HLS Programming Guideline

Basic

Advanced Topic

Tutorial Example

1. Introduction

Vishare hls_tools are primarily used to simplify the HLS synthesis and connection process. It can automatically generate relevant SystemVerilog test cases and verify the results in ModelSim. Additionally, it generates an appropriate cpp file which contains the hardware function call interface, which can be run on riscv simulator (asim).

Features:

1. Identify and extract the functions that are surrounded by the IMPL() macro from the source file 'hls.cpp'
2. Automatic generate code to capture test data and create test cases for the above functions
3. Run Vitis HLS to batch synthesis the HLS functions
4. Capture the test data
5. Batch verify the test cases in Vitis co-simulation
6. Generate RTL function and testbench by autonet to run in ModelSim
7. Export the rtl code could be copied and placed to RTL code to simplify the process and minimize the typo error
8. Generate the hardware function call interface for RISC-V

IMPL() macro should be added for each HLS function so that the hls_tools can identify and automatically generate additional functions for capturing data and performing co-simulation. Additionally, several interfaces are provided for HLS functions to share the data with RISC-V. They are XCACHE, DCACHE and APCALL arguments, which will be described in a later section.

2. System Synthesis Steps

- Identify a list of accelerator functions
- Determine the data structured to be mapped to xcache
- Based on the bandwidth and storage requirements of accelerated functions, define the number of cache banks, widths of different banks and storage capacity of each bank
- Rewrite the C code to make use of the hardware accelerator and xcache
    - define the functions
    - declare the data in xcache data structure
- Synthesis with accelerator hardware with HLS
- Synthesis the system with
    - decide the parameter of standard components
        - RISC core number
        - L1 cache size
        - L2 cache size
- Synthesis the RTL code
- Perform simulation
- Check the correctness and evaluate the performance on the FPGA board

3. Installation

3.1 Prerequisite

Vitis HLS should be installed.
https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis.html

In the Windows platform, please execute the hls_tools in the WSL environment.

Python 3.12 should be used (which is the default python version shipped in Ubuntu 24.04).
Please contact us to generate a new package if a different python environment is used.

Please install below package:

```
sudo apt update

sudo apt install python3 python3-pip python3-venv gcc-multilib
g++-multilib libclang-dev clang cmake build-essential

pip3 install libclang
pip3 install tabulate
pip3 install colorama
```

If it has an error on installing libclang and colorama, please type below command to activate python virtual environment and install relevant python package on it.

```
sudo apt install python3-venv
python3 -m venv myenv
source myenv/bin/activate
pip3 install libclang
pip3 install tabulate
pip3 install colorama
```

3.2 Install RISC-V compiler toolchain

```
wget -O riscv32-toolchain.tar.gz
https://github.com/stnolting/riscv-gcc-prebuilt/releases/download/rv32i-
4.0.0/riscv32-unknown-elf.gcc-12.1.0.tar.gz
sudo mkdir -p /opt/riscv
sudo tar -xzf riscv32-toolchain.tar.gz -C /opt/riscv/
rm -f riscv32-toolchain.tar.gz
```

```
#append the toolchain to the PATH variable in .bashrc
echo 'export PATH=$PATH:/opt/riscv/bin:/opt/asim' >> ~/.bashrc

source ~/.bashrc
```

4. Run the Example

The example is placed in the hls_tools/tutorial_example directory

1. Specify the vitis_hls path in run.sh. You could also modify the project path and export destination in run.sh according to different projects.

```
cd tutorial_example/hls_tools_script

nano run.sh

# Edit the following variable to specify the executable path:

# xilinx_vitis=/mnt/d/Xilinx/Vitis_HLS/2023.2/bin/vitis_hls
```

1. Synthesis all HLS functions

```
./run.sh xhls all
```

2. (Optional) Capture the test data

```
./run.sh capture vector_add
```

#The default is to capture 10,000 times, you can manually capture a specific count by adding the desired number before the function```

```
./run.sh capture 1000 vector_add
```

3. (Optional) Run the Vitis C simulation

```
./run.sh csim vector_add
```

4. Connect all HLS to common bus interface

```
./run.sh xgen all
```

After finishing, relevant RTL and SystemVerilog testbench will be generated to the export destination. Simply copy the rtl code to a dedicated directory to run in the FPGA.

4.1 Other useful option:

- Specify the desired HLS function list in the file, separated by new lines. The parser will ignore the line starts with #

```
./run.sh xhls hls_func_list.txt
```

- Specify which HLS functions should be synthesized, accept multiple function name

```
./run.sh xhls vector_add
./run.sh xhls vector_add array_xor
```

- Specify the HLS function list that connect to the riscv

```
./run.sh xgen hls_func_list.txt
```

5. Detail Operation

When execute ./run.sh command, it will

1. Scan and extract the function list from the hls source file hls.cpp

2. Generate hls_enum.h (`enum_header_file`) to assign unique ID for each HLS function

3. Automatic generate the source code used to capture the test data and c testbench for Vitis (hls_capture.cpp and hls_tb.cpp) (`${hls_src_file}_capture.cpp`, `${hls_src_file}_tb.cpp`) which could be used in below capture, csim, cosim and cosimwave argument.

4. Depends on the current input argument to perform below operation

argument	**Description**
xhls	batch synthesized all HLS functions by calling \vitis_batch_generate_rtl\gen_hls script
capture	rebuilt the source file with macro `CAPTURE_COSIM=1`, the wrapper function CAPTURE_(hls_func) will be invoked to dump the input / output data.
csim	perform Vitis C simulation by using hls_tb.cpp
cosim	perform Vitis cosimulation by using hls_tb.cpp
cosimwave	perform Vitis cosimulation with waveform shown by using hls_tb.cpp
xgen	It involves several operation: 1. Reorder the XCACHE element in xmem.h (Classified as scalar type, array type, cyclic type) 2. Generate XCACHE verilog connection 3. Connect the HLS functions to the bus 4. Generate verilog testbench in the autonet/export/tutorial_example 5. Generate C routine in hls_apcall.h and hls_apcall.cpp to invoke the hardware function

6. HLS Arguments

XCACHE, DCACHE and APCALL arguments could be used to share the variable between RISC-V and these HLS functions.

6.1 XCACHE

Xcache is a shared application specific cache among different HLS functions and RISC-V. The number of banks, data width and capacity can be optimized to meet the bandwidth and storage requirement of different accelerators. Meanwhile, RISC software can still access XCACHE via C code. Access times can vary depending on the cache miss rate. Currently, the size of XCACHE is configured to limit at 256MB.

6.2 DCACHE

The DCACHE interface allows the HLS module to access the standard D-cache embedded in each RISC-V core. Access times can vary depending on the cache miss rate and are generally slower than those of the XCACHE variables.

6.3 APCALL argument

You can choose to pass the arguments using the APCALL argument. The significant limitations include: support for input type only, a maximum of 8 arguments and a maximum width of 32 bits in each argument. Generally, using the XCACHE region is a better way to share data between HLS and RISC-V.

6.4 Example:

```
typedef struct {
    int x   _ALIGN;
    int y   _ALIGN;
}vector_2d;

typedef struct xmem_t{
    uint8_t xor_val8                                    _ALIGN;
    uint16_t xor_val16                                  _ALIGN;
    uint32_t xor_val32                                  _ALIGN;
    int arr_complete[5]                                 _ALIGN;
    vector_2d vec_s1                                    _ALIGN;
    vector_2d vec_s2                                    _ALIGN;
    vector_2d vec_d1                                    _ALIGN;
    int tc_arr[2]                                       _ALIGN;
    uint8_t no_p_arr[2]                                 _ALIGN;
    uint8_t no_q_arr[2]                                 _ALIGN;

    int arr_s1[10]                                      _ALIGN;
    int arr_s2[10]                                      _ALIGN;
```

```
    int arr_d1[10]                                  _ALIGN;
    uint8_t pix_base[128*128]                       _ALIGN;
}xmem_t;
```

1. It is recommended to add the xm_ prefix to distinguish the variable from XCACHE and APCALL arguments

2. _ALIGN macro should be added to each variable

3. The _i and _o postfixes generated by the HLS function are ignored. In other words, vec_i, vec_o, vec are treated as the same variable

The argument not in the XCACHE will be treated as APCALL argument. Therefore, if the argument is declared as an array and is not declared in XCACHE, during connection with XCACHE, it will show the error message because APCALL only accepts 8 maximum 32-bit native size of variables.

Programming constraints

- Recommend to add the ‘xm_’ prefix to identify the XCACHE data member
- must add the _ALIGN macro to each variable declared inside the XCACHE data structure
- Cannot use the _i and _o suffix
- The number of function call arguments must be less than 8
- Does not support array of structures inside XCACHE

Performance optimization

- If an HLS function has multiple DCACHE arguments, the hardware can only access one DCACHE argument in each cycle and so it takes multiple cycles to access all arguments. Furthermore, frequent and high speed data cache access may pollute the cache data used by other software functions, resulting in increasing cache miss cycles experienced by RISC cores.
- An HLS function with multiple XCACHE arguments may access different xcache data in parallel if they are connected with different xcache banks, thus sustaining the high memory bandwidth for enabling high accelerator throughput.

7. XCACHE

Generally, the data could be shared by XCACHE for fast and convenient access. Each core has its own XCACHE region.

```
typedef struct {

    int xm_a        _ALIGN;

    int xm_y        _ALIGN;

    int xm_arr[10]  _ALIGN;

    struct vector_2d xm_v _ALIGN;

}xmem_t;
```

Support datatype:

- Native var type (int, unsigned short, uint8_t, uint16_t, bool, enum....etc..)
- Structure var type
- Nested structure
- Array (complete array partition)
- Array (single port ap_memory interface)
- Array (cyclic partition)

Currently, it doesn't support arrays of structure (struct person[10]..etc..).

7.1 Attention:

1. The width of the variable in the XCACHE must match the memory fetch size of the load / store instruction. For example, a 1-byte data (uint8_t) declared in XCACHE should be accessed using dedicated LB (Load Byte) / SB (Store Byte) instructions, rather than the 2-byte or 4 byte instructions like SW (Store Word), LW(Load Word), LH (Load Halfword),SH (Store Halfword) and so on. To ensure that the correct instructions are used, you could utilize the XMEM_SET, XMEM_ASSIGN, XMEM_COPY related macros or xmem_assign(), xmem_set(), xmem_copy() C++ template functions.

2. To access an array starting from a specific index IDX, we must pass the offset as an additional argument. For example, if xm_dst and xm_src are the array types, and you want to perform an XOR operation starting from IDX, you should call the HLS function like this

   ```
   xor(xmem->xm_dst, xmem->xm_src, IDX);
   ```

You should not use the following statement:

```
//wrong
xor(xmem->xm_dst, &xmem->xm_src[IDX]);
```

Using the second statement is incorrect in hardware terms, as it does not properly account for the intended offset and can lead to erroneous operations.

8. DCACHE

If the variable size is too large / not suitable to fit into xmem, you could use the DCACHE interface to access the variable content in the risc-v.

Usage:

```
void IMPL(dcache_example_hls)(DCACHE_ARG(uint8_t, table, 38400),
uint32_t dcache[DCACHE_SIZE]){

    uint8_t *table = (uint8_t*)DCACHE_GET(table);



    //... access the variable. for example,

    uint8_t lookup_value = table[N];

}
```

Please note that the access time depends on the cache miss rate, if the data is not inside the cache, the access time will be longer.

A.1 Parent and Child Function

You could use HLS_CHILD_CALL() to call a HLS child function in the parent HLS function.

```
HLS_CHILD_INIT();

HLS_CHILD_CALL(hls_id, arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7);
```

In the parent HLS function, HLS_CHILD_INIT() should be invoked first before any HLS child function.

If the variable is modified by a child function, the volatile keyword should be added to the parent function's argument so that Vitis HLS can identify the data that might change after HLS_CHILD_CALL is involved.

Using the flow detection feature in asim can identify the variable dependency between the parent function and the child function, making it easier to determine which arguments should add the volatile keyword.

Prerequisite

1. The flow detect option should be enabled in asim
2. The riscv ELF should be in single-thread mode (without pthread_create) because asim only supports running flow detection in single-thread mode.

Procedure

1. Run the elf with asim, ensure that the flow detection option is enabled.

```
#./asim --elfFileName program.elf program {additional argument}

./asim --elfFileName example.elf example
```

2. After asim executed, the report will be exported to out/dataflow.csv

3. In libreoffice, open the csv file and apply the standard filter with below expression:

- XmemOffset <> None
- Read Function contains [parent function]

Check the **field** column. If it shows **child write parent read**, it means that the dedicated xmem variable is altered by the child function and subsequently read by the parent function. Therefore, **volatile** keywords should be added to the associated argument.

T.1 Tutorial Example

1.1 Port C code to HLS

In hls.cpp, suppose below cnn_hls function which consists of 3 inputs and 2 arrays type. According to the vitis HLS user guide, the output argument in C should be declared as a pointer / reference type. And the array type must explicitly specify the array size, such as uint8_t filter_map[8x8] rather than uint8_t *filter_map.

```
void cnn_hls(int width, int height, int filter, char *pixel, char
*filter_map, int *sum)
```

The IMPL() and HLS_COMMON_ARG macro should be added to these HLS functions so that it can be recognized by hls_tools and generate valid hardware modules.

IMPL(<function_name>)(HLS_COMMON_ARG <arguments…>)

Hence, the cnn function argument rewrite as below where pixel and sum are array type, width, height and filter are input type:

```
void IMPL(cnn_hls)(HLS_COMMON_ARG int width, int height, int filter, char
pixel[(MAX_WIDTH_SIZE + MAX_FILTER_SIZE -1 ) * (MAX_HEIGHT_SIZE +
MAX_FILTER_SIZE - 1)], char filter_map[MAX_FILTER_SIZE * MAX_FILTER_SIZE], int
sum[MAX_WIDTH_SIZE * MAX_HEIGHT_SIZE])
```

pixel and sum should also be declared in XCACHE.

```
typedef struct xmem_t{
//...
    char filter_map[MAX_FILTER_SIZE * MAX_FILTER_SIZE] _ALIGN;
    char pixel[(MAX_WIDTH_SIZE + MAX_FILTER_SIZE -1 ) * (MAX_HEIGHT_SIZE +
MAX_FILTER_SIZE - 1)] _ALIGN;
//...
}xmem_t;
```

In HLS, the array type could be customized by using different pragma options to implement one of the following memory type:

- generic ap_memory interface (default)
- complete array partitioning
- cyclic array partitioning

ap_memory means that during synthesis, the array will be treated as single / dual port memory which consists of ce (enable), q (read data port) and we (write enable), d (write data port) signal.

Complete partitioning decomposes the array into individual elements.In other words, a[5] will be decomposed into a_0, a_1, a_2, a_3, a_4 individual variable during synthesis.

Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. Suppose the pragma cyclic factor = 4, it will create 4 ap_memory with each memory size divided by 4.

Please refer to AMD Vitis High Level Synthesis User Guide (UG1399) for detailed description.

Both native and array types are supported in XCACHE as below.

Argument Type	Region	#HLS pragma should be added
input native type <= 32-bit (e.g. uint32_t, uint8_t, int)	XCACHE / APCALL argument	(none)
input native type > 32bit (e.g. uint64_t, long long, double)	XCACHE	(none)
output native type (pointer or reference)	XCACHE	(none)
array	XCACHE	(none)
array (complete partition)	XCACHE	#pragma HLS array_partition variable=<name> type=complete_partition
array (cyclic)	XCACHE	#pragma HLS array_partition variable=<name> type=cyclic factor=<int>
struct	XCACHE	#pragma HLS disaggregate variable=<variable>

Table 2: Argument Type in XCACHE region

After executing the `./run.sh xgen` command, the script will parse the Verilog code and the XCACHE structure, reordering the element in xmem.h if necessary. It will classify and reorganize the xmem into three categories: scalar, array, and cyclic.

1.2 hls_tools usage

Basic example is provided to demonstrate how to use the hls_tools

1. Please follow the installation section to install required components.
2. Configure the Xilinx Vitis path

```
cd tutorial_example
cd hls_tools_script
```

Please edit $xilinx_vitis in ./run.sh to point to Vitis HLS program

```
xilinx_vitis=/mnt/d/Xilinx/Vitis_HLS/2023.2/bin/vitis_hls
```

Ensure the script has executable permission

```
chmod +x ./run.sh
```

3. Synthesis HLS functions To synthesis all HLS functions, type:

```
./run.sh xhls all
```

To synthesis individual function, for example, cnn_hls, type:

```
./run.sh xhls cnn_hls
```

4. (Optional) Capture the test data The default is to capture the input/output argument of specify function 10000 times

```
./run.sh capture vector_add
```

You can customize the capture times by specify the number:

```
./run.sh capture 10 vector_add
```

5. (Optional) Run the C simulation of the HLS function

```
./run.sh csim cnn_hls
```

6. (Optional) Run the co-simulation of the HLS function

```
./run.sh cosim cnn_hls
```

7. (Optional) Run the co-simulation with the waveform generated by the HLS function

```
./run.sh cosimwave cnn_hls
```

8. Link up the HLS functions and generate the relevant systemVerilog testbench by tying:

```
./run.sh xgen all
```

Relevant Verilog files will be generated under the autonet/export/tutorial_example directory The hls_apcall.cpp will also be updated in the tutorial_example project, which served as the hardware function call interface that can be checked on the RISC-V simulator (asim)

9. Run on RISC-V simulator (asim)

```
cd ../build/riscv-sim
source ./make-Makefiles.sh
source ./run.sh
```