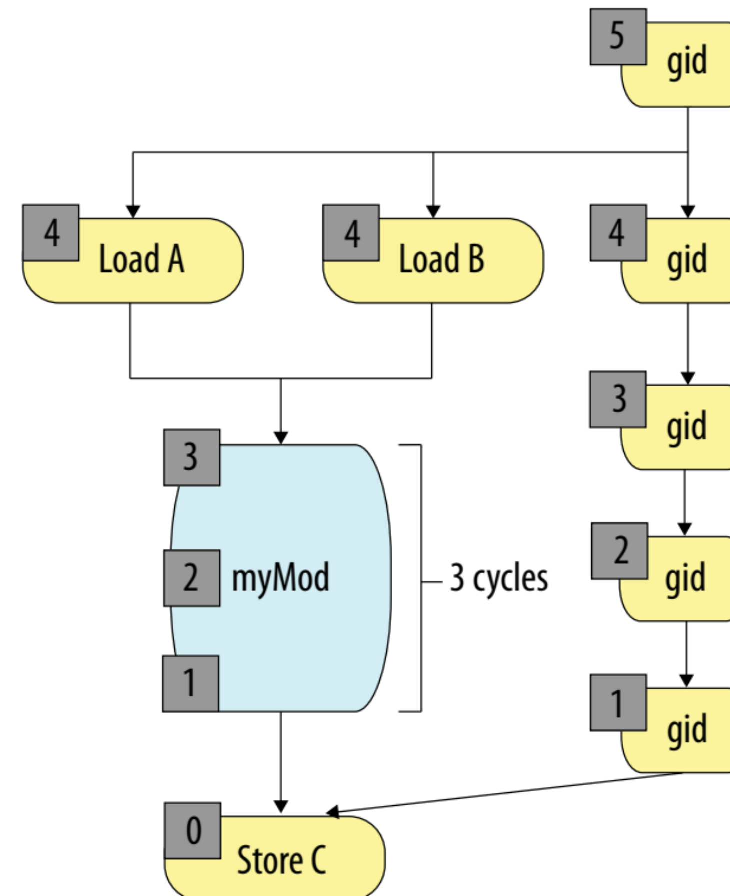
Function Name: myadd

Copyright © 2022 Intel Corporation

Integration of an RTL Module into the Pipeline Architecture

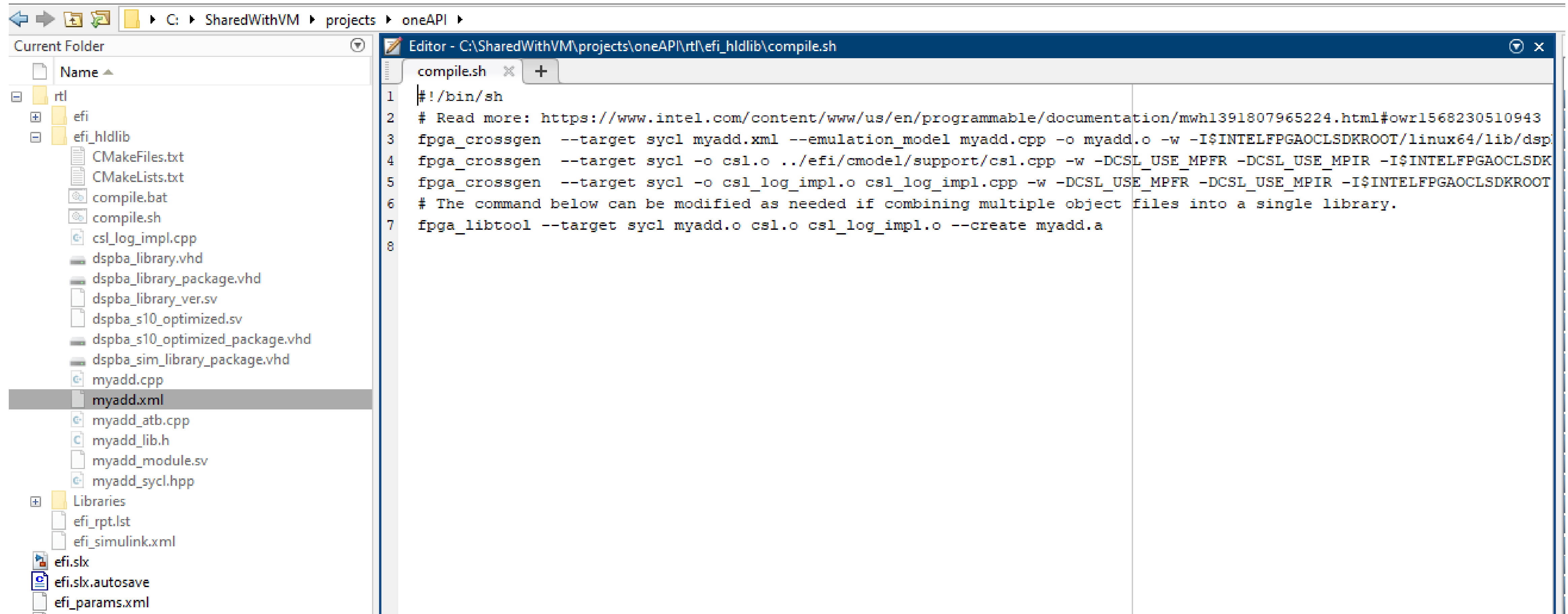
```
extern int myMod(int, int);
void kernel pe(global int* A,
               global int* B,
               global int* C){

    int gid = get_global_id(0);
    int a = A[gid];
    int b = B[gid];
    C[gid] = myMod(a, b);
}
```





Copyright © 2022 Intel Corporation

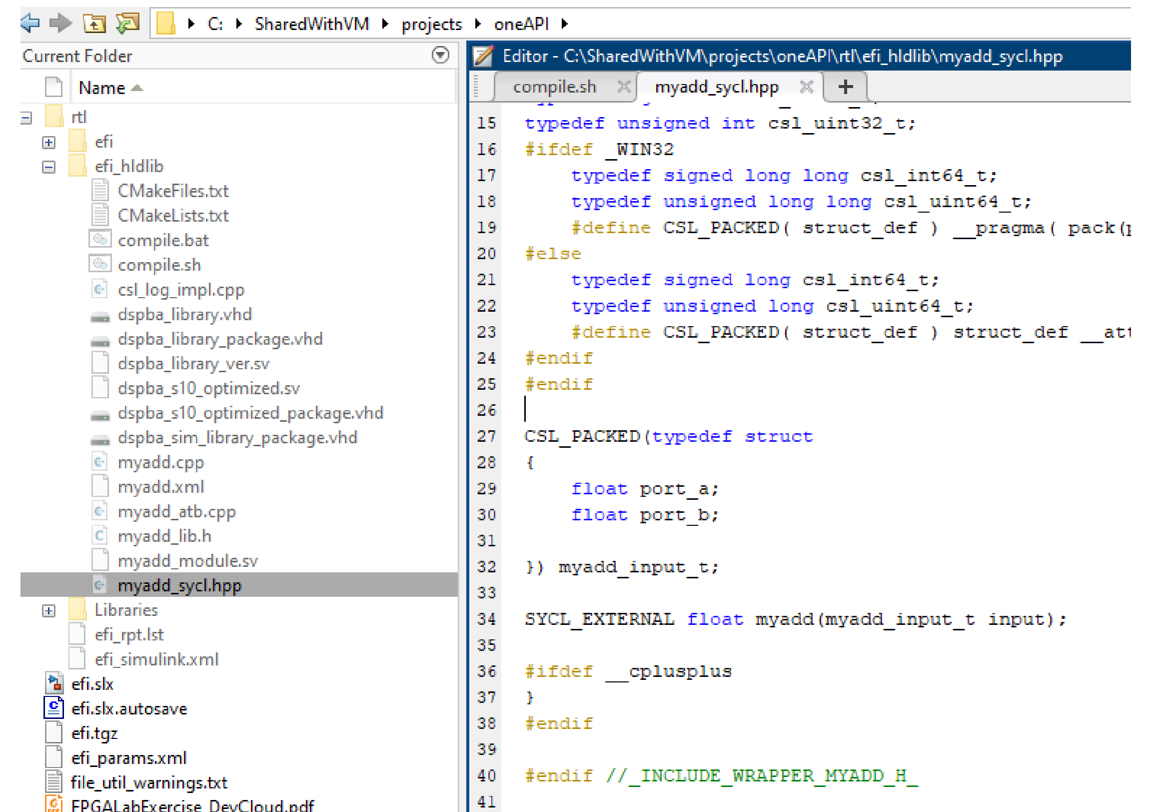
Generating the component



Copyright © 2022 Intel Corporation

Integration and Use of your custom function

- oneAPI-samples-master/DirectProgramming/DPC++/DenseLinearAlgebra/vector-add
- Change the makefile
  - Copy myadd.a to the src folder, myadd_sycl.hpp to top folder
- Add the header file #include “myadd_sycl.hpp”
- Change your function call from “+” to “myadd”
  - Use the intermediate struct autogenerated by the DSP Builder wrapper
- Run emulation to validate your changes (uses autogenerated C model for your design)
- Test on HW



Copyright © 2022 Intel Corporation

intel®