Programming FPGAs for Economics
An Introduction to Electrical Engineering Economics

TUTORIAL
[WORK IN PROGRESS]

by

BHAGATH CHEELA
University of Pennsylvania, Electrical and System Engineering
cheelabhagath@gmail.com

ANDRÉ DeHON
University of Pennsylvania, Electrical and System Engineering
andre@seas.upenn.edu

JESÚS FERNÁNDEZ-VILLAVERDE
University of Pennsylvania, Economics
jesusfv@econ.upenn.edu

ALESSANDRO PERI
University of Colorado, Boulder, Economics
alessandro.peri@colorado.edu

Last Update: Friday 6th January, 2023

Acknowledgements

First, we wish to thank Syed Ahmed (UPenn, Electrical and System Engineering). The material in Chapters 2 and 3 is built on the teaching material created by Syed for the ESE 532 Class offered at UPenn. Chapter 3 draws on the tutorial created by Xilinix, Inc. Second, we wish to thank Lucas Ladenburger and Marina Leah Mccann (CU Boulder, Economics) for helping building this tutorial. Last but not least, we wish to thank Giuseppe Bruno and Riccardo Russo (Bank of Italy) for their help in testing a previous version of this tutorial. This project was funded by the Undergraduate Research Experiences for Diversity Grant, 2021, Institute of Behavioral Science, University of Colorado, USA. This project used the RMACC Summit supercomputer, supported by the National Science Foundation (awards ACI-1532235 and ACI-1532236), the University of Colorado Boulder, and Colorado State University. This project was also supported by the Undergraduate Research Experiences for Diversity Grant, 2021, Institute of Behavioral Science, University of Colorado, USA.

Contents

CHAPTER 1

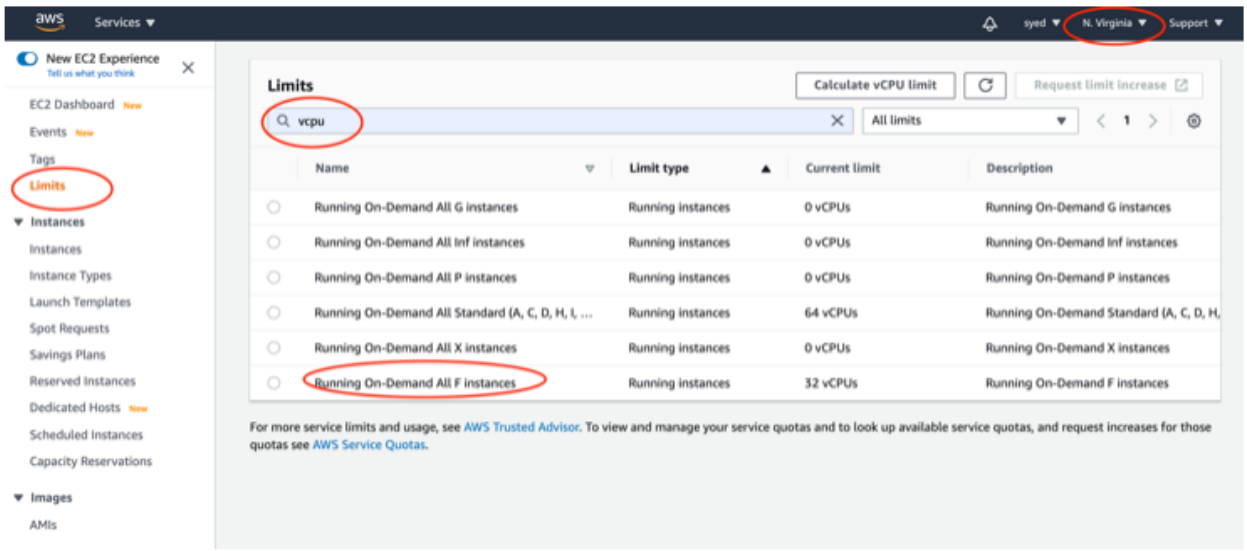
Setup and Walk-through

To implement a function in hardware (e.g., the Krusell and Smith (1998) algorithm), it will ultimately be necessary to perform low-level placement and routing of the hardware onto the FPGA substrate. That is, the tools must decide which particular instance of each primitive is used (placement) or which wires to use for connections (routing). These tasks take typically longer time (at least 30 minutes, sometimes hours) than the compilation time for software (a few minutes). This means you will need to plan your time carefully for these tutorials. One way to optimize our development time is to be careful about when we invoke low-level placement and routing and when we can avoid it. The content of this chapter was curated by Syed Ahmed.[1]

1.1 Getting Started with Vitis on Amazon F1 Instance

Make sure you complete the following pre-requisites before continuing with this tutorial:

1. You have an AWS account and know how to create AWS instances. Check Getting Started on Amazon EC2 for a refresher.



2. Read about Vitis from here.

[1]University of Pennsylvania, Electrical and System Engineering. *email:* stahmed@seas.upenn.edu

In this tutorial, we will use two instances:

- z1d.2xlarge referred to as the ***build*** instance where we will compile and build our FPGA binary. It costs 0.744 $/hr. You can create this instance in any AWS region.
- f1.2xlarge referred to as the **runtime** instance where we will run our FPGA binary. It costs 1.65 $/h. We can only use us-east-1 (N. Virginia) for this instance.

1.2 Step 1: Launch the build instance

1. Navigate to the AWS Marketplace
2. Click on **Continue to Subscribe**
3. Accept the EULA and click **Continue to Configuration**
4. Select version v1.10.0 and US East (N.Virginia)
5. Click on **Continue to Launch**
6. Select **Launch through EC2** in the *Choose Action* drop-down and click **Launch**
7. Search and select **FPGA Developer AMI**
8. Select z1d.2xlarge Instance type from the Filter **All instance families**
9. At the top of the console, click on **6. Configure Security Groups**
10. Click **Add Rule**. Note: Add a new rule. Do NOT modify existing rule.
    (a) Select **Custom TCP Rule** from the **Type** pull-down menu
    (b) Type 8443 in the **Port Range** field
    (c) Select **Anywhere** from the Source pull-down

    Note: This steps will enable us to install a NICE DCV Server on the instance.
11. Click **Review and Launch**. This brings up the review page.
12. Click **Launch** to launch your instance.
13. Select a valid key pair and **check** the acknowledge box at the bottom of the dialog
14. Select **Launch Instances**. This brings up the launch status page
15. When ready, select **View Instances** at the bottom of the page
16. Login to your build instance by doing:

```
ssh -i <AWS key pairs.pem> centos@<IPv4 Public IP of EC2 instance>
```

1.3 Step 2: Setup remote desktop

We will use **NICE DCV** as our remote desktop server on Amazon. We will use the remote desktop to work with several **Vitis GUI** utilities. For the setup we follow the Amazon GUI FPGA Development Environment with NICE DCV Tutorial.

1. Attach **NICE DCV** license to your z1d.2xlarge instance by doing the following:
   (a) Sign in to the **AWS Management Console** and open the **IAM console** at link.
   (b) In the navigation pane of the IAM console, choose **Roles**, and then choose **Create role**.
   (c) For **Select type of trusted entity**, choose **AWS service**.
   (d) For **Choose a use case**, select **EC2** and then click **Next: Permissions**.
   (e) Click on **Next: Tags** to move forward.
   (f) Click on **Next: Review** to move forward.
   (g) Enter a name, e.g. *"DCVLicenseAccessRole"* and click **Create role**.
   (h) Click on **Policies** in the left menu.
   (i) Click on **Create policy**.
   (j) Click on the **JSON** tab and paste the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::dcv-license.us-east-1/*"
        }
    ]
}
```

   Note: The NICE DCV software needs to access the NICE DCV license, and the license is located in the s3 bucket. Change us-east-1 to the region you are using (if different). For more information, see link..
   (k) Click on **Next: Tags** to move forward.
   (l) Click on **Next: Review** to move forward.
   (m) Enter a name, e.g. *"DCVLicensePolicy"* and click **Create policy**.
   (n) Search for your new policy and click on it to open it.
   (o) Click on **Policy usage** and then on **Attach**.

(p) Enter your DCV role name, select the role and click on **Attach policy**.

(q) Go to your console home page and click on **Instances**.

(r) Right-click on your z1d.2xlarge instance and click on ***Security*** and then **Modify IAM role**.

(s) From the drop-down menu, select your DCV role name and click save. Your instance will now be able to use the server.

2. Login to your z1d.2xlarge instance and install NICE DCV pre-requisites. More info at link.

```
sudo yum update
sudo yum install kernel-devel
sudo yum groupinstall "GNOME Desktop"
sudo yum install glx-utils
```

Note: You may receive the message: **Failed to set locale, defaulting to C**. Locales define language and country-specific setting for your programs and shell session. If you want to fix it (not required) you can follow the instructions at this link.

3. Install also the crudini rpm package to modify the nice dcv server configuration preferences (see more here).

```
sudo yum install crudini
```

4. Install NICE DCV Server. More info at link.

```
sudo rpm --import https://s3-eu-west-1.amazonaws.com/nice-dcv-publish/NICE-GPG-KEY
wget https://d1uj6qtbmh3dt5.cloudfront.net/2019.0/Servers/nice-dcv-2019.0-7318-el7.tgz
tar xvf nice-dcv-2019.0-7318-el7.tgz
cd nice-dcv-2019.0-7318-el7
sudo yum install nice-dcv-server-2019.0.7318-1.el7.x86_64.rpm
sudo yum install nice-xdcv-2019.0.224-1.el7.x86_64.rpm
cd ~

sudo systemctl enable dcvserver
sudo systemctl start dcvserver
```

5. Setup a password

```
sudo passwd centos
```

6. Change firewall settings: Disable firewall to allow all connections

```
sudo systemctl stop firewalld
sudo systemctl disable firewalld
```

7. Create a virtual session to connect to.
   Note: You will have to create a new session if you restart your instance. Put this in your /.bashrc so that you automatically create a session on login..

```
dcv create-session --type virtual --user centos centos
```

8. Connect to the DCV Remote Desktop session
   - Download and install the DCV Client in your computer[2].
   - Use the Public IP address to connect
9. Logging in should show you your new GUI Desktop

1.4 Step 3: Setup AWS CLI

1. Go to the Amazon AWS Console and then from the top right, select your account name, and then **My Security Credentials**.
2. Click on Access Keys and **Create New Access Key**.
3. Note down your **Access Key ID** and **Secret Access Key**.
4. Login to your z1d.2xlarge instance and issue the following command:

```
aws configure
```

5. Enter your access key, add us-east-1 as region and output to be json.

1.5 Step 4: Edit Source Files in Build Instance.

To edit your source files, you can use vim or emacs directly in the remote terminal. Or you can ssh from an editor in your local machine to edit files remotely. For instance: Remotely edit files using SSH from VS Code in Mac/Linux/Windows.

1.6 Step 5: Build Phase

The build phase is conducted entirely in the z1d.2xlarge instance. The build phase consists of

- **Profiling of the Code**, where you use the ***Vitis Analyzer*** to figure out bottlenecks in your application. To learn how to use ***Vitis Analyzer*** read here.
- **Synthesis of the Code**, which create the AFI executable which you can run on the f1 instance

[2] **IMPORTANT:** use the 2020.2 version. The latest version is not otherwise compatible with the setup.

In order to profile and synthesize your code you need to use the ***Vitis HLS*** software. This section guides you on the steps on how to launch Vitis, create a ***Project*** in Vitis. The next chapters discuss the Code profiling and Synthesis in the context of the different applications.

1.6.1 Initialize the Environment

If you are just starting a new project from scratch,

1. Login to your instance and initialize your environment as follows:

```
tmux
git clone https :// github.com/aws/aws-fpga.git $AWS_FPGA_REPO_DIR
source $AWS_FPGA_REPO_DIR/vitis_setup.sh
export PLATFORM_REPO_PATHS=$(dirname $AWS_PLATFORM)
```

   Note: Make sure to run under tmux! It will save you hours..

2. Clone your git repository using the following command:

```
git clone GETYOURREPO
```

These are one-time operations which you do not need to repeat later.

1.6.2 Create a Project in Vitis HLS

Creating a new project in Vitis HLS is explained here. Make sure you enter the **top-level function** during the creation of the project (although you can also change it later). The top-level function is the function that will be called by the part of your application that runs in software. ***Vitis HLS*** needs it for synthesis. You can also indicate which files you want to create. It is wise to add a **Testbench file** too, while you are creating the project, to check that your application runs correctly.

1. To get started
   (a) Launch (or restart) your z1d.2xlarge in AWS
   (b) In a terminal, ssh into your z1d.2xlarge instance (wait for the instance to be ready!). Start the DCV server using the following:

```
        dcv create-session --type virtual --user centos centos

```

   Note: This command launches a DCV session in the building instance to which you can connect remotely from your computer.

   (c) Open the NICE DCV Viewer in your computer
   - Enter the public IP address of the z1d.2xlarge instance.

- Enter centos as user and the password you set during DCV setup.

You should now the see the desktop of your building instance!!

2. To launch the ***Vitis HLS*** Software
   (a) In the desktop of your building instance, select ***Applications*** > ***System Tools*** > ***Terminal***
   (b) Launch ***Vitis HLS*** by typing vitis_hls & in the terminal. You should now see the Integrated Development Environment (IDE).
3. To create a ***New Project***
   - In the drop-down click on ***File*** and select ***New Project***
   - Give a name to the Project and select the location where to store the project.
   - Specify TBD as top function.
   - Add to the source files
     - all the .c files
     - all the .h files
   - Add Testbench.cpp to the TestBench files
   - Select the xcvu9p-flgb2104-2-i in the device selection.
   - Use a #CLOCK SPEED ns clock, and select ***Vitis Kernel Flow Target***.
   - Click Finish.

We will specialize the Project creation depending on the target application in the Chapters to come.

1.6.3 C Simulation and Code Debugging

We encourage you to implement a testbench file (e.g. Testbench.cpp) to debug your code. A testbench application is not different from any other software applications written in C:

- they have a main function that is invoked
- the main function includes any functionality needed to test your function, including calling the top function that you would like to test.
- they return 0 if the function is correct, otherwise it should return another value

To run the Testbench.cpp

1. Select ***Project*** → ***Run C Simulation*** from the menu.

- A window should pop up. The default settings of the dialog should be fine. You can dismiss the dialog by pressing ***OK***.

2. You can see in the ***Console*** whether your test has passed.
3. If your test fails, you can run the test in debug mode.
   - This can be done by repeating the same procedure, except that you should check the box in front of ***Launch Debugger*** this time before you dismiss the dialog.
   - This will take you to the ***Debug*** perspective, where you can set breakpoints and use the step into/step over buttons to debug.
4. You can go back to the original perspective by pressing the ***Synthesis*** button in the top, right corner. To rebuild the code, you should go back to Synthesis mode, and click ***Run C Simulation*** again to rebuild the code.

1.6.4 Synthesis in Vitis HLS

Once you have verified that the code is free of bugs, run ***Solution*** →***Run C Synthesis*** →***Active Solution*** from the menu to synthesize your design.

- ***C/RTL Cosimulation.*** You can also verify the synthesized version of your accelerator in your testbench. If you choose to do so, Vitis HLS will run your accelerator in a simulator, so this method is called C/RTL Cosimulation. The employed cycle-level simulation is much slower than realtime execution, so this method may not be practical for every testbench. It avoids needing to run low level-placement and routing and will give you more visibility into the behavior of your design. Anyway, you can start it by choosing ***Solution*** →***Run C/RTL Cosimulation*** from the menu.

The *Vitis HLS* Kernel

- The RTL export will produce an .xo file (Vitis Kernel)
- Then go to the terminal and use the makefile to create the xclbin

The Synthesis will produce a **Vitis Kernel**, that is a Xilinx object file (.xo) that describes the hardware implementation of our application.

The next section discusses how to optimize it.

1.6.5 HLS Kernel Optimization using the *Vitis HLS* IDE

The optimization follows a bottom-up approach

1. Profile the Code using the ***Vitis Analyzer*** . To learn how to use the ***Vitis Analyzer*** read here.

2. Optimize your hardware function using the ***Vitis HLS*** IDE;

   - ***Vitis HLS*** controls the hardware implementation wit the **#pragma** command. Examples:

   ```
   #pragma HLS unroll 2
   #pragma HLS pipeline
   ```

   The different **#pragma** that you can use are listed in the Vitis HLS User Guide.

3. Re-compile it;

4. Once you happy, you are ready to move the code to the FPGA

Note

We are using the GUI mode of ***Vitis HLS*** (using NICE DCV) so that we can see the HLS schedule. If your remote desktop connection is lagging, you can run ***Vitis HLS*** from the command line. You can learn more about the TCL commands from: link 1, link 2 Note that the only way to see the HLS schedule is through the GUI. If you are unable to use the GUI in AWS or try to install Vitis toolchain locally.

1.6.6 Compile the Hardware Function

Once you are happy with your ***Vitis HLS*** acceleration:

1. Export Vitis Kernel: When you have obtained a satisfying hardware description in ***Vitis HLS*** , you will Export Vitis Kernel, i.e. a Xilinx object file (.xo). We will then use this object file/kernel and link it together in our existing Vitis application.

2. **Compile a hardware function.** Build the hardware function by doing make afi EMAIL=<your email>, substituting your email. Depending on the complexity of your function, this build can take hours. In the end:

   - it will wait for you to confirm a **subscription** from your email account.
   - Open your email and confirm the subscription and wait to receive an email that your Amazon FPGA Image (AFI) is available (takes about 30 minutes to an hour).

3. **Copy binaries to the runtime instance**

   - Create a github repository and clone it in your z1d.2xlarge instance.
   - Add the host, mmult.awsxclbin and xrt.ini files to the repository; commit and push

1.7 Step 6: Runtime Phase

Once you have created your executable and have your AFI it is time to run your application on the f1.2xlarge.

1.7.1 Set up a runtime instance

Follow the steps from Section 1.2, but instead of choosing a z1d.2xlarge instance, choose f1.2xlarge.

1.7.2 Run the application on the FPGA

To run your application, execute the following commands in your f1.2xlarge instance

```
source $AWS_FPGA_REPO_DIR/vitis_runtime_setup.sh
# Wait till the MPD service has initialized . Check systemctl status mpd
./host ./mmult.awsxclbin
```

You should see the following files generated when you ran:

```
profile_summary.csv
timeline_trace.csv
xclbin.run_summary
```

Note: Make sure to shut down your F1 instance! It costs 1.65 $/hr..

CHAPTER 2

Matrix Multiplier

This chapter

- illustrates the use of ***Vitis HLS***
- discusses the main parallelism pragmas

in the context of a matrix multiplication algorithm. The content of this chapter was curated by Syed Ahmed.[1]

2.1 Directory Structure

```
code/
    Makefile
    design.cfg
    xrt.ini
    common/
        Constants.h
        EventTimer.h
        EventTimer.cpp
        Utilities.cpp
        Utilities.h
    hls/
        export_hls_kernel.sh
        run_hls.tcl
        MatrixMultiplication.h
        MatrixMultiplication.cpp
        Testbench.cpp
    Host.cpp
```

2.2 The code

- There are 5 targets in the Makefile. Use make help to learn about them
- design.cfg defines several options for the ***v++ compiler***. Learn more about it here
- xrt.ini defines the options necessary for ***Vitis Analyzer***
- The common folder has header files and helper functions.

[1]University of Pennsylvania, Electrical and System Engineering. *email:* stahmed@seas.upenn.edu

- The hls/MatrixMultiplication.cpp file has the function that gets compiled to a hardware function (known as a kernel in Vitis). The Host.cpp file has the "driver" code that transfers the data to the fpga, runs the kernel, fetches back the result from the kernel and then verifies it for correctness.

2.2.1 Host.cpp: the main

The Host.cpp file has the "driver" code that transfers the data to the FPGA, runs the kernel, fetches back the result from the kernel and then verifies it for correctness.

```
#include " Utilities .h"

// ------------------------------------------------------------------------------------
// Main program
// ------------------------------------------------------------------------------------
int main(int argc, char** argv)
{
//  Initialize an event timer we'll use for monitoring the application
    EventTimer timer;
// ------------------------------------------------------------------------------------
// Step 1: Initialize the OpenCL environment
// ------------------------------------------------------------------------------------
    timer.add("OpenCL Initialization ");
    cl_int err;
    std::string binaryFile = argv[1];
    unsigned fileBufSize;
    std::vector<cl::Device> devices = get_xilinx_devices();
    devices.resize(1);
    cl::Device device = devices[0];
    cl::Context context(device, NULL, NULL, NULL, &err);
    char* fileBuf = read_binary_file(binaryFile, fileBufSize);
    cl::Program::Binaries bins{{fileBuf, fileBufSize}};
    cl::Program program(context, devices, bins, NULL, &err);
    cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE, &err);
    cl::Kernel krnl_mmult(program,"mmult", &err);

// ------------------------------------------------------------------------------------
// Step 2: Create buffers and initialize test values
// ------------------------------------------------------------------------------------
    timer.add("Allocate contiguous OpenCL buffers");
    // Create the buffers and allocate memory
    cl::Buffer in1_buf(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY, sizeof(matrix_type) * MATRIX_SIZE, NULL, &
      err);
    cl::Buffer in2_buf(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY, sizeof(matrix_type) * MATRIX_SIZE, NULL, &
      err);
    cl::Buffer out_buf_hw(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_WRITE_ONLY, sizeof(matrix_type) * MATRIX_SIZE, NULL,
      &err);

    timer.add("Set kernel arguments");
    // Map buffers to kernel arguments, thereby assigning them to specific device memory banks
    krnl_mmult.setArg(0, in1_buf);
    krnl_mmult.setArg(1, in2_buf);
    krnl_mmult.setArg(2, out_buf_hw);

    timer.add("Map buffers to userspace pointers");
    // Map host-side buffer memory to user-space pointers
    matrix_type *in1 = (matrix_type *)q.enqueueMapBuffer(in1_buf, CL_TRUE, CL_MAP_WRITE, 0, sizeof(matrix_type) * MATRIX_SIZE);
```

```
45      matrix_type *in2 = (matrix_type *)q.enqueueMapBuffer(in2_buf, CL_TRUE, CL_MAP_WRITE, 0, sizeof(matrix_type) * MATRIX_SIZE);
46      matrix_type *out_sw = Create_matrix();
47
48      timer.add("Populating buffer inputs");
49      // Initialize the vectors used in the test
50      Randomize_matrix(in1);
51      Randomize_matrix(in2);
52
53  // ------------------------------------------------------------------------------------
54  // Step 3: Run the kernel
55  // ------------------------------------------------------------------------------------
56      timer.add("Set kernel arguments");
57      // Set kernel arguments
58      krnl_mmult.setArg(0, in1_buf);
59      krnl_mmult.setArg(1, in2_buf);
60      krnl_mmult.setArg(2, out_buf_hw);
61
62      // Schedule transfer of inputs to device memory, execution of kernel, and transfer of outputs back to host memory
63      timer.add("Memory object migration enqueue host->device");
64      cl::Event event_sp;
65      q.enqueueMigrateMemObjects({in1_buf, in2_buf}, 0 /* 0 means from host*/, NULL, &event_sp);
66      clWaitForEvents(1, (const cl_event *)&event_sp);
67
68      timer.add("Launch mmult kernel");
69      q.enqueueTask(krnl_mmult, NULL, &event_sp);
70      timer.add("Wait for mmult kernel to finish running");
71      clWaitForEvents(1, (const cl_event *)&event_sp);
72
73      timer.add("Read back computation results (implicit device->host migration)");
74      matrix_type *out_hw = (matrix_type *)q.enqueueMapBuffer(out_buf_hw, CL_TRUE, CL_MAP_READ, 0, sizeof(matrix_type) *
          MATRIX_SIZE);
75      timer.finish();
76
77  // ------------------------------------------------------------------------------------
78  // Step 4: Check Results and Release Allocated Resources
79  // ------------------------------------------------------------------------------------
80      multiply_gold(in1, in2, out_sw);
81      bool match = Compare_matrices(out_sw, out_hw);
82      Destroy_matrix(out_sw);
83      delete [] fileBuf;
84      q.enqueueUnmapMemObject(in1_buf, in1);
85      q.enqueueUnmapMemObject(in2_buf, in2);
86      q.enqueueUnmapMemObject(out_buf_hw, out_hw);
87      q.finish();
88
89      std::cout << "---------------- Key execution times ----------------" << std::endl;
90      timer.print();
91
92      std::cout << "TEST " << (match ? "PASSED" : "FAILED") << std::endl;
93      return (match ? EXIT_SUCCESS : EXIT_FAILURE);
94  }
```

Listing 2.1: Host.cpp

2.2.2 MatrixMultiplication.cpp: the kernel

The MatrixMultiplication.cpp file has the function that gets compiled to a hardware function (known as a kernel in Vitis).

```
#include " MatrixMultiplication .h"

void mmult(const matrix_type Input_1[MATRIX_WIDTH * MATRIX_WIDTH],
  const matrix_type Input_2[MATRIX_WIDTH * MATRIX_WIDTH],
  matrix_type Output[MATRIX_WIDTH * MATRIX_WIDTH]) {
#pragma HLS INTERFACE m_axi port=Input_1 bundle=aximm1
#pragma HLS INTERFACE m_axi port=Input_2 bundle=aximm2
#pragma HLS INTERFACE m_axi port=Output bundle=aximm1
 matrix_type Buffer_1[MATRIX_WIDTH][MATRIX_WIDTH];
 matrix_type Buffer_2[MATRIX_WIDTH][MATRIX_WIDTH];

  Init_loop_i : for (int i = 0; i < MATRIX_WIDTH; i++)
   Init_loop_j : for (int j = 0; j < MATRIX_WIDTH; j++) {
    Buffer_1[i][j] = Input_1[i * MATRIX_WIDTH + j];
    Buffer_2[i][j] = Input_2[i * MATRIX_WIDTH + j];
   }

 Main_loop_i: for (int i = 0; i < MATRIX_WIDTH; i++)
  Main_loop_j: for (int j = 0; j < MATRIX_WIDTH; j++) {
   matrix_type Result = 0;
   Main_loop_k: for (int k = 0; k < MATRIX_WIDTH; k++) {
    Result += Buffer_1[i][k] * Buffer_2[k][j];
   }
   Output[i * MATRIX_WIDTH + j] = Result;
  }
}
```

Listing 2.2: MatrixMultiplication.cpp

2.2.3 design.cfg: Compiler Flags

Defines several options for the ***v++ compiler***. Learn more about it here

```
platform=xilinx_aws-vu9p-f1_shell-v04261818_201920_2
debug=1
 profile_kernel =data: all : all : all
save-temps=1

[ connectivity ]
nk=mmult:1:mmult_1
sp=mmult_1.Input_1:DDR[1]
sp=mmult_1.Input_2:DDR[2]
sp=mmult_1.Output:DDR[1]
```

Listing 2.3: design.cfg

2.2.4 xrt.ini: Vitis Analyzer

xrt.ini defines the options necessary for ***Vitis Analyzer***

```
[Debug]
profile=true
timeline_trace=true
data_transfer_trace=fine
stall_trace=all
```

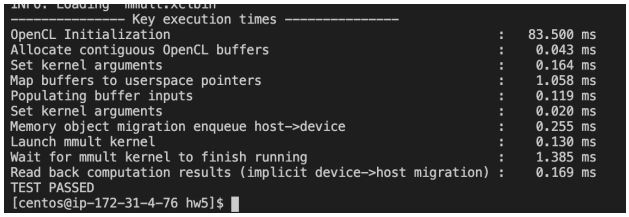


Figure 2.1: CPU Implementation

2.3 CPU implementation.

To set a benchmark for our HLS acceleration, let us first run our application on the CPU. Connect to your z1d.2xlarge and execute the following commands from the terminal to run your application on the CPU.

```
# compile
source $AWS_FPGA_REPO_DIR/vitis_setup.sh
export PLATFORM_REPO_PATHS=$(dirname $AWS_PLATFORM)
make all TARGET=sw_emu

# run
source $AWS_FPGA_REPO_DIR/vitis_runtime_setup.sh
export XCL_EMULATION_MODE=sw_emu
./host mmult.xclbin
```

The latency is **86.93**ms and will provide our benchmark.
Note: The .xclbin is a binary format optimized for FPGA. Yet, you can run as a normal app on your CPU (although you would not run it usually as it is not optimized for it).

2.4 Create a Project in Vitis

1. Launch the build instance z1d.2xlarge and ***Vitis HLS*** following the steps in Section 1.6.2
2. Create a ***Project*** in ***Vitis HLS*** as follows
   - In the drop-down click on ***File*** and select ***New Project***
   - Give a name to the Project and select the location where to store the project.
   - Specify mmult as top function.
   - Add to the source files
     - hw5/fpga/hls/MatrixMultiplication.cpp
     - hw5/fpga/hls/MatrixMultiplication.h
   - Add Testbench.cpp to the TestBench files
   - Select the xcvu9p-flgb2104-2-i in the device selection.
   - Use a 8 ns clock, and select ***Vitis Kernel Flow Target***.

- Click Finish.

Vitis HLS automatically does loop pipelining. For the purpose of this project, we will turn it off, since we are going to do it ourselves. To do so,

- Right-click on ***solution 1*** and select ***Solution Settings***.
- In the ***General*** tab, click on ***Add***.
- Select ***config_compile*** command and set ***pipeline_loops*** to 0.

2.5 C Simulation and Code Debugging

We will now follow the steps in 1.6.3 to debug the code using Testbench.cpp in ***Vitis HLS*** .
Note: The test bench generates random matrices and attempts matrix multiplication using both our mmult function (from HW) and the standard software matrix multiply function. The test-bench then compares both of the outputs and makes sure they are exactly the same..

- ***Run C simulation*** by right-clicking on the project on the ***Explorer view***
- Figure 2.2 verifies that the test passes

```
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
   Compiling ../../../../hls/Testbench.cpp in debug mode
   Compiling ../../../../hls/MatrixMultiplication.cpp in debug mode
   Generating csim.exe
TEST PASSED
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Figure 2.2: Testbench Console

2.6 Synthesis in Vitis HLS

Let us now synthesize our code using ***Vitis HLS*** . To do so, run
Solution →***Run C Synthesis*** →***Active Solution***
from the menu to synthesize your design.

2.6.1 Synthesis Report

To open the ***Synthesis Report***

- Expand the ***solution 1*** tab in the ***Explorer View***
- Browse to ***syn/report*** and open the .rpt file.

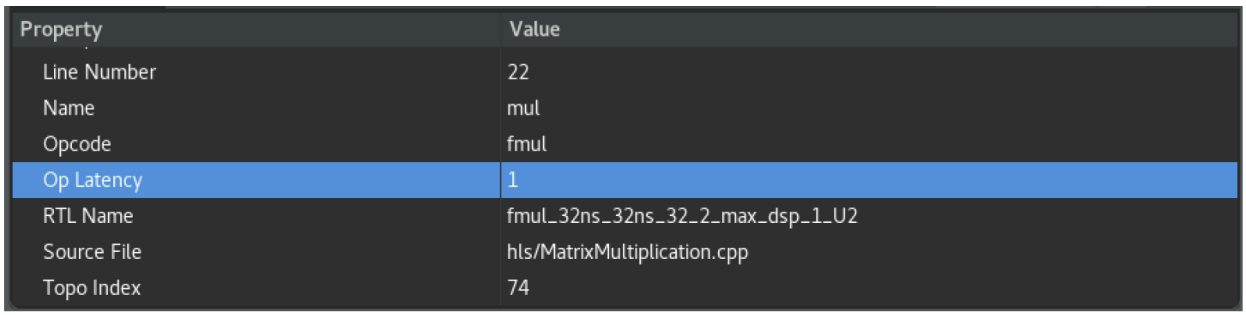


Figure 2.3: ***Scheduler View***

2.6.2 Latency

The total latency of the hardware accelerator is **16.976**, so slower than the CPU (**86.93**). The reason is that the current implementation does not use any kind of parallelism of the computation which the software baseline may have. Table 2.1 reports the resource utilization.

Resources	BRAM	DSP Units	Flip-Flops	LUTs
Usage	20	5	1793	1933

Table 2.1: Resource Utilization

2.6.3 *Scheduler View*

Use the ***Scheduler View*** under the ***Analysis Perspective*** to analyze how the computations are scheduled in time. From the ***Scheduler View*** it appears that the multiplication takes 1 cycle (Figure 2.4)

2.6.4 Data Flow

Dataflow and FSM diagram for main loop of MatrixMultiplication.cpp

2.7 HLS Kernel Optimization: Loop Unrolling

- Go back to the ***Synthesis perspective***
- Unroll the loop with label Main_loop_k 2 times using **#pragma HLS UNROLL** .

```
1 Main_loop_k: for (int k = 0; k < MATRIX_WIDTH; k++) {
2  #pragma HLS unroll factor=2
3  Result += Buffer_1[i][k] * Buffer_2[k][j];
```

Listing 2.4: MatrixMultiplication.cpp with **#pragma HLS UNROLL**

For other examples see here.

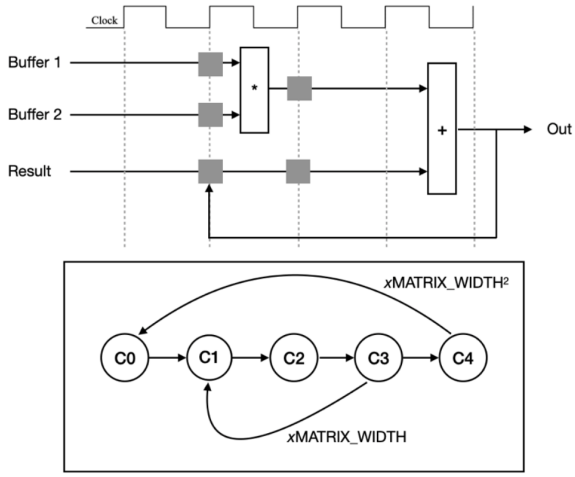


Figure 2.4: ***Scheduler View***

- Synthesize the code
- Look at the ***Scheduler View***

The unroll is able to save cycles by performing the multiplies in parallel. (The original loop had to wait for next read to perform another multiply). To understand how the unrolling work, notice that we could have performed the unrolling manually as shown here

```
1 Main_loop_k: for (int k = 0; k < MATRIX_WIDTH; k=k+2) {
2  Result += Buffer_1[i][k] * Buffer_2[k][j] + Buffer_1[i][k+1] * Buffer_2[k+1][j];
3 }
```

Listing 2.5: MatrixMultiplication.cpp

2.7.1 *Resource Profile*

Now use the ***Resource Profile*** view of the ***Analysis Perspective*** to inspect the resource usage. As we unroll more and more, the number of:

- fadd's increases but
- the number of fmul's does not.

This implies that the fmul s are shared by multiple operations!

2.7.2 Full Unroll

- Unroll the loop with label Main_loop_k completely.
- Synthesize the design again.

You may notice that the estimated clock period in the ***Synthesis Report*** is shown in red. Due to variation among ***Vitis HLS*** versions, sometimes it works and nothing is flagged.

Change the clock

Change the clock period to 20ns, and synthesize it again. The new latency is **4.062ms**.

Resources

Resources	BRAM	DSP Units	Flip-Flops	LUTs
Usage	20	14	5586	5174

Table 2.2: Resource Utilization

Note: You may have noticed that all floating-point additions are scheduled in series. This suggests that they cannot be parallelized. Floating-Point addition is non-associative; this forces us to perform them in the original serial order in order to guarantee we achieve the same result as the original, serial C code. In contrast, Integer and Fixed-Point additions are associative, giving the compiler more freedom to re-order operations and exploit parallelism.

2.8 HLS Kernel Optimization: Pipelining

Pipeline using **#pragma HLS PIPELINE**

- Remove the unroll pragma, and pipeline the Main_loop_j loop with the minimal initiation interval (II) of 1 using the **#pragma HLS PIPELINE**. (Xilinx link)
- Restore the clock period to 8ns.
- Synthesize the design again.

2.8.1 Understanding the Initiation Interval (II)

Note the initiation interval is **32** for the pipelined loop *j*. To understand this result, Figure 2.5 draws a schematic for the data path of Main_loop_j and shows how it is connected to the memories. You can find the variables that are mapped onto memories in the ***Resource Profile*** view of the ***Analysis Perspective*** . The memory for each of the Buffers is stored in one bank, in 8 BRAMS. There are only two port to read from, despite needing 64 values. Assuming a continuous flow of input data, we need to read a full row of Buffer1, meaning 64 values. The BRAM only lets us read at most 2 words per cycle, but we need 64 for loop iteration, which results in a delay (II) of 32.

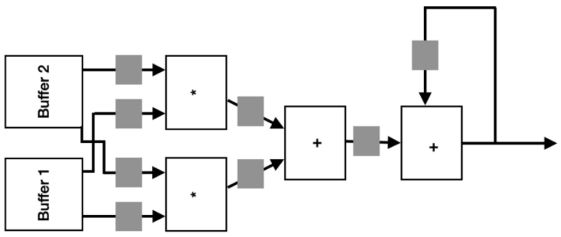


Figure 2.5: ***Scheduler View***

2.8.2 Partitioning Arrays to Improve Pipelining

To improve the II of the pipelining, we can partition Buffer_1 and Buffer_2 to achieve a better performance. To do so, we partition the input buffer into 32 pairs of columns for Buffer 1. This way, the two ports can read both the values in each BRAM at once and get all 64 values in 1 cycle. For buffer 2, we need to read all the rows of one column at once so we partition it into 32 pairs of rows. To partition the buffers we use the **#pragma HLS ARRAY_PARTITION**. For examples on how to use the pragma see here.

2.8.3 Export the Vitis Kernel

To conclude pipeline the Init_loop_j loop also with an II of 1.

```
#include " MatrixMultiplication .h"

void mmult(const matrix_type Input_1[MATRIX_WIDTH * MATRIX_WIDTH],
  const matrix_type Input_2[MATRIX_WIDTH * MATRIX_WIDTH],
  matrix_type Output[MATRIX_WIDTH * MATRIX_WIDTH]) {
#pragma HLS INTERFACE m_axi port=Input_1 bundle=aximm1
#pragma HLS INTERFACE m_axi port=Input_2 bundle=aximm2
#pragma HLS INTERFACE m_axi port=Output bundle=aximm1
 matrix_type Buffer_1[MATRIX_WIDTH][MATRIX_WIDTH];
 matrix_type Buffer_2[MATRIX_WIDTH][MATRIX_WIDTH];

#pragma HLS ARRAY_PARTITION variable=Buffer_1 complete dim=2
#pragma HLS ARRAY_PARTITION variable=Buffer_2 complete dim=1

 Init_loop_i : for (int i = 0; i < MATRIX_WIDTH; i++)
  Init_loop_j : for (int j = 0; j < MATRIX_WIDTH; j++) {
   Buffer_1[i][j] = Input_1[i * MATRIX_WIDTH + j];
   Buffer_2[i][j] = Input_2[i * MATRIX_WIDTH + j];
  }

 Main_loop_i: for (int i = 0; i < MATRIX_WIDTH; i++)
  Main_loop_j: for (int j = 0; j < MATRIX_WIDTH; j++) {
   #pragma HLS PIPELINE II=1
   matrix_type Result = 0;
   Main_loop_k: for (int k = 0; k < MATRIX_WIDTH; k++) {
    Result += Buffer_1[i][k] * Buffer_2[k][j];
   }
   Output[i * MATRIX_WIDTH + j] = Result;
  }
```

```
}
```

Listing 2.6: MatrixMultiplication.cpp

- Synthesize your design.
- **Export.** Export your synthesized design:
  - right-click on ***solution 1*** and then select ***Export RTL***.
  - Choose ***Vitis Kernel (.xo)*** as the Format.
  - Select output location to be your directory
  - Select OK.
- Save your design and quit ***Vitis HLS*** .
- Open a terminal and go to your directory. Make sure your terminal environment is initialized as follows.

```
source $AWS_FPGA_REPO_DIR/vitis_setup.sh
export PLATFORM_REPO_PATHS=$(dirname $AWS_PLATFORM)
```

2.9 Run on the FPGA

Connect to your f1.2xlarge and execute the following commands from the terminal to run your application on the FPGA.

```
source $AWS_FPGA_REPO_DIR/vitis_runtime_setup.sh
# Wait till the MPD service has initialized . Check systemctl status mpd
./host ./mmult.awsxclbin
```

You should see the following files generated when you ran:

```
profile_summary.csv
timeline_trace.csv
xclbin.run_summary
```

Listing 2.7: FPGA Run Output

Add, commit and push these files in the repository you created and then shutdown your F1 instance.
Note: Make sure to shut down your F1 instance! It costs 1.65 $/hr.

2.10 Additional Documentation

- Read this to learn about the syntax of the code in hls/MatrixMultiplication.cpp.
- Read this to learn about how the hardware function is utilized in Host.cpp.

- Read this to learn about simple memory allocation and OpenCL execution.
- Read this to learn about aligned memory allocation with OpenCL.

CHAPTER 3

Krusell Smith (1998)

This section describes the FPGA acceleration of the Krusell and Smith (1998) algorithm in Cheela et al. (2022).

3.1 Directory Structure

The directory is structured in four folders. The folder common contains code shared by FPGA and CPU acceleration. The folders cpu and fpga contain code which is specific to the two acceleration platforms. Results are stored in the folder results.

```
code/
    /common
        / libs
            ap_common.h
            ap_decl.h
            ap_fixed_base.h
            ap_fixed_ref.h
            ap_fixed_special.h
            ap_fixed.h
            ap_int_base.h
            ap_int_ref.h
            ap_int_special.h
            ap_int.h
            xcl2.cpp
            xcl2.hpp
            xcl2.mk
        /shocks
            agshock.txt
            idshock.txt
        / util
            2run_me.sh
            compare_results.py
            input_pack.py
            matlab_compare.m
            OpenMPI_install.sh
            power.sh
            save_results.sh
        app.cpp
        app.h
        cons.h
        definitions.h
        dev_options.h
        init.cpp
        init.h
        stopwatch.h
    /cpu
        sw.cpp
        sw.h
    / executables
        / fpga_afi
        / host_executables
    /fpga
        design.cfg
        hls_config.tcl
        hw.cpp
        hw.h
    / results
        /fpga
            /double
            / fixed
        /matlab
        /openmpi
            /double
        /power_usage
            /double
        /seq_cpu
            /double

Makefile
README.md
xrt.ini
```

3.2 The Code

- **Makefile.** Run the Makefile to execute the application. The Makefile has 3 main targets that allow you to choose the execution mode:
  - Serial execution on CPU: make cpu,
  - Parallel execution on CPU using Open MPI: make openmpi,
  - Specified FPGA Target and Device: make fpga.

  There are other auxiliary targets. Execute make help to learn more about them. See section 3.3 for a complete guide on how to setup and launch the application.
- **Main.** The /common/app.cpp is the main file that initializes the variables, transfers the data to the fpga, launches the hardware execution (cpu serial, cpu parallel, fpga), fetches back the result from the kernel.
- **Kernel.** The /fpga/hw.cpp contains the Vitis kernel for FPGA execution. The /cpu/sw.cpp contains the kernel executed on the CPU.
- **Results.** Results are stored in /results.
- **Header Files.** Header files and helper functions are contained in the following directory
  - /common: files shared by FPGA and CPU codes
  - /common/libs: libraries for FPGA software emulation
  - /cpu: files unique to CPU execution
  - /fpga: files unique to FPGA execution
- **Hardware Design.**
  - design.cfg, hls_config.tcl defines several options for the ***v++ compiler***. Learn more about it here.
  - xrt.ini defines the options necessary for ***Vitis Analyzer***.

3.3 Setup and Launch

This section summarizes the steps required to compile and run the application under the different acceleration modes provided in the Makefile.

3.3.1 Shared Instructions

1. Open /code/common/app.cpp and set the number of models N_MODEL you want to compute (1200 in our benchmark specification):

```
#define N_MODEL 6 // total number of models
```

2. Open /code/common/definitions.h and set the grid size:

```
#define NKGRID 100 ///< number of grid points
#define NKM_GRID 4 ///< number of grid points for the mean of capital distribution grid
```

   In the FPGA execution the user can only choose NKGRID $\in \{100, 200, 300\}$ and NKM_GRID $\in \{4, 8\}$.

3. Open /code/common/dev_options.h and set:

```
// Set only one of the below 4 to 1. For best performance, set _ACROSS_ECONOMY to 1 and rest 0
#define _BASELINE 0
#define _PIPELINE 0
#define _WITHIN_ECONOMY 0
#define _ACROSS_ECONOMY 1
```

3.3.2 Serial execution on CPU

- **Setup.** Complete the steps 1-3 in section 3.3.1

- **Setup.** To use the Jump search algorithm similar to that implemented in the FPGA, select the _CUSTOM_BINARY_SEARCH in /code/common/dev_options.h

```
// set only one for the below 3 to 1. For best CPU performance, set _CUSTOM_BINARY_SEARCH to 1 and rest 0
#define _LINEAR_SEARCH 0
#define _BINARY_SEARCH 0
#define _CUSTOM_BINARY_SEARCH 1
```

- **Compile and run.** Go to the directory /code. From there, you can use the following terminal instructions to compile and run two alternative versions of the application:

```
make cpu
./ app
```

3.3.3 Parallel execution on CPU using Open MPI

```
sh OpenMPI_install.sh
```

- Set the environment by executing the following commands in the terminal from the parent directory

```
export PATH=$PATH:$HOME/openmpi/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/openmpi/lib
```

- **Setup.** Complete the steps 1-3 in section 3.3.1

- **Compile and run.** Go to the directory /code. From there, you can use the following terminal instructions to compile and run two alternative versions of the application:

```
make openmpi
mpirun -n N ./openmpi_app // replace N with the number of CPU cores
```

3.3.4 FPGA execution

- Go to the directory /code.

- **Setup.** Complete the steps 1-3 in section 3.3.1

- The FPGA execution has two running modalities: the software emulation and the hardware image generation

  1. **Software emulation**

     - **Description.** The main goal of software emulation (sw_emu) is to ensure functional correctness of the host program and kernels. Software emulation provides a purely functional execution, without any modeling of timing delays, or latency; it does not give any indication of the accelerator performance. Hence, the sw_emu target can be built and executed on the build instance which may not have an FPGA connected to it. Click here to know more about this.
     - **Compile and Run.** From the folder /code, execute the following instruction in the terminal to compile and run the application:

```
// setup environment
source $AWS_FPGA_REPO_DIR/vitis_setup.sh
export PLATFORM_REPO_PATHS=$(dirname $AWS_PLATFORM)
// build the target
make fpga TARGET=sw_emu
// run
source $AWS_FPGA_REPO_DIR/vitis_runtime_setup.sh
export XCL_EMULATION_MODE=sw_emu
./host ./fpga/build/runOnfpga.xclbin
```

  2. **System Hardware Target**

     - **Description.** . When the build target is the hardware, v++ builds the FPGA binary for the Xilinx device by running Vivado synthesis and implementation

on the design. It is normal for this build target to take a longer period of time than generating either the software or hardware emulation targets in the Vitis IDE. Therefore, we recommend using a lower cost build instance (z1d.2xlarge) to generate the fpga target. Click here to know more about this.

- **Compile.** From the folder /code, execute the following instruction in the terminal to generate the host and the fpga target files on the build instance:

```
make clean
// setup environment
source $AWS_FPGA_REPO_DIR/vitis_setup.sh
export PLATFORM_REPO_PATHS=$(dirname $AWS_PLATFORM)
export XCL_EMULATION_MODE=hw
// build the target
make afi EMAIL=<email address>
```

- **Run.** Launch a new runtime instance (f1.2xlarge) and copy the host and the fpga targets (host, runOnfpga.awsxclbin) files from build instance to the runtime instance. Then execute the following commands to set up the vitis environment and run on the fpga device. If you like to recreate the results from the paper using the fpga binaries from the git repository, refer to section 3.8 that makes use of a bash script file.

```
git clone https :// github.com/aws/aws-fpga.git $AWS_FPGA_REPO_DIR //AWS repo
git clone https :// github.com/AleP83/KS-FPGA.git -b "dev_accel" //KS-FPGA Project
source $AWS_FPGA_REPO_DIR/vitis_setup.sh
source $AWS_FPGA_REPO_DIR/vitis_runtime_setup.sh
export PLATFORM_REPO_PATHS=$(dirname $AWS_PLATFORM)
./host ./runOnfpga.awsxclbin
```

3.4 Header Files

File: /code/common/definitions.h
Description: This is the main header files. It defines all variables and structures, it defines and initializes the model parameters, the simulation parameters, the number of states, the tolerance for convergence or the number of iterations, the file paths, among others.
Note. The file describes the main structures:

- env_t: stores model parameters, stochastic transition matrix, grids, wealth function, tax rate, wage, interest rate, and auxiliary variables for the agents optimization problem;
- input_t: stores aggregate and idiosyncratic shocks;
- var_t: stores equilibrium individual capital holdings, cross-sectional distribution, coefficients of aggregate law of motion of capital and time series of aggregate capital holdings;

- out_t: stores the computed results of cross-sectional distribution, individual capital policy functions, coefficients for good and bad states, r2 values;
- preinit_t: stores the initial values of the aggregate capital and wealth.

File: /code/common/dev_options.h
Description: This header file defines the macros used for the hardware acceleration, including: unrolling factors, finite precision of operations, and associated debugging macros.

File: /code/common/app.h
Description: This header file contains auxiliary C libraries in support of I/O operations, math operations, timing etc.

File: /code/common/cons.h
Description: This header file stores as constant the encoded aggregate and idiosyncratic shocks used in the Krusell and Smith simulation.

Files: /code/common/libs/*.h
Description: This folder contains a collection of header files which provides both integer and fixed-point arbitrary precision data types for OpenCL C++ API. The advantage of arbitrary precision data types is that they allow the C code to be updated to use variables with smaller bit-widths and then for the C simulation to be re-executed to validate that the functionality remains identical or acceptable.

Files: /code/fpga/hw.h
Description: This header file declares variables and functions in support of the FPGA acceleration kernel. In particular it declares:

- the kernel function runOnfpga;
- the structure hw_env_t which is a stripped down version excluding the of the env_t with only necessary structure members. This can be removed in the future by utilizing the definition from definitions.h;
- the regression functions;
- the linear interpolation function hw_findrange and its variations;
- auxiliary math functions.

Files: /code/cpu/sw.h
Description: This header file declares variables and functions in support of the CPU acceleration kernel, and it is comparable to hw.h for the FPGA.

Files: /code/cpu/init.h
Description: This header file declares the functions used in init.cpp

Files: /code/cpu/stopwatch.h
Description: This header file contains the class definition for the stopwatch timer which is used for measuring all latencies.

3.5 Main: app.cpp

The file /common/app.cpp is the main. The application uses the following macros to activate the alternative acceleration options: serial CPU (_SERIAL_CPU_MODE), Open MPI parallel CPU cores (_OPENMPI_MODE), FPGA acceleration (_FPGA_MODE)

```
# ifdef _OPENMPI_MODE
    #define OMPI_MODE 1      // 1 ON, 0 OFF
# elif _FPGA_MODE
    #define FPGA_MODE     1        // 1 ON, 0 OFF
# elif _SERIAL_CPU_MODE
    #define SERIAL_CPU_MODE   1      // 1 ON, 0 OFF
#endif
```

When we issue the make commands make cpu, make openmpi, make fpga, the appropriate flag gets defined using -D flag which would set only one of the above modes.

3.5.1 Overview

The rest of the section describes the FPGA acceleration associated with _FPGA_MODE.

1. Setting up the OpenCL environment
2. Allocating the buffers
3. Set up the kernels and Initialize Buffers
4. Buffer transfer to the FPGA
5. Kernel execution on FPGA
6. Buffer transfer from FPGA
7. Event synchronization
8. Post processing and release of resources

3.5.2 Setting up the OpenCL environment

The host code in the Vitis core development kit follows the OpenCL programming paradigm. To setup the runtime environment properly, the host application needs to initialize the standard OpenCL structures: target platform, devices, context, command queue, and program.
Note: The users can follow the native OpenCL C API. However, in this tutorial, we use OpenCL C++ wrapper API which is supported by XRT and many of the Vitis Examples are written using the C++ API. For more information on this C++ wrapper API, refer to this link.. However, for the CPU implementation, we only use C programming language apart from the object-oriented class in stopwatch.h file.

It is always a good coding practice to use error checking after each of the OpenCL API calls. This can help debugging and improve productivity when you are debugging the host and kernel code in the emulation flow, or during hardware execution.

```
368 cl_int err = CL_SUCCESS;
```

The second argument to the host executable stores the path to the FPGA binary file (.xclbin or .awsxclbin)

```
369 std::string binaryFile = argv[1];
```

After a Xilinx platform is found, the application needs to identify the corresponding Xilinx devices. In case of larger f1 instances, this may go up to 8 devices.

```
373 auto devices = xcl::get_xil_devices();
```

and count them.

```
374 auto device_count = devices.size();
375 int NUM_DEVICES = (int) device_count;
```

The OpenCL program is written such that it automatically scales up depending on the number of FPGA devices that are found attached to the device. Since each of the FPGA's can be individually programmed, we create a 1 dimensional vectors of context, programs, queues, binaries. In the code example, the cl::Context API is used to create a context for each of the device.

```
380 vector<cl::Context> contexts(device_count);
```

Create a program from a vector of source strings and the default context. Does not compile or link the program.

```
381 vector<cl::Program> programs(device_count);
```

Create one command queue vector for each of the FPGA devices

```
382 vector<cl::CommandQueue> queues(device_count);
```

Create a vector of kernels. Since the design makes use of three-kernel compute units per FPGA device, we create a vector of 3 kernels for each device

```
383 vector< vector<cl::Kernel> > kernels(device_count, vector<cl::Kernel>(NUM_KERNELS));
```

Attribute device name to each FPGA device

```
384 vector<std::string> device_name(device_count);
```

cl::Program creates an OpenCL program object for a context and loads the binary bits specified by the binary in each element of the vector binaries into the program object.

```
385 vector<cl::Program::Binaries> bins(device_count);
```

Upon initialization, the host application needs to identify a platform composed of one or more Xilinx devices. The command cl::Platform::get stores the list of available platforms in the vector *platform*.

```
368 vector<cl :: Platform> platform;
```

Our application assigns NUM_KERNELS kernels per device to the variable. So each FPGA-kernel compute unit is in charge of computing sequentially COMP_PER_DEVICE economies

```
390 int COMP_PER_DEVICE = ceil(N_MODEL/(NUM_DEVICES*NUM_KERNELS));
```

For example in our baseline application we execute 1200 models, N_MODEL. When we accelerate using the f1.16xlarge instance we can launch 3 kernels on each of the 8 devices in parallel. Each of the 24 FPGA-kernel compute units is in charge of computing (1200/(8*3)) = 50 economies sequentially.

3.5.3 Allocate the Buffers and Events

In the OpenCL API, data transfer between the host and the device (fpga) can be achieved by creating buffers using the command cl::Buffer API and then assigning the data pointer to it. In order to create these buffers in the stack memory, we need the size of the buffers (in bytes). This variable is used to keep track of the number of IHP iterations. Since the hardware expects a fixed size buffer, 300 elements is arbitrarily chosen for our algorithm.

```
393 const size_t hw_iter_size = 300; ///< arbitrary number chosen to represent max iterations
```

To determine the amount of bytes allocated per buffer we multiply total number of elements by the size of the data type used to represent the data

```
395 const size_t hw_preinit_size_bytes = sizeof( preinit_t );
396 const size_t hw_out_size_bytes = sizeof(out_t);
397 const size_t hw_iter_size_bytes = sizeof(int) * (hw_iter_size);
```

Initialize a 2D vector array for inputs and outputs. In this example, we are going to run the same economy several times, therefore we only need to initialize the input once which can be sent several times to different kernels on different fpga's. The output result from each of the fpga kernel is copied to different files and stored.

```
403 vector<vector< preinit_t > > hw_preinit(NUM_DEVICES, vector<preinit_t> (NUM_KERNELS));
404 vector<vector<out_t> > hw_out(NUM_DEVICES, vector<out_t> (NUM_KERNELS));
```

Initialize a 3D vector array, in which the size of the 1st dimension is the number of devices, the size of the 2nd dimension is the number of kernels (per device), and the 3rd dimension is the length of each of the variable.

For example, in the previous code, we instantiate a 2D vector structure variable of type preinit_t. The dimensions of this vector is the number of FPGA-kernel computing units NUM_DEVICES x NUM_KERNELS.

Initialize 2 dimensional OpenCL buffers for each of the variable that needs to be transferred between the host and the device.

```
vector< vector<cl::Buffer> > buffer_agshock(device_count, vector<cl::Buffer>(NUM_KERNELS));
vector< vector<cl::Buffer> > buffer_idshock(device_count, vector<cl::Buffer>(NUM_KERNELS));
vector< vector<cl::Buffer> > buffer_preinit(device_count, vector<cl::Buffer>(NUM_KERNELS));
vector< vector<cl::Buffer> > buffer_out(device_count, vector<cl::Buffer>(NUM_KERNELS));
vector< vector<cl::Buffer> > buffer_hw_iter(device_count, vector<cl::Buffer>(NUM_KERNELS));
```

Vector of events are created to coordinate the read, compute, and write operations such that each iteration is independent of each other, which allows for overlap between the data transfer and compute.

```
vector< vector< vector<cl::Event> >> memory_read_events(NUM_DEVICES, vector< vector<cl::Event> >(NUM_KERNELS, std::vector<cl::
    Event>(1)));
vector< vector< vector<cl::Event> >> task_events(NUM_DEVICES, vector< vector<cl::Event> >(NUM_KERNELS, std::vector<cl::Event>(1))
    );
vector< vector< vector<cl::Event> >> memory_write_events(NUM_DEVICES, vector< vector<cl::Event> >(NUM_KERNELS, std::vector<cl::
    Event>(1)));
```

For example, in the above code, we instantiate a 3D vector of type cl::Event for using it for read events in later sections. The dimensions of this vector are NUM_DEVICES x NUM_KERNELS x 1.

3.5.4 Set Up Kernels and Initialize Buffers

After setting up the runtime environment, such as identifying devices, creating the context, command queue, and program, the host application should identify the kernels that will execute on the device, and set up the kernel arguments.

OpenCL context, queues and device names are initialized for each of the FPGA's.

```
OCL_CHECK(err, contexts[d] = cl::Context(devices[d], props, nullptr, nullptr, &err));
OCL_CHECK(err, queues[d] = cl::CommandQueue(contexts[d], devices[d], CL_QUEUE_PROFILING_ENABLE |
    CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err));
OCL_CHECK(err, device_name[d] = devices[d].getInfo<CL_DEVICE_NAME>(&err));
```

Each of the FPGA devices needs to be loaded and programmed with a binary file.

```
fileBuf[d] = xcl::read_binary_file(binaryFile);
bins[d].push_back({fileBuf[d].data(), fileBuf[d].size()});
programs[d] = load_cl2_binary(bins[d], devices[d], contexts[d]);
```

The OpenCL API cl::Kernel should be used to access the kernels contained within the .xclbin file (the "program"). The cl::Kernel object identifies a kernel in the program loaded into the FPGA that can be run by the host application. In our paper we propose a design that can at most instantiate three kernels into the three different compute units (SLRs) of our FPGA device. Therefore, we identify each of the three kernels with the extension shown below. The kernel names are defined as in the *design.cfg* file. For example, in the below code, we have the NUM_KERNELS set to 3. So, the three kernel names that will be implemented in a single FPGA will be of the names runOnfpga_1, runOnfpga_2 and runOnfpga_3. Buffers are created for each of the FPGA devices separately as shown below.

```
for (int k = 0; k < NUM_KERNELS; k++) {
```

```
439  if( k%5 == 0 ){
440   OCL_CHECK(err, kernels[d][k] = cl::Kernel(programs[d], "runOnfpga:{runOnfpga_1}", &err));
441  }
442  if( k%5 == 1 ){
443   OCL_CHECK(err, kernels[d][k] = cl::Kernel(programs[d], "runOnfpga:{runOnfpga_2}", &err));
444  }
445  if( k%5 == 2 ){
446   OCL_CHECK(err, kernels[d][k] = cl::Kernel(programs[d], "runOnfpga:{runOnfpga_3}", &err));
447  }
448 }
```

Interactions between the host program and hardware kernels rely on creating buffers and transferring data to and from the memory in the device. This process makes use of functions like cl::Buffer and clEnqueueMigrateMemObjects. There are two methods for allocating memory buffers, and transferring data:

1. Letting XRT Allocate Buffers
2. Using Host Pointer Buffers

In the case where XRT allocates the buffer, use cl::enqueueMapBuffer to capture the buffer handle. In the second case, allocate the buffer directly with CL_MEM_USE_HOST_PTR, so you do not need to capture the handle.

On data center platforms, it is more efficient to allocate memory aligned on 4k page boundaries. On embedded platforms it is more efficient to perform contiguous memory allocation. In either case, you can let the XRT allocate host memory when creating the buffers. This is done by using the CL_MEM_ALLOC_HOST_PTR flag when creating the buffers, and then mapping the allocated memory to user-space pointers using cl::EnqueueMapBuffer . With this approach, it is not necessary to create a host space pointer aligned to the 4K boundary.

The cl::EnqueueMapBuffer API maps the specified buffer and returns a pointer created by XRT to this mapped region. Then, fill the host side pointer with your data, followed by cl::EnqueueMigrateMemObject to transfer the data to and from the device. The following code example uses this style:

```
450 std::cout << "Creating Buffers[" << d << "][" << k << "]... " << std::endl;
451 OCL_CHECK(err, buffer_agshock[d][k] = cl::Buffer(contexts[d], CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY, (cl::size_type)
        AGSHOCK_ARR_SIZE, NULL, &err));
452 OCL_CHECK(err, buffer_idshock[d][k] = cl::Buffer(contexts[d], CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY, (cl::size_type)
        IDSHOCK_ARR_SIZE, NULL, &err));
453 OCL_CHECK(err, buffer_preinit[d][k] = cl::Buffer(contexts[d], CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
        hw_preinit_size_bytes, &hw_preinit[d][k], &err));
454 OCL_CHECK(err, buffer_out[d][k] = cl::Buffer(contexts[d], CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, hw_out_size_bytes, &
        hw_out[d][k], &err));
455 OCL_CHECK(err, buffer_hw_iter[d][k] = cl::Buffer(contexts[d], CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
        hw_iter_size_bytes, hw_iter[d][k].data(), &err));
```

There are two main parts of a cl_mem object: host side pointer and device side pointer. Before the kernel starts its operation, the device side pointer is implicitly allocated on the device side

memory (for example, on a specific location inside the device global memory) and the buffer becomes a resident on the device. Using cl::EnqueueMigrateMemObjects this allocation and data transfer occur upfront, much ahead of the kernel execution. This especially helps to enable software pipelining if the host is executing the same kernel multiple times, because data transfer for the next transaction can happen when kernel is still operating on the previous data set, and thus hide the data transfer latency of successive kernel executions.
In the Vitis software platform, two types of arguments can be set for kernel objects:

1. Scalar arguments are used for small data transfer, such as constant or configuration type data. These are write-only arguments from the host application perspective, meaning they are inputs to the kernel.

2. Memory buffer arguments are used for large data transfer. The value is a pointer to a memory object created with the context associated with the program and kernel objects. These can be inputs to, or outputs from the kernel.

Kernel arguments can be set using the cl::Kernel::setArg command, as shown in the following example for setting kernel arguments for two scalar and two buffer arguments.

```
461 for (int d = 0; d < NUM_DEVICES; d++) {
462  for (int k = 0; k < NUM_KERNELS; k++) {
463   OCL_CHECK(err, err = kernels[d][k].setArg(0, buffer_agshock[d][k]));
464   OCL_CHECK(err, err = kernels[d][k].setArg(1, buffer_idshock[d][k]));
465   OCL_CHECK(err, err = kernels[d][k].setArg(2, buffer_preinit[d][k]));
466   OCL_CHECK(err, err = kernels[d][k].setArg(3, buffer_out[d][k]));
467   OCL_CHECK(err, err = kernels[d][k].setArg(4, buffer_hw_iter[d][k]));
468   std::cout << "Comleted Setting Arguments"<< std::endl;
469   agshock_ptr[d][k] = (unsigned char *) queues[d].enqueueMapBuffer(buffer_agshock[d][k], CL_TRUE, CL_MAP_WRITE, 0,
        AGSHOCK_ARR_SIZE);
470   idshock_ptr[d][k] = (unsigned char *) queues[d].enqueueMapBuffer(buffer_idshock[d][k], CL_TRUE, CL_MAP_WRITE, 0,
        IDSHOCK_ARR_SIZE);
471  }
472 }
```

We then allocate NUM_DEVICES X NUM_KERNELS number of inputs that we keep reusing to launch across these kernels COMP_PER_DEVICE number of times.

```
483 env_t env[NUM_DEVICES][NUM_KERNELS];
484 input_t in[NUM_DEVICES][NUM_KERNELS];
485 vars_t vars[NUM_DEVICES][NUM_KERNELS];
```

For each of the economy, we initialize the inputs that will be transferred to the fpga device.

```
495  init_all (&env[d][k], &in[d][k], &vars[d][k]);
496
497 for(int i=0; i<NSTATES; i++){
498  hw_preinit[d][k].kprime[i] = vars[d][k].kprime_a[i];
499 }
500
501 for(int i=0; i<NSTATES; i++){
502  hw_preinit[d][k].wealth[i] = env[d][k].wealth[i];
503 }
```

```
memcpy(agshock_ptr[d][k], in[d][k].agshock, AGSHOCK_ARR_SIZE);
memcpy(idshock_ptr[d][k], in[d][k].idshock, IDSHOCK_ARR_SIZE);
```

3.5.5 Copy Input from Host to Device

Transfer the data from host to global memory using the OpenCL API call enqueueMigrateMemObjects. The definition of this API can be found here.

```
printf("Migrating buffers to kernel\n");
if(i == 0){
OCL_CHECK(err,
 err = queues[d].enqueueMigrateMemObjects({
  buffer_agshock[d][k], buffer_idshock[d][k], buffer_preinit[d][k] },
  0 /* 0 means from host*/, nullptr, &memory_read_events[d][k][0]));
}
else{
OCL_CHECK(err,
 err = queues[d].enqueueMigrateMemObjects({
  buffer_agshock[d][k], buffer_idshock[d][k], buffer_preinit[d][k] },
 0 /* 0 means from host*/, &memory_write_events[d][k], &memory_read_events[d][k][0]));
}
```

3.5.6 Submit Kernel for Execution

Often the compute intensive task required by the host application can be defined inside a single kernel, and the kernel is executed only once to work on the entire data range. Though the kernel is executed only one time, and works on the entire range of the data, the parallelism is achieved on the FPGA inside the kernel hardware. If properly coded, the kernel is capable of achieving parallelism by various techniques such as instruction-level parallelism (loop pipeline) and function-level parallelism (dataflow).
In this tutorial, to keep things less complicated, we create a single kernel for each of the SLR compute units in the FPGA device(s). Therefore we can have a maximum of 24 independent kernels (in the f1.16xlarge) running in parallel. Each kernel has a command queue. When organizing the allocation of economies across kernels, it is advisable to break them equally among all available kernels. In this case, an out-of-order command queue can determine how the kernel tasks are processed as explained in Command Queues.

```
OCL_CHECK(err,
 err = queues[d].enqueueTask(kernels[d][k], &memory_read_events[d][k],
 &task_events[d][k][0]));
```

3.5.7 Copy the results back

After the kernel computation is completed, the host code can initiate the read back of the computed results. Depending on whether the kernel tasks are launched In-Order or Out-of-Order,

the results are read back once the cl::event indicates that the data is ready as explained in the next sections.

```
562 OCL_CHECK(err,
563  err = queues[d].enqueueMigrateMemObjects( {buffer_out[d][k], buffer_hw_iter[d][k]},
564  CL_MIGRATE_MEM_OBJECT_HOST, &task_events[d][k], &memory_write_events[d][k][0]));
```

3.5.8 Event Synchronization

All OpenCL enqueue-based API calls are asynchronous. These commands will return immediately after the command is enqueued in the command queue. To pause the host program to wait for results, or resolve any dependencies among the commands, an API call such as clFinish or clWaitForEvents can be used to block execution of the host program.

```
578 queues[d].finish();
```

Note how the commands have been used in the example above:

1. The clFinish API has been explicitly used to block the host execution until the kernel execution is finished. This is necessary otherwise the host can attempt to read back from the FPGA buffer too early and may read garbage data.

2. cl::Event

3.5.9 Printing Results

We copy the results into text files and store the values of each of the computed economy.

```
590 for (int d = 0; d < NUM_DEVICES; d++) {
591  for(int k=0; k < NUM_KERNELS; k++){
592
593   FILE * cfile;
594   char FileName[512];
595   printf("Migrating buffers from kernel\n"); //add kgrid, km grid to file names
596   sprintf(FileName, "%sfpga_nkM%d-nk%d_i%d_d%d_k%d.txt", KP_OUT_FILE, NKM_GRID, NKGRID, i, d, k);
597   cfile = fopen(FileName, "w");
598   for(int i=0; i<NSTATES; i++){
599    fprintf(cfile, "%.15lf \n", hw_out[d][k].kprime[i]);
600   }
601   fclose(cfile);
602  .
603  .
604  .
605  }
606 }
```

In addition to storing several values, we print some of the main results on the serial console for a quick check.

```
639 for(int d=0; d<NUM_DEVICES; d++){
640  for (int k = 0; k < NUM_KERNELS; k++) {
641   printf("i=%d d=%d k=%d Bad Coeff 0: %.15lf\n", i, d, k, hw_out[d][k].coeff[0]);
```

```
   printf("i=%d d=%d k=%d Bad Coeff 1: %.15lf\n", i, d, k, hw_out[d][k].coeff[1]);
   printf("i=%d d=%d k=%d Bad R2: %.15lf\n", i, d, k, hw_out[d][k].r2[0]);
   printf("i=%d d=%d k=%d Good Coeff 0: %.15lf\n", i, d, k, hw_out[d][k].coeff[2]);
   printf("i=%d d=%d k=%d Good Coeff 1: %.15lf\n", i, d, k, hw_out[d][k].coeff[3]);
   printf("i=%d d=%d k=%d Good R2: %.15lf\n\n", i, d, k, hw_out[d][k].r2[1]);
   printf("i=%d d=%d k=%d Total EGM iter: %d\n", i, d, k, total_egm_iter[d][k]);
   printf("i=%d d=%d k=%d Total Main loop iter : %d\n\n", i, d, k, hw_iter[d][k][0]);
  }
}
```

Free resources. At the end of the host code, all the allocated resources in the heap memory should be released. If the resources are not properly released, the Vitis core development kit might not able to generate a correct performance related profile and analysis report. Most of the OpenCL C++ API's have the destructor defined. Therefore we do not have to de-allocate most of them.

```
for(int d=0; d<NUM_DEVICES; d++){
 for (int k = 0; k < NUM_KERNELS; k++) {
  free_all(&in[d][k]);
 }
}
```

3.5.10 Open MPI

This subsection describes the Open MPI-specific code associated with _OPENMPI_MODE.

Begin by initializing the MPI environment.

```
mpi_enabled = MPI_Init(NULL, NULL);
```

Collect the number of processes (available cores).

```
int n_tasks;
MPI_Comm_size(MPI_COMM_WORLD, &n_tasks);
```

Collect the rank of the processes.

```
int id_task;
MPI_Comm_rank(MPI_COMM_WORLD, &id_task);
```

Block all processes in the communicator MPI_COMM_WORLD until all processes have called it.

```
MPI_Barrier(MPI_COMM_WORLD);
```

Specify the range of models for each process to compute. We assign the economies equally across processes.

```
// Range of tasks per processor.
int i_min_task_id, i_max_task_id;

// Define the Block to be assigned to each task
parameters_range_pertask(0,N_MODEL-1,n_tasks,id_task,&i_min_task_id,&i_max_task_id);
```

Next, the processes compute their assigned economies in parallel.

```
for(int i = i_min_task_id; i <= i_max_task_id; i++) {
.
.
.
 env_t env;
 input_t in;
 vars_t vars;
 out_t out;
 int hw_iter[500];

  init_all (&env, &in, &vars);
.
.
.
 runOncpu(&env, &vars, in.agshock, in.idshock, &out, hw_iter);
}
```

Save the results of each of the computed model.

```
FILE * cfile ;
char FileName[512];
printf("Migrating buffers from kernel\n");
sprintf (FileName, "%scpu-core-%d_of_%d_nKM%d-nk%d.txt", OPENMPI_KP_OUT_FILE, id_task, n_tasks, NKM_GRID, NKGRID);
cfile = fopen(FileName, "w");
for(int i=0; i<NSTATES; i++){
  fprintf ( cfile , "%.15lf \n", out.kprime[i]);
}
fclose ( cfile );
.
.
.
```

Print the final values of R2 score and the Coefficient values for each model in the terminal.

```
printf("Total EGM iter: %d\n", total_egm_iter);
printf("Total Main loop iter : %d\n", hw_iter[0]);
printf("Bad Coeff 0: %.15lf\n", out.coeff[0]);
printf("Bad Coeff 1: %.15lf\n", out.coeff[1]);
printf("Good Coeff 0: %.15lf\n", out.coeff[2]);
printf("Good Coeff 1: %.15lf\n", out.coeff[3]);
printf("Bad R2: %.15lf\n", out.r2[0]);
printf("Good R2: %.15lf\n\n", out.r2[1]);
```

After the processes have completed their assigned economies, terminate the MPI environment and exit.

```
MPI_Finalize();
```

3.6 Kernel: hw.cpp

The file /common/hw.cpp contains the hardware design of the kernel.

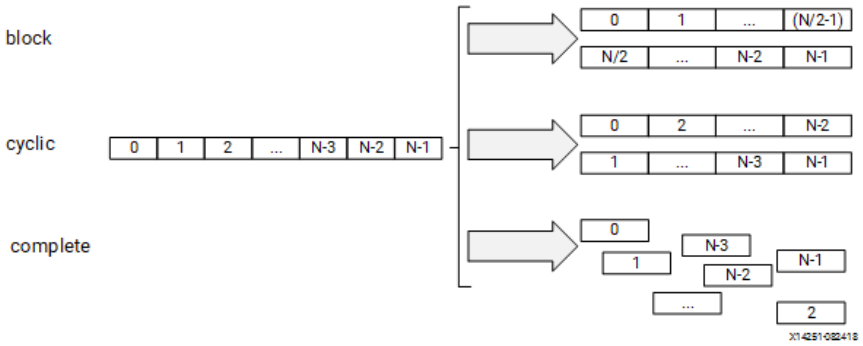
3.6.1 Common HLS Optimization Pragmas

This section describes the main #PRAGMAs used to design the hardware acceleration of our algorithm.

3.6.1.1 #pragma HLS ARRAY_PARTITION

Each memory block (BRAM, URAM) consists of a limited number of memory ports to read or write from the memory. For example a BRAM block usually consist of 2 ports. When data is stored in a BRAM in a contiguous manner, we can only read a maximum of 2 elements in the same clock cycle for a dual port BRAM block. This may create a bottleneck when we want to access more than two elements simultaneously. To overcome this challenge, Xilinx suggest to store the data across multiple blocks of memory instead of storing it in a contiguous manner. By partitioning an array across N memory blocks, we utilize N number of memory blocks each of which can have up to 2 memory ports thereby enabling a maximum of $2N$ memory accesses in a single cycle. We can instruct the Vitis complier to split the elements of an array and then map them to smaller arrays using **#pragma HLS ARRAY_PARTITION**. There are 3 main ways to partition an array as described in Figure 3.1. *Source:* Xilinx link.

Figure 3.1: Partitioning Arrays: Three types



Note: Array partition using the three types: *(i)* Block; *(ii)* Cyclic; and *(iii)* Complete. The image is taken from Xilinx UG1393.

3.6.1.2 #pragma HLS UNROLL

In order to make use of the fpga resources, the designer can spatially unroll loops to create multiple independent operations rather than a single collection of operations. The **#pragma**

Figure 3.2: Partitioning Dimensions of an Arrays

Note: This figure shows how the same array can be partitioned across different axis (0, 1, 3) resulting in 240, 10 and 4 separate arrays respectively. The image is taken from Xilinx UG1393.

HLS UNROLL transforms loops by creating in hardware multiple copies of a loop body such that they can all occur in parallel. By default the unrolling is set to complete, however, the user can set a specific number using the object factor. *Source:* Xilinx link.

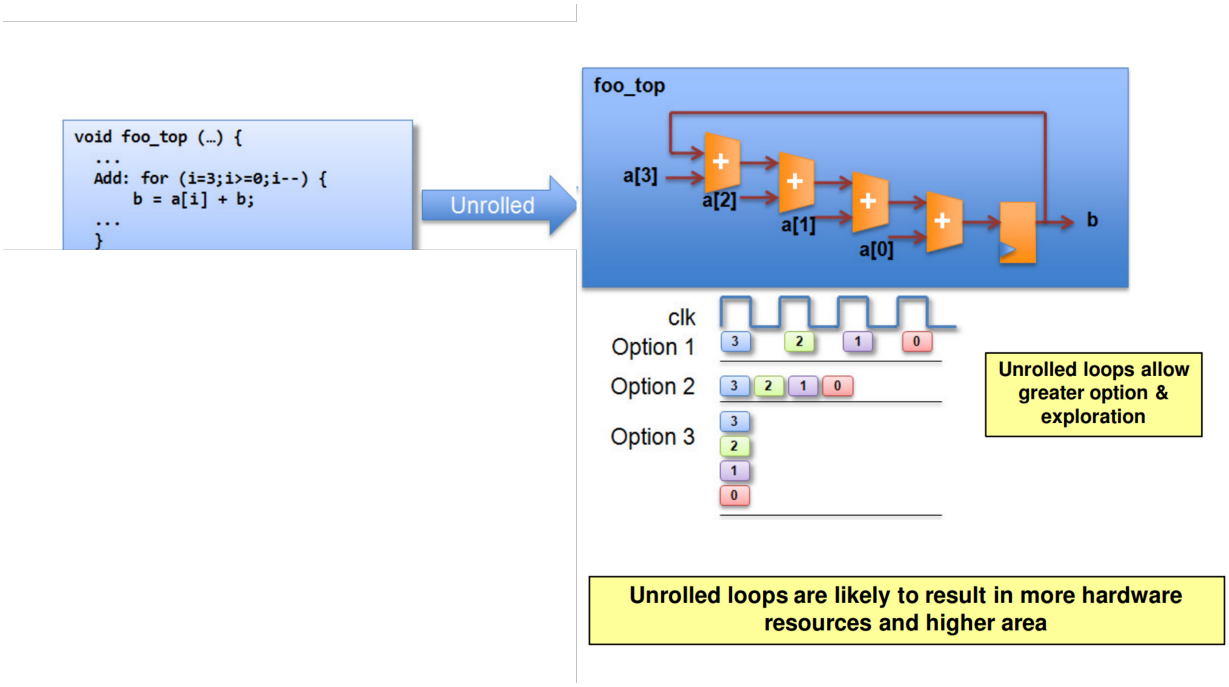
3.6.1.3 #pragma HLS PIPELINE

A pipelined function (or loop) processes new inputs every N clock cycles, where N is the Initiation Interval (II) of the loop or a function. By default, the II for the **#pragma HLS PIPELINE** is set to 1. However, a user can specify the required value using the II option for the pragma.

The Figure 3.4 shows a case where placing the pipeline pragma at different loop locations results in 3 different unrolling of the inner loops along with the increased hardware resources and memory accesses. The user needs to make a conscious choice about the placement of the pipeline pragma. If the data accessed inside the loop is unable to process in a single cycle, the II of the loop would change from 1 to N, where N is the number of clock cycles after which the data of the next loop iteration can be accessed.

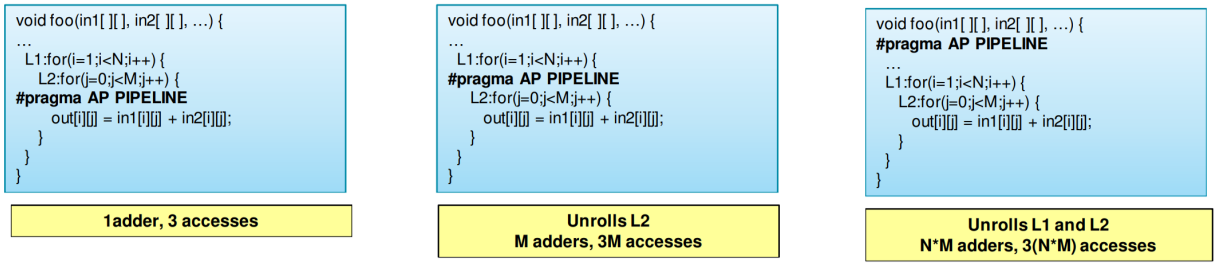
The loop pipelining can be prevented when there are loop carry dependency or if the inner loops consist of variable loop bounds. It can also be limited if the required data is unable to be accessed in a single clock cycle. In that case, the designer can solve the problem by using the **#pragma HLS ARRAY_PARTITION** discussed in the previous section. *Source:* Xilinx link.

Figure 3.3: Impact of various factors of loop Unrolling



Note: This figure shows how the unrolling by different factors decreasing the overall latency of the loop while increasing the hardware resources.

Figure 3.4: Impact of Pipeline pragma at different levels

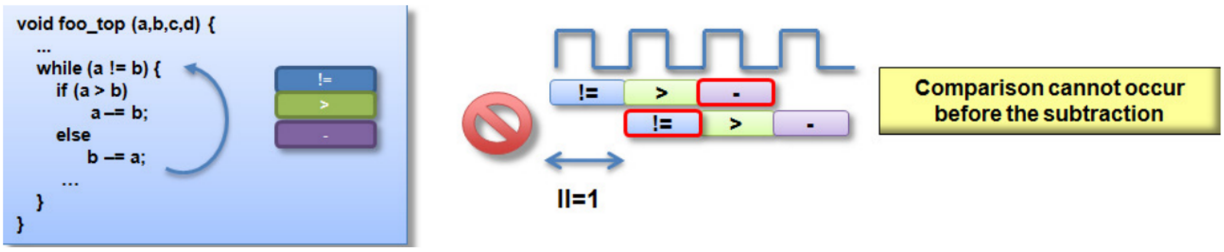


3.6.1.4 #pragma HLS LOOP_TRIPCOUNT

This pragma does not perform any optimization and has no impact on the results of the synthesis. However, for a undefined loop bounds, this can be applied to manually specify the expected number of iterations.

When we are in the process of generating the output binary file, after the first step of C synthesis, the Vitis HLS provides us with the synthesis reports. This reports consists of several important information regarding the latencies for all the major loops. Wherever, the loop has a data dependent variable, the tool will be unable to estimate the latencies. Hence, the above

Figure 3.5: Data dependency preventing II=1



Note: This figure shows how the data dependency in a loop prevents the pipeline in achieving an II=1.

pragma instructs the tool to calculate the latencies for the given number of iterations. This information helps us to keep track of the results of the optimizations that we perform.

In this example, the loop_1 is specified to have a minimum, average and maximum trip counts of 12, 14 and 16 respectively. Without this pragma, the tool cannot determine the loop latency.

```
void foo (num_samples, ...) {
  int i;
  ...
  loop_1: for(i=0;i< num_samples;i++){
   #pragma HLS loop_tripcount min=12 max=16
    ...
     result = a + b;
  }
}
```

Source: Xilinx link.

3.6.1.5 #pragma HLS INLINE

Removes a function as a separate entity in the hierarchy. This reduces the overhead for the function call and can allow the function to be optimized into the caller. When you inline, you will have a separate set of hardware for each place where the function is inlined. *Source:* Xilinx link.

3.6.2 Overview

The kernel is organized in:

- a parent function that manages data transfers from and to the host and executes the fixed point algorithm: runOnfpga;
- four functions that executes the KS algorithm: hw_sim_alm, hw_sim_ihp, hw_sim_ast, sim_alm_coeff;

- auxiliary functions that support or accelerate the algorithm: hw_pow, hw_exp, hw_log, hw_sqrt, hw_fabs, hw_init_env, hw_rail_values, hw_fxd_rail_values,, hw_findrange, hw_findrange_n4, hw_findrange_n100, regression, RSqauredCalc.

3.6.3 Parent Kernel Function: runOnfpga

runOnfpga is the parent kernel function which:

1. manages the FPGA interface
2. manages the memory allocation
3. executed the nested fixed point algorithm
4. send the results back to the host

3.6.3.1 Memory Management

The kernel function name of the complete synthesised logic is runOnfpga. The code snippet below lists the parameter that are passed to the kernel from host. Most of the parameters here refers to the pointers to the off-chip DRAM memory which resides in the external DDR memory in the data center. The memory latency to an off-chip memory access is extremely large and cost a lot of energy compared to on-chip memory access. Therefore, the first step is to allocate on-chip memories for all the data-variables which are accessed multiple times and then initialize the on-chip memories with the data from the off-chip memory. We discuss some of the memory allocations of different variables by making use of the different on-chip memory resources such as BRAM, URAM and Registers.

```
99  void runOnfpga(
100   const unsigned char *hw_agshock,
101   const unsigned char *hw_idshock,
102    preinit_t  * preinit ,
103   out_t  * results ,
104   int  *hw_iter)
```

The structure variables which are declared outside the main function are treated as static variables and the data is retained across multiple inferences. It is recommended to limit the usage of global variables.

```
10  /* Static on-PL memories */
11  static hw_env_t st_env;
```

Throughout the program, we make use of the structure variable st_env which is derived of the structure type hw_env_t consisting of the calibration parameters and some of the temporary data variables as defined in the fine hw.h.

We can create local variables whose scope is limited to the function that they are allocated in. In our program, we allocate the following variables that are common across different functions. By default, the Vitis compiler would try to choose a memory type depending on the data access patterns. For example, if the program only reads a value from a pre-initialized data variable, the tool may choose to synthesize that variable using single ported BRAM. This consumes less hardware resources as compared to the dual port BRAM resources. Most of the default memory allocations work well with the designs. However, the user is free to change the default memory types as per their requirement using the **#pragma HLS BIND_STORAGE**.

We optimize the memory resource for storing the Individual Shocks which is declared here as idshock. The program uses a #ifdef condition which checks for PACK_IDS. If this is enabled in the dev_options.h file, we instruct the tool to allocate NEW_IDSHOCK_SIZE number of rows of width 72bits. Usually, the x86 machines are limited to using a `double` to store large numbers. However, we can choose to use a custom fixed point number that can be larger than 64 bits. More details about this is explained below. In the case where the PACK_IDS is disabled, the tool is free to choose a suitable memory, which is observed to be BRAM18.

```
106   unsigned char agshock[AGSHOCK_ARR_SIZE];
107 #if PACK_IDS
108   ap_uint<72> idshock[NEW_IDSHOCK_SIZE] = {0};
109 #else
110   unsigned char idshock[IDSHOCK_ARR_SIZE] = {0};
111 #endif
112
113   real st_kcross[N_AGENTS];
114   real st_kprimes[NUM_KPRIMES][NSTATES];
115   real kmts[SIM_STEPS];
116   real r2[NSTATES_AG];
117   real kmprime[NSTATES_AG * NKM_GRID];
118   real coeff[NCOEFF] = {0, 1, 0, 1};
119   real metric_coeff = 1000; // some large number
```

```
121 #if PACK_IDS
122 #pragma HLS bind_storage variable = idshock type = RAM_1P impl = URAM
123 #endif
```

In our program, we optimize the memory usage for some of the data variables. the variable is specified using the keyword variable, the type of memory is selected using type and the implementation using impl. Xilinx provides a complete list of possible combinations that can be found here. By choosing these options, the tool will now use URAM memory of type single port RAM to implement the idshock variable. We choose a single port RAM as we are going to write the data to this variable only once and read the data from here only once in a single clock cycle. The data read for idshock is further explained in the section (hw_sim_ast). Note that for all the arrays, the size needs to be specified for it to be synthesised.

```
130 #pragma HLS array_partition variable = st_env.k complete dim = 1
131 #pragma HLS array_partition variable = st_env.km complete dim = 1
```

The memory containing the individual capital and the mean of the aggregate capital distribution needs to be accessed multiple times in the same clock cycle. Therefore, these two variables are

partitioned completely.

After allocating the on-chip memories for the different data variables, we now need to initialize the local on-chip memories with the data from the off-chip memory before we start using them in Eq. ??. To perform this step efficeintly, Xilinx recommends to use *Burst Transfer*. Burst transfer refers to reading or writing chunks of data to or from the global memory in a single request. This is the most effective optimization to reads/writes data to external memory which is usually the DDR. The below code copies the aggregate shocks using the pointer hw_agshock pointing to a location in the external memory to the data variable agshock which resides on the on-chip memory.

```
136  for (int i = 0; i < AGSHOCK_ARR_SIZE; i++)
137  {
138   agshock[i] = hw_agshock[i];
139  }
```

Similarly, now we want to burst transfer the id shocks. In the code snippet below, we have two different options provided to demonstrate the improvement by using URAM. When the PACK_IDS is enabled, we instruct the compiler to copy 8 elements of the input data elements which is of 8 bits size into a single element of on-chip unsigned fixed point data type that is of size 64 bits. By doing so, we can access 64 bits of idshocks by accessing a single element of the idshocks. Otherwise, the compiler would use the default BRAM memory to store the idshock where we can access a maximum of 8 different idshock s for each access to an element in the array.

```
141 #if PACK_IDS
142  // use URAM to store the idshocks
143  // 8 idshocks are packed into 1 byte-> (1,100 * 10,000 / 8) = 1,375,000 bytes
144  // copy to data variable of size 64 bits. Hence, 8 input bytes are copied to one element
145 main_2: // loop over each of the 1,100 time step. (10,000 / 8) = 1250
146  for (int i = 0, j = 0; i < IDSHOCK_ARR_SIZE; i = i + 1250)
147  {
148  main_2_2: // for each time step, copy 8 bytes into a single element of size 64 bits
149   for (int k = 0; k < 1250; j++)
150   {
151    // handle edge case where last 2 bytes are remaining since 1,250 is not devisible by 8
152    if (k == 1248)
153    {
154     idshock[j] = (hw_idshock[i + k + 1] << 8) | (hw_idshock[i + k]);
155     k = k + 2;
156    }
157    else
158    {
159     idshock[j] = (((ap_uint<72>)hw_idshock[i + k + 7] << 56) |
160           ((ap_uint<72>)hw_idshock[i + k + 6] << 48) |
161           ((ap_uint<72>)hw_idshock[i + k + 5] << 40) |
162           ((ap_uint<72>)hw_idshock[i + k + 4] << 32) |
163           ((ap_uint<72>)hw_idshock[i + k + 3] << 24) |
164           ((ap_uint<72>)hw_idshock[i + k + 2] << 16) |
165           ((ap_uint<72>)hw_idshock[i + k + 1] << 8) |
166           ((ap_uint<72>)hw_idshock[i + k + 0]));
167     k = k + 8;
168    }
169   }
```

```
  }

#else
 // use BRAM to store the idshocks
 main_2:
  for (int i = 0; i < IDSHOCK_ARR_SIZE; i++)
  {
   idshock[i] = hw_idshock[i];
  }
#endif
```

Further, we created a function call to initialize the remaining data variables.

```
hw_top_init(st_kprimes, st_kcross);
```

The capital function for t=0 - kprimes, and kcross are burst copied from the global memory.

```
void hw_top_init(
 real st_kprimes[NUM_KPRIMES][NSTATES], real st_kcross[N_AGENTS]
)
{
 init_1 :
  for (int j = 0; j < NSTATES; ++j)
  {
   real val = kp_in[j];
   for (int k = 0; k < NUM_KPRIMES; ++k)
   {
    st_kprimes[k][j] = val;
   }
  }

 init_2 :
  for (int j = 0; j < N_AGENTS; ++j)
  {
   st_kcross[j] = env__kss;
  }
```

Note that the initialization from here on-wards can be moved to the host side and the initialized data can be sent to the device. This is left for future experiments. To minimize some of the one-time initialized data variables, we pre-compute the result and store it locally.

```
  st_env.irate_factor[0] = 0.356400000000000;
  st_env.irate_factor[1] = 0.363600000000000;

  st_env.wage_factor[0] = 0.633600000000000;
  st_env.wage_factor[1] = 0.646400000000000;

  st_env.cons2_factor[0] = 0.150000000000000;
  st_env.cons2_factor[1] = 1.094444444444445;
  st_env.cons2_factor[2] = 0.150000000000000;
  st_env.cons2_factor[3] = 1.104861111111111;

  hw_init_env();

  return;
```

After all the burst reads, we initialize the global env structure variable using the following code.

```
900 void hw_init_env()
901 {
902 #pragma HLS inline
903   st_env.alpha = env__alpha;      // 0.36 (Output capital share)
904   st_env.beta = env__beta;      // 0.99 (Quarterly subjective discount factor)
905   st_env.delta = env__delta;      // 0.025 (Quarterly depreciation rate)
906   st_env.mu = env__mu;      // 0.15 (Unemployment benefits in terms of wages)
907   st_env.l_bar = env__l_bar;      //
908   st_env.delta_a = env__delta_a; // 0.01
909
910   st_env.l_bar_inv = env__l_bar_inv; // 0.9 (Time endowment)?
911   st_env.gamma_inv = env__gamma_inv;
912   st_env.gamma_neg = env__gamma_neg;
913   st_env.gamma_neg_inv = env__gamma_neg_inv;
914
915   st_env.epsilon_u = env__epsilon_u;
916   st_env.epsilon_e = env__epsilon_e;
917
918   st_env.ur[0] = env__ur_0;
919   st_env.er[0] = (1 - st_env.ur[0]);
920   st_env.ur[1] = env__ur_1;
921   st_env.er[1] = (1 - st_env.ur[1]);
922
923   st_env.er_inv[0] = 1 / st_env.er[0];
924   st_env.er_inv[1] = 1 / st_env.er[1];
925
926   // st_env.kss = hw_pow((1./st_env.beta-(1.-st_env.delta))/st_env.alpha,1./(st_env.alpha-1));
927   st_env.kss = env__kss;
928
929   // transition
930   st_env.P[0] = 0.525;
931   st_env.P[1] = 0.35;
932   st_env.P[2] = 0.03125;
933   st_env.P[3] = 0.09375;
934   st_env.P[4] = 0.038889;
935   st_env.P[5] = 0.836111;
936   st_env.P[6] = 0.002083;
937   st_env.P[7] = 0.122917;
938   st_env.P[8] = 0.09375;
939   st_env.P[9] = 0.03125;
940   st_env.P[10] = 0.291667;
941   st_env.P[11] = 0.583333;
942   st_env.P[12] = 0.009115;
943   st_env.P[13] = 0.115885;
944   st_env.P[14] = 0.024306;
945   st_env.P[15] = 0.850694;
946
947   // parmshocks
948   st_env.epsilon[0] = st_env.epsilon_u;
949   st_env.epsilon[1] = st_env.epsilon_e;
950 #if AST_UNROLL
951   for (int k = 0; k < NUM_KCROSS; ++k)
952   {
953 #pragma HLS pipeline off
954     st_env.epsilon2[k][0] = 0;
955     st_env.epsilon2[k][1] = 1;
956   }
957 #else
958   st_env.epsilon2[0] = 0;
959   st_env.epsilon2[1] = 1;
```

```
960 #endif
961
962   st_env.ag[0] = 1 - st_env.delta_a;
963   st_env.ag[1] = 1 + st_env.delta_a;
964   st_env.ag2[0] = 0;
965   st_env.ag2[1] = 1;
966   return;
967 }
```

3.6.3.2 Fixed Point Algorithm

The following data variables are used to keep track of the total number of iterations required for the convergence of the ALM coefficients hw_main_iter and individual household **IHP** problem curr_ihp_iter, and an array to store the number of **IHP** iterations at every ALM coefficient loop iteration. These variables (among others) are used in the validation phase to debug and compare the results with the MATLAB code.

```
155   int hw_main_iter = 0;   // total number of ihp calls
156   int curr_ihp_iter = 0;   // number of ihp iterations in each ihp call
157   int hw_ihp_iter[300] = {0};  // local mem array to store the number of ihp iterations
```

After completing all the memory initialization, the runOnfpga function launches the nested fixed point algorithm:

- hw_sim_alm: updates the expectations about the first moment of the capital distribution, m';
- hw_sim_ihp: solves the individual household (**IHP**) problem
- hw_sim_ast: performs the stochastic simulation
- sim_alm_coeff: updates the estimates of the Aggregate Law of Motion coefficients.

```
163   while (metric_coeff > TOLL_COEFF)
164   {
165    hw_main_iter++;
166    hw_sim_alm(kmprime, coeff); // step 1
167
168     curr_ihp_iter = 0;
169    hw_sim_ihp(st_kprimes, kmprime, curr_ihp_iter); // step 2
170    hw_ihp_iter[hw_main_iter] = curr_ihp_iter; // start from 1st element of hw_ihp_iter
171
172     real kcross_l[N_AGENTS];
173     kc_t kcross_mean = 0;
174
175    ast_kcross:
176     for (int is = 0; is < N_AGENTS; is++)
177     {
178       kcross_l[is] = st_kcross[is];
179       kcross_mean += (kc_t) st_kcross[is];
180     }
181
182     hw_sim_ast(kmts, st_kprimes, kcross_l, agshock, idshock, kcross_mean); // step3
```

```

    sim_alm_coeff(kmts, coeff, &metric_coeff, r2, agshock); // step 4

    if (metric_coeff > TOLL_COEFF * 100)
    {
     // Replace the old with new capital distribution
     for (int j = 0; j < N_AGENTS; j++)
     {
      st_kcross[j] = kcross_1[j];
     }
    }

#if PRINT_LOOP_CNT
    iter_main++;
    printf("main loop iter = %d\n", iter_main);
#endif
   }
```

3.6.4 Aggregate Law of Motion: hw_sim_alm

Description:This function computes the next period expected aggregate physical capital.
Acceleration: None / Instruction Level Parallelism.

```
void hw_sim_alm(real *kmprime, real * coeff)
{
 small_idx_t cidx = 0;
 real c0, c1;
 small_idx_t kidx = 0;

alm_1:
 for (int ia = 0; ia < NSTATES_AG; ++ia)
 {
  c0 = coeff[cidx];
  c1 = coeff[cidx + 1];
  cidx += REGRESSORS;
 alm_2:
  for (int ikm = 0; ikm < NKM_GRID; ++ikm)
  {
#pragma HLS unroll factor = 1
   // add pipeline registers to split the computation into multiple stages
   real t_log = hw_log(km_grid[ikm]);
   real t_mul = c1 * t_log;
   real t_add = c0 + t_mul;
   real val = hw_exp(t_add);       // hw_exp(c0 + c1 * st_env.log_env_km[ikm])
   hw_rail_values(&val, KM_MAX, KM_MIN); // eq 15
   kmprime[kidx++] = val;
  }
 }
 return;
}
```

The function computes the next period expected aggregate physical capital. We note that the important step (in the code snippet above) is the computation of the logarithm of the coefficient and updating the kmprime. The exponential operator consumes a large number of resources to implement and this function only takes a small fraction of the total compute time. Therefore, we

instruct the vitis compiler to only create 1 copy of the inner loop using the unroll pragma. Further, we increase the number of pipeline registers in this inner loop by storing the intermediate results in separate registers thereby improving the setup and hold timing.

3.6.5 Individual Household Problem: hw_sim_ihp

Description:This function solves the individual agent problem

$$k' = \left[\mu(1-\epsilon) + (1-\tau)\bar{l}\epsilon\right] w + (1-\delta+r)k - \left\{\lambda + \beta\mathbb{E}\left[\frac{1-\delta+r'}{\left((\mu(1-\epsilon') + (1-\tau')\bar{l}\epsilon'\right)w' + (1-\delta+r')k' - k'(k')\right)^{\gamma}}\right]\right\}^{-1/\gamma} \tag{3.1}$$

at every state, $k, \epsilon, m, A \in \mathbf{K} \times \{0,1\}_\epsilon \times \mathbf{M} \times \mathbf{A}$. More compactly,

$$\hat{k'}_{i+1} = \Phi k'_i$$

Acceleration: Array Partition, Pipeline, Unroll.

3.6.5.1 Memory Management.

```
253 #if (NUM_KPRIMES == 8 && _WITHIN_ECONOMY)
254 #pragma HLS array_partition variable = st_env.P complete
255 #pragma HLS array_partition variable = st_kprimes complete dim = 1
256 #pragma HLS bind_storage variable = st_kprimes type = RAM_1WNR impl = BRAM
257 #else
258 #pragma HLS array_partition variable = st_kprimes complete dim = 1
259 #pragma HLS bind_storage variable = st_kprimes type = RAM_2P impl = BRAM
260 #endif
261
262 #if AST_UNROLL
263 #pragma HLS array_partition variable = st_env.epsilon2 complete dim = 1
264 #endif
```

We will later see that the st_env.P is accessed only once in the inner most loop. Therefore, it needs to have at least 4 read ports when the outerloop, ihp_2 is pipelined. Since the size of this structure member consist of only 16 elements, we partition it completely. However, it is sufficient to have a cyclic partition with a factor of 4.

```
253   /** Lookup tables */
254   // substitute for IXV call
255   static const small_idx_t li_2d_aux_idx_base[4] = {
256    0,
257    NKGRID,
258    NKM_GRID * NSTATES_ID * NKGRID,
259    (NKGRID + NKM_GRID * NSTATES_ID * NKGRID)};
260
261 #pragma HLS array_partition variable = li_2d_aux_idx_base complete
```

```
262
263   // Local kprime/new copies
264   real kprime_new[NSTATES];
265   real metric = 1;
266 #if PRINT_LOOP_CNT
267   unsigned int iter_cnt = 0;
268 #endif
```

We then proceed with initializing a lookup table to calculate the indexes of nested loops and unroll it completely. Further, we allocate memory for kprime_new and do not perform any memory optimization as it only accessed once for every iteration of ihp_2 and therefore a single memory port is sufficient.

3.6.5.2 Individual Household Problem (IHP) Loop.

This loop determines the number of iterations hw_egm_iter = i required to estimate the individual capital-holdings policy functions, $k'(k, \epsilon, m, A) : \mathbf{K} \times \{0,1\}_\epsilon \times \mathbf{M} \times \mathbf{A} \to \mathbb{R}_+$. **endogenous convergence** : This modality is for determining the policy functions. TOLL_K stores the convergence tolerance ε_k, while metric is initialized to 1 and it is iteratively updated.

```
253 // Convergence loop: 4 x NSTATES interp over kprime[]
254 ihp_1:
255   while (metric > (real)TOLL_K) // eq 14
256   {
257     hw_ihp_iter++;
```

Since the ihp_1 loop iterations are data dependent, the vitis compiler will not be able to estimate the loop latencies as discussed in the section??. Hence, we use the **#pragma HLS LOOP_TRIPCOUNT** to inform the compiler about the maximum number of iterations.

```
287 #pragma HLS loop_tripcount min = 1 avg = 200 max = 2000
```

Initializations. Before executing the **IAP Iteration Step** (in the next section):

```
289   spread_t spread_scalar = VERY_SMALL_SCALAR;
290
291   // Reset index values for [1600] loop
292   pidx_t p_idx_outer = 0b0100; // 4
293   small_idx_t hundreds_cnt = NKGRID;
294   small_idx_t kp_iter_cnt = (NSTATES_ID * NKGRID);
295   small_idx_t kidx = 0;
```

- we initialize spread_scalar to a small number. spread_scalar stores the maximum absolute difference (across the state space) between the guessed policy function and the policy function implied by Equation (3.1), $\max_{(k,\epsilon,m,A)\in\mathbf{K}\times\{0,1\}_\epsilon\times\mathbf{M}\times\mathbf{A}} |k'_{i+1} - k'_i|$. This variable is updated in the next loop.
- we reset the indexes

At each iteration the loop iterates over the states

$$\rho\left(k'_{i+1}, k'_i\right) = \max_{(k,\epsilon,m,A)\in\mathbf{K}\times\{0,1\}_\epsilon\times\mathbf{M}\times\mathbf{A}} |k'_{i+1} - k'_i| < \varepsilon_k = 1e(-8)$$

3.6.5.3 IHP Iteration Step.

This loop over the state space $(k, \epsilon, m, A) \in \mathbf{K} \times \{0,1\}_\epsilon \times \mathbf{M} \times \mathbf{A}$

```
296  ihp_2:
297    for (small_idx_t is = 0; is < NSTATES; ++is)
298    {
299  #if SMALL_PL
300  #pragma HLS unroll factor = 1
301  #else
302  #pragma HLS pipeline
303  #endif
```

takes as given:

- tomorrow's predicted aggregate capital kmp = m', as computed in hw_sim_alm
- the guessed individual capital-holding policy function, kp = $k'_i(k, \epsilon, m, A)$

and uses Equation (3.1) to update the guess

$$\hat{k'}_{i+1} = \Phi k'_i \qquad (a) \quad \text{Solve (3.1)}$$

$$k'_{i+1} = \eta_k \hat{k'}_{i+1} + (1 - \eta_k) k'_i \qquad (b) \quad \text{Update Guess}$$

To do so, the IAP Iteration Step performs the following operations:

1. **Index Handling (Technical).**

```
305    pidx_t p_idx_inner = 0; // IIDP x IAP
306    real kmp, temp_base;
307    emu_s_t emu_s = 0.;
308    real kp = st_kprimes[0][ is ];
309
310    // Index handling
311    if (++kp_iter_cnt >= NSTATES_ID * NKGRID)
312    {
313     kp_iter_cnt = 0;
314     kmp = kmprime[kidx++];
315     temp_base = kmp * (real)env__l_bar_inv;
316    }
317    if (++hundreds_cnt >= NKGRID)
318    {
319     hundreds_cnt = 0;
320     //(changes between 0 and 4 for every 100 iterations uptil is = 800,
321     // and changes between 8 and 12 for every 100 iterations uptil is = 1600)
322     p_idx_outer ^= (pidx_t)0b0100; // (XOR at every bit) 0100 ^ 0100 = 0000 -> 0 (explicit conversion to short) decimal
        value
323    }
324    if (is == (NKM_GRID * NSTATES_ID * NKGRID)) // 800 ia
325     p_idx_outer |= (pidx_t)0b1000;    // (OR at every bit) 0000 | 1000 = 1000 -> 8 (explicit conversion to short) decimal
        value
326
```

2. **Compute the conditional expectation** emu_s:

$$\mathbb{E}\left[\frac{1-\delta+r'}{\left(\left(\mu(1-\epsilon')+(1-\tau')\bar{l}\epsilon'\right)w'+(1-\delta+r')k_i'-k_i'(k_i')\right)^{\gamma}}\right]$$

To compute the conditional expectation the algorithm iterates over next period aggregate and idiosyncratic shocks' states:

(a) For each tomorrow's aggregate-shock state, A', compute wages, interest rate and labor-income taxes:

```
328    ihp_3:
329    for (int iap = 0; iap < NSTATES_AG; ++iap)
330    {
331 #if SMALL_PL
332 #pragma HLS unroll factor = 1
333 #endif
334      real temp = temp_base * st_env.er_inv[iap];
335      real irate = st_env.irate_factor[iap] * hw_pow(temp, env__alpha_c);
336      real imrt = env__delta_c + irate ;
337      real wage = st_env.wage_factor[iap] * hw_pow(temp, env__alpha);
338      small_idx_t kpb = iap << 2;
```

(b) For each tomorrow's aggregate shock, A', and idiosyncratic-shock, ϵ', state

```
339    ihp_4:
340     for (int iidp = 0; iidp < NSTATES_ID; ++iidp)
341     {
```

(c) Use a linear interpolation scheme to determine tomorrow's individual capital-holding choice fp = $k'' = k_i'(k_i') = \left(k'(k,\epsilon,m,A),\epsilon',m',A'\right)$

```
345        small_idx_t i1_min = hw_findrange((fixed_t)kmp, fxd_km_grid, NKM_GRID);
346        small_idx_t i1_max = i1_min + 1;
347        real i1_min_val = km_grid[i1_min];
348        real i1_max_val = km_grid[i1_max];
349        small_idx_t i2_min = hw_findrange((fixed_t)kp, fxd_k_grid, NKGRID);
350        small_idx_t i2_max = i2_min + 1;
351        real i2_min_val = k_grid[i2_min];
352        real i2_max_val = k_grid[i2_max];
353        small_idx_t idx_base = li_2d_aux_idx_base[p_idx_inner];
354        small_idx_t i1_min_base = idx_base + (NSTATES_ID * NKGRID * i1_min);
355        small_idx_t i1_max_base = idx_base + (NSTATES_ID * NKGRID * i1_max);
356        real tz_num = (kmp - i1_min_val);
357        real tz_den = (i1_max_val - i1_min_val);
358        real tz = tz_num / tz_den;
359        real tw_num = (kp - i2_min_val);
360        real tw_den = (i2_max_val - i2_min_val);
361        real tw = tw_num / tw_den;
362        real sub_tz = (1.0 - tz);
363        real sub_tw = (1.0 - tw);
364        real sub_tz_sub_tw = sub_tz * sub_tw;
365        real tz_tw = tz * tw;
366        real sub_tz_tw = sub_tz * tw;
367        real tz_sub_tw = tz * sub_tw;
368 #if (NUM_KPRIMES == 1)
```

```
        real fp_1 = st_kprimes[0][i1_min_base + i2_min] * sub_tz_sub_tw +
          st_kprimes[0][i1_min_base + i2_max] * sub_tz_tw;
        real fp_2 = st_kprimes[0][i1_max_base + i2_min] * tz_sub_tw +
          st_kprimes[0][i1_max_base + i2_max] * tz_tw;
# elif (NUM_KPRIMES == 4)
        real fp_1 = st_kprimes[0][i1_min_base + i2_min] * sub_tz_sub_tw +
          st_kprimes[1][i1_min_base + i2_max] * sub_tz_tw;
        real fp_2 = st_kprimes[2][i1_max_base + i2_min] * tz_sub_tw +
          st_kprimes[3][i1_max_base + i2_max] * tz_tw;
# elif (NUM_KPRIMES == 8)
        real fp_1 = st_kprimes[kpb + 0][i1_min_base + i2_min] * sub_tz_sub_tw +
          st_kprimes[kpb + 1][i1_min_base + i2_max] * sub_tz_tw;
        real fp_2 = st_kprimes[kpb + 2][i1_max_base + i2_min] * tz_sub_tw +
          st_kprimes[kpb + 3][i1_max_base + i2_max] * tz_tw;
#endif
        real fp = fp_1 + fp_2;
```

Note: The algorithm implements a fixed-size, parallel search algorithm as discussed in the paper..

(d) Given tomorrow's individual capital-holding choice fp and tomorrow's wealth, compute tomorrow's consumption cons2 = $(\mu(1-\epsilon') + (1-\tau')\bar{l}\epsilon')\, w' + (1-\delta+r')k_i' - k_i'(k_i')$ and the marginal utility of tomorrow's consumption mu2

```
        real cons2_1 = imrt * kp;
        real cons2_2 = wage * st_env.cons2_factor[p_idx_inner];
        real cons2 = (cons2_1 + cons2_2) - fp;
        if (cons2 < 0) // eq 11
         cons2 = CONS2_MIN;
        real mu2 = hw_pow(cons2, env__gamma_neg);
        real emu_s_1 = imrt * mu2;
        emu_s += (emu_s_t)(st_env.P[p_idx_outer + p_idx_inner] * emu_s_1);
        ++p_idx_inner;
```

(e) Compute $\mathbb{E}\left[\dfrac{1-\delta+r'}{\left((\mu(1-\epsilon') + (1-\tau')\bar{l}\epsilon')\, w' + (1-\delta+r')k_i' - k_i'(k_i')\right)^{\gamma}}\right]$

```
    emu_s_t emu_s = 0.;
```

```
   ihp_3:
    for (int iap = 0; iap < NSTATES_AG; ++iap)
    {
```

```
    ihp_4:
     for (int iidp = 0; iidp < NSTATES_ID; ++iidp)
     {
```

3. Compute the RHS of Equation (3.1) and store it in new_kp = $\hat{k'}_{i+1}$

$$\hat{k'}_{i+1} = \left[\mu(1-\epsilon) + (1-\tau)\bar{l}\epsilon\right]\; w + (1-\delta+r)k - \left\{\lambda + \beta\mathbb{E}\left[\frac{1-\delta+r'}{\left((\mu(1-\epsilon') + (1-\tau')\bar{l}\epsilon')\, w' + (1-\delta+r')k_i' - k_i'(k_i')\right)^{\gamma}}\right]\right\}^{-1/\gamma}$$

```
397 real new_kp = init_wealth[is] - hw_pow(env__beta * (real)emu_s, env__gamma_neg_inv); // eq 10
```

Note: Notice, following Maliar et. al (2010) we set the multipler λ to 0..

3.6.5.4 Closing the IAP Loop.

1. **Update the guess.**

```
406 ihp_5:
407   for (small_idx_t is = 0; is < NSTATES; ++is)
408   {
409 #pragma HLS pipeline
410     real updated_kp = UPDATE_K * kprime_new[is] + UPDATE_K_C * st_kprimes[0][is]; // eq 13
411     for (small_idx_t k = 0; k < NUM_KPRIMES; ++k)
412       st_kprimes[k][is] = updated_kp;
413   }
```

$$k'_{i+1} = \eta_k \hat{k'}_{i+1} + (1 - \eta_k) k'_i$$

Note: To reduce the memory ports access bottleneck we created NUM_KPRIMES copies of the policy function guess k'_i, which all need to be initialized with the new guess..

2. **Update the** metric $= \rho\left(k'_{i+1}, k'_i\right)$.

$$\rho\left(k'_{i+1}, k'_i\right) = \max_{(k,\epsilon,m,A)\in \mathbf{K}\times\{0,1\}_\epsilon\times\mathbf{M}\times\mathbf{A}} |k'_{i+1} - k'_i| < \varepsilon_k = 1e(-8)$$

```
406 ihp_5:
407   for (small_idx_t is = 0; is < NSTATES; ++is)
408   {
409 #pragma HLS pipeline
410     real updated_kp = UPDATE_K * kprime_new[is] + UPDATE_K_C * st_kprimes[0][is]; // eq 13
411     for (small_idx_t k = 0; k < NUM_KPRIMES; ++k)
412       st_kprimes[k][is] = updated_kp;
413   }
```

The metric is updated and before the start of next iteration, it is checked if lower (equal) to TOLL_K (ε_k), the loop exits.

```
415   // ~ Update metric
416   metric = (real) spread_scalar ;
```

3.6.6 Stochastic Simulation: hw_sim_ast

Description: This function simulates the time series of the cross-sectional average (per-capita) stock of capital $\{m_t\}_{t=1}^{1100}$ which is then used by the aggregate law of motion function sim_alm_coeff to estimate the expected evolution of the capital distribution.
Acceleration: Array Partition, Pipeline, Unroll.

3.6.6.1 Memory Management.

We first determine the number of reads for each of the arrays and perform the array_partition as per the requirement. For example, the array st_kcross is a double precision 1D array with 10,000 (N_AGENTS) elements. As we will see in later section of the code, for every iteration of the inner most loop, there is a read and write operation requiring at least 2 ports for a single pipeline. In the baseline model, we require 8 parallel pipelines which translates to requiring 16 IO ports. In the below code, where the PARTITION_KCROSS is set to 8, we partition the array in a cyclic manner with a factor of 8 resulting us with 16 ports. Since we explicitly specify the memory type to be RAM_S2P, we get 8 read ports and 8 write ports all of which can be accessed in the same clock cycle.

```
439 #if (PARTITION_KCROSS == 1)
440 #pragma HLS array_partition variable = st_kcross type = cyclic factor = 1
441 #elif (PARTITION_KCROSS == 4)
442 #pragma HLS array_partition variable = st_kcross type = cyclic factor = 4
443 #elif (PARTITION_KCROSS == 8)
444 #pragma HLS array_partition variable = st_kcross type = cyclic factor = 8
445 #endif
446
447 #pragma HLS bind_storage variable = st_kcross type = RAM_S2P impl = BRAM
```

The interpolated values are read 4 times in a random manner for each of the pipeline.In the baseline model, we have 8 parallel pipelines. Therefore, we allocate the memory for two copies each of which have NUM_KCROSS number of copies. In total, we create $NUM\_KCROSS * 2 = 16$ copies of the interpolated values. When we partition then using a dual port RAM across the first dimension, we get 32 read ports which can then satisfy our requirement of 4 reads over 8 pipelines.

```
448 #if AST_UNROLL
449   real kprime_interp0[NUM_KCROSS][NSTATES_ID * NKGRID];
450   real kprime_interp1[NUM_KCROSS][NSTATES_ID * NKGRID];
451 #pragma HLS array_partition variable = kprime_interp0 complete dim = 1
452 #pragma HLS array_partition variable = kprime_interp1 complete dim = 1
453 #pragma HLS array_partition variable = st_env.epsilon2 complete dim = 1
454 #else
455   real kprime_interp0[NSTATES_ID * NKGRID];
456   real kprime_interp1[NSTATES_ID * NKGRID];
457 #endif
```

As discussed in section ??, we provide an option to optimize the memory usage for storing the IDSHOCKS when the PACK_IDS is enabled. In the below code, we set the count to start from the number of IDSHOCKS stored in each of the array elements.

```
459 #if PACK_IDS
460   small_idx_t idshock_cnt = 64;
461   ap_uint<72> temp_ids = idshock[0];
462 #else
463   small_idx_t idshock_cnt = 8;
464 #endif
```

The temporary variables are declared to keep track of the shocks.

```
int idshock_idx = 0;
idx_t agshock_idx = 0;
shock_t curr_ids;
shock_t curr_ags;
small_idx_t ags_phase = AGS_PACK_FACTOR;
```

The initial value of the moment of the capital distribution is passed in to this function. For every next iteration, this value is calculated at the end of its previous iteration. This value is then checked to be within the bounds of 30,50.

```
real curr_kmts = (real)kcross_mean * N_AGENTS_INV;
hw_rail_values(&curr_kmts, KM_MAX, KM_MIN);
```

3.6.6.2 Loop.

For each time period $t \in \{0, \ldots, 1099\}$[1]

```
ast_1:
 for (int t = 0; t < SIM_STEPS; ++t)
 {
```

1. **Interpolation.** For each individual $j = 1, \ldots, 10,000$, use an interpolation scheme to determine the next period individual capital holdings, given the period t idiosyncratic $(k_{t,j}, \epsilon_{t,j})$ and aggregate (m_t, A_t) state.

```
  kmts[t] = curr_kmts;

  // Read next packed agshock value when needed
  if (++ags_phase >= AGS_PACK_FACTOR)
  {
   curr_ags = agshock[agshock_idx++];
   ags_phase = 0;
  }

  bool p0 = (curr_ags & 0b1) ? 0b1 : 0b0;

  curr_ags >>= 1;
  real p1 = kmts[t];
  small_idx_t i2_min = hw_findrange((fixed_t)p1, fxd_km_grid, NKM_GRID);
  small_idx_t i2_max = i2_min + 1;
  real i2_min_val = km_grid[i2_min];
  real i2_max_val = km_grid[i2_max];
  real ty = (p1 - i2_min_val) / (i2_max_val - i2_min_val);
  real P = (p0 == 1) ? 0 : (1.0 - ty);
  real Q = (p0 == 1) ? 0 : (ty);
  real R = (p0 == 1) ? (1.0 - ty) : 0;
  real S = (p0 == 1) ? (ty) : 0;
  small_idx_t i1_min_base = 0;     // L4D_D3 * i1.min(0)
  small_idx_t i1_max_base = L4D_D3; // L4D_D3 * i1.max
  small_idx_t i2_min_base = L4D_D2 * i2_min;
```

[1] Notice the recasting of the time indexes from $\{1, \ldots, 1100\}$ to $\{0, \ldots, 1099\}$ in order to accommodate the array indexing convention in C.

```
    small_idx_t  i2_max_base = L4D_D2 * i2_max;
    small_idx_t  i12_min_min = i1_min_base + i2_min_base;
    small_idx_t  i12_min_max = i1_min_base + i2_max_base;
    small_idx_t  i12_max_min = i1_max_base + i2_min_base;
    small_idx_t  i12_max_max = i1_max_base + i2_max_base;
    small_idx_t  kpi_idx  =  0;

```

Begin by initializing values of the aggregate shock A_t and the average of individual capital holdings m_t for interpolation Initialize values for interpolation given each idiosyncratic shock to the employment status, $\epsilon_{t,j} \in \{0, 1\}_\epsilon$

```
      small_idx_t  i3_min_base =  0;      // L4D_D1 * i3.min (0)
      small_idx_t  i3_max_base = L4D_D1; // L4D_D1 * i3.max (1)
      real  tz  = st_env.epsilon[ iid ];

```

Initialize values for interpolation given each point in the individual capital holdings grid, $k_{t,j} \in \mathbf{K}$

```
    ast_3:
     for  (int  ik  =  0;  ik  <  NKGRID; ++ik)
     {
#pragma HLS pipeline
      int  i4_min  =  ik;
      real  p  =  (1.0  -  tz);
      real  r  =  tz;
     }

```

Use linear interpolation to determine the next period individual capital holdings fp = $k'(k, \epsilon, m, A)$

```
      small_idx_t  kp_idx_0 = i4_min + i3_min_base + i12_min_min;
      small_idx_t  kp_idx_2 = i4_min + i3_max_base + i12_min_min;
      small_idx_t  kp_idx_4 = i4_min + i3_min_base + i12_min_max;
      small_idx_t  kp_idx_6 = i4_min + i3_max_base + i12_min_max;
      small_idx_t  kp_idx_8 = i4_min + i3_min_base + i12_max_min;
      small_idx_t  kp_idx_10 = i4_min + i3_max_base + i12_max_min;
      small_idx_t  kp_idx_12 = i4_min + i3_min_base + i12_max_max;
      small_idx_t  kp_idx_14 = i4_min + i3_max_base + i12_max_max;
//  ** LI3D
# if  ((NUM_KPRIMES == 4) || (NUM_KPRIMES == 8))
      real  fp  = st_kprimes[0][kp_idx_0] * P * p +
          st_kprimes[0][kp_idx_2] * P * r +
          st_kprimes[1][kp_idx_4] * Q * p +
          st_kprimes[1][kp_idx_6] * Q * r +
          st_kprimes[2][kp_idx_8] * R * p +
          st_kprimes[2][kp_idx_10] * R * r +
          st_kprimes[3][kp_idx_12] * S * p +
          st_kprimes[3][kp_idx_14] * S * r;
# elif  (NUM_KPRIMES == 1)
      real  fp  = st_kprimes[0][kp_idx_0] * P * p +
          st_kprimes[0][kp_idx_2] * P * r +
          st_kprimes[0][kp_idx_4] * Q * p +
          st_kprimes[0][kp_idx_6] * Q * r +
          st_kprimes[0][kp_idx_8] * R * p +
          st_kprimes[0][kp_idx_10] * R * r +
          st_kprimes[0][kp_idx_12] * S * p +
```

```
            st_kprimes[0][kp_idx_14] * S * r;
#endif

```

Store the solution given each point in the capital holdings grid as kprime_interp0 and kprime_interp1

```
#if AST_UNROLL
      for (int k = 0; k < NUM_KCROSS; ++k)
      {
       kprime_interp0[k][kpi_idx] = fp;
       kprime_interp1[k][kpi_idx] = fp;
      }
#else
      kprime_interp0[kpi_idx] = fp;
      kprime_interp1[kpi_idx] = fp;
#endif
      ++kpi_idx;

```

Initialise the aggregate capital to 0

```
    //aggregate capital initialized to 0
    kc_t agg_capital = 0;

```

Iterate over N_AGENTS using 8 parallel pipelines. The **#pragma HLS PIPELINE** unrolls the inner loop completely creating 8 pipelines. The IDSHOCKS when the PACK_IDS is enabled, consists of 64 shocks in each element, hence a new element is fetched from the array only once for every 8 iterations of ast_4

```
    small_idx_t kidx = 0;
// Loop 1.3: AST agents interp over kprime_interp
// Unroll factor dictated by inner loop over k
#if PACK_IDS
    idshock_cnt = 8;
#endif
  ast_4:
    for (int j = 0; j < (N_AGENTS / IDS_PACK_FACTOR) / IDS_AGG_X; j++)
    {
#pragma HLS pipeline
#if PACK_IDS
      if (idshock_cnt >= 8)
      {
       idshock_cnt = 0;
       temp_ids = idshock[idshock_idx];
       idshock_idx++;
      }
      curr_ids = temp_ids & 0xFF;
      idshock_cnt++;
      temp_ids >>= 8;
#else
      curr_ids = idshock[idshock_idx++];
#endif

```

Initialize values for interpolation over kprime_interp0 and kprime_interp1 from above

```
603     real p1b = st_kcross[kidx];
604     small_idx_t i2b_min = hw_findrange((fixed_t) st_kcross[kidx], fxd_k_grid, NKGRID);
605     small_idx_t i2b_max = i2b_min + 1;
606     real i2b_min_val = k_grid[i2b_min];
607     real i2b_max_val = k_grid[i2b_max];
608     bool p0b = (curr_ids & 0b1) ? 0b1 : 0b0;
609     curr_ids >>= 1;
610     small_idx_t i1b_min_base = 0;       // NKGRID * i1b_min(0)
611     small_idx_t i1b_max_base = NKGRID; // NKGRID * i1b_max(1)
612     real bw = (p1b - i2b_min_val) / (i2b_max_val - i2b_min_val);
613     real sub_bw = (1.0 - bw);
614     real bz_bw = (p0b == 1) ? bw : 0;
615     real sub_bz_sub_bw = (p0b == 1) ? 0 : sub_bw;
616     real bz_sub_bw = (p0b == 1) ? sub_bw : 0;
617     real bw_sub_bz = (p0b == 1) ? 0 : bw;
618
```

Use linear interpolation to compute and store next period aggregate capital given each agent's individual savings decision

```
618     real fbp_1 = (kprime_interp0[k][i1b_min_base + i2b_min] * sub_bz_sub_bw) +
619         (kprime_interp0[k][i1b_min_base + i2b_max] * bw_sub_bz);
620     real fbp_2 = (kprime_interp1[k][i1b_max_base + i2b_min] * bz_sub_bw) +
621         (kprime_interp1[k][i1b_max_base + i2b_max] * bz_bw);
622     kc_t fpb = kc_t(fbp_1 + fbp_2);
623     hw_fxd_rail_values(&fpb, KMAX, KMIN);
624     st_kcross[kidx] = (real)fpb;
625     agg_capital += fpb;
626     kidx++;
627
```

2. **Accumulation.** For each time period t, compute m_t, the cross-sectional average of individual capital holdings

$$m_t = \frac{1}{\mathcal{J}} \sum_{j=1}^{\mathcal{J}} k_{j,t}.$$

```
688 curr_kmts = ((real) agg_capital * N_AGENTS_INV);
689
```

For values that fall outside the capital grid, $\mathbf{M} = [m_{\min}, m_{\max}]$, set as the range value

```
689 hw_rail_values(&curr_kmts, KM_MAX, KM_MIN);
690
```

3.6.7 Aggregate Law of Motion: sim_alm_coeff

Description: This function estimates the i-iteration ALM coefficients $\hat{b}^i(a) = (\hat{b}_1^i(a), \hat{b}_2^i(a))$ and updates them.

Acceleration: Array Partitioning, Pipelining..

1. **House keeping.** Store old coefficient $b_l^i(a)$, $a \in \{a_b, a_g\}$, Prevent automatic array partitioning of coeff array

```
 real coeff[NCOEFF] = {0.};
sim_alm_1:
 for (small_idx_t i = 0; i < NCOEFF; i++)
 {
#pragma HLS pipeline off
  coeff[i] = coeff_updated[i];
 }
```

Initializations

```
  small_idx_t agshock_idx = 0;
  small_idx_t ags_phase = AGS_PACK_FACTOR;
  shock_t curr_ags = 0;
  shock_t curr_shock_val = 0;
  real coeff_new[NCOEFF] = {0.};
  real x_good_v[1000] = {0.};
  real y_good_v[1000] = {0.};
  real x_bad_v[1000] = {0.};
  real y_bad_v[1000] = {0.};

  int ibad = 0;
  int igood = 0;
  agshock_idx = 0;
  ags_phase = AGS_PACK_FACTOR;
```

```
sim_alm_2:
  for (int t = 0; t < SIM_STEPS; t++)
  {
#pragma HLS pipeline off
#pragma HLS unroll factor = 1
    // Read new value when needed
    if (++ags_phase >= AGS_PACK_FACTOR)
    {
     curr_ags = agshock[agshock_idx++];
     ags_phase = 0;
    }
    curr_shock_val = curr_ags & 0b1; // take the least significant bit from the byte
    curr_ags >>= 1;          // right shift by 1
  // Discard first 100
  sim_alm_3:
    if (t < NDISCARD || t > SIM_STEPS - 2)
     continue;
```

Organize the time series. The best linear approximation of the conditional expectation of next period log-aggregate capital depends on the aggregate shock. So after discarding the first 100 observations the code split the simulated data $\{m_t\}_{t=100}^{1,100}$ into two time series. To estimate the coefficients:

2.
   - when the aggregate shock is $a_t = a_b$, $\{b_1(a_t), b_2(a_b)\}$

$$E[\ln m_{t+1} | a_t = a_b] = b_1(a_b) + b_2(a_b) \ln m_t, \quad t = 100, \ldots, 1100$$

```
sim_alm_4:
 if (curr_shock_val == 0)
 {
  y_bad_v[ibad] = hw_log(kmts[t + 1]);
  x_bad_v[ibad] = hw_log(kmts[t]);
  ibad++;
 }
```

it collects

$$\{\ln m_{l+1}, \ln m_l\}_{l \in \{t \in \{100,\ldots,1100\}: a_t = a_b\}}$$

- when the aggregate shock is $a_t = a_g$, $\{b_1(a_t), b_2(a_g)\}$

$$E[\ln m_{t+1} | a_t = a_g] = b_1(a_g) + b_2(a_g) \ln m_t, \quad t = 100, \ldots, 1100$$

```
 else
 {
  y_good_v[igood] = hw_log(kmts[t + 1]);
  x_good_v[igood] = hw_log(kmts[t]);
  igood++;
 }
```

it collects

$$\{\ln m_{l+1}, \ln m_l\}_{l \in \{t \in \{100,\ldots,1100\}: a_t = a_g\}}$$

```
real badcoeff[2] = {0.}; // initialize to prevent garbage values
real goodcoeff[2] = {0.};
regression(badcoeff, x_bad_v, y_bad_v, ibad);
regression(goodcoeff, x_good_v, y_good_v, igood);
real rbad = RSquaredCalc(badcoeff, x_bad_v, y_bad_v, ibad);
real rgood = RSquaredCalc(goodcoeff, x_good_v, y_good_v, igood);
coeff_new[0] = badcoeff[0]; // bb
coeff_new[1] = badcoeff[1];
coeff_new[2] = goodcoeff[0];
coeff_new[3] = goodcoeff[1];
R2[0] = rbad;
R2[1] = rgood;
```

Estimate the coefficients. For each aggregate state $a_t \in \{a_b, a_g\}$ it uses the matrixfunction to run the OLS regressions

$$\ln m_{l+1} = b_1(a_l) + b_2(a_l) \ln m_l + \epsilon_{l+1}, \qquad l \in \{t \in \{100, \ldots, 1100\} : a_l = a_b\}$$
$$\ln m_{l+1} = b_1(a_l) + b_2(a_l) \ln m_l + \epsilon_{l+1}, \qquad l \in \{t \in \{100, \ldots, 1100\} : a_l = a_g\}$$

and estimate the coefficients governing the transition from a bad state badcoeff = $\{b_1(a_t), b_2(a_b)\}$. and good state goodcoeff = $\{b_1(a_t), b_2(a_g)\}$.

```
// Update metric for convergence test (eq 17)
real norm = 0.;
sim_alm_5:
```

```
768   for (int ib = 0; ib < NCOEFF; ++ib)
769   {
770 #pragma HLS pipeline off
771     norm += (coeff_new[ib] - coeff[ib]) * (coeff_new[ib] - coeff[ib]);
772   }
773   *metric = hw_sqrt(norm);
```

Compute the Euclidean Norm.

$$\sqrt{\sum_{l\in\{1,2\},a\in\{a_b,a_g\}} (b_l^{i+1}(a) - b_l^i(a))^2} < \varepsilon_b = 1e(-8)$$

```
773   // Update ALM coefficients vector
774   sim_alm_6:
775   for (int ib = 0; ib < NCOEFF; ++ib)
776   {
777 #pragma HLS pipeline off
778     coeff_updated[ib] = coeff_new[ib] * UPDATE_B + coeff[ib] * (1. - UPDATE_B); //
779   }
```

Update the Coefficients.

$$b_l^{i+1}(a) = \eta_b \hat{b}_l^i(a) + (1-\eta_b) b_l^i(a), \qquad l \in \{1,2\}, \quad a \in \{a_b, a_g\}$$

3.6.7.1 Regression Coefficients: Regression

Description:This function computes the estimated coefficients. Since the mathematical operators such as pow, div consumes significant amount of hardware resources, and the execution time of this function is considerably small, we decided to turn-off the automatic pipeline to make use of the hardware resources for more time-consuming tasks. We instruct the compiler using **#pragma HLS UNROLL** to unroll the loop by a factor of 1 and use **#pragma HLS LOOP_TRIPCOUNT** to specify the number of loop iterations.
Acceleration: No acceleration.

```
783 void regression(real *resultmatrix, real *x, real *y, int ndim)
784 {
785   real twobytwo[4] = {0, 0, 0, 0};
786 RG_1:
787   for (int i = 0; i < ndim; i++)
788   {
789 #pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
790 #pragma HLS unroll factor = 1
791 #pragma HLS pipeline off
792     twobytwo[0] += 1;
793     twobytwo[1] += x[i];
794     twobytwo[2] += x[i];
795     twobytwo[3] += hw_pow(x[i], 2);
796   }
797   // get inverse
```

```
798   real a = twobytwo[0]; // switching indices and multiplying by determinant
799   real b = twobytwo[1];
800   real c = twobytwo[2];
801   real d = twobytwo[3];
802   real det = (a * d - b * c);
803
804   real inv_det = (1.0 / det);
805   real inv_d = inv_det * d;
806   real inv_b = inv_det * (b) * -1;
807   real inv_c = inv_det * (c) * -1;
808   real inv_a = inv_det * a;
809   real acc1 = resultmatrix [0];
810   real acc2 = resultmatrix [1];
811  // multiply by transpose of matrix and y
812  RG_2:
813   for (int i = 0; i < ndim; i++)
814   {
815 #pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
816 #pragma HLS unroll factor = 1
817 #pragma HLS pipeline off
818     real acc_t1 = inv_b * x[i];
819     real acc_t2 = inv_d + acc_t1;
820     acc1 += acc_t2 * y[i];
821   }
822    resultmatrix [0] = acc1;
823  RG_3:
824   for (int i = 0; i < ndim; i++)
825   {
826 #pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
827 #pragma HLS unroll factor = 1
828 #pragma HLS pipeline off
829     real acc2_t1 = inv_a * x[i];
830     real acc2_t2 = inv_c + acc2_t1;
831     acc2 += acc2_t2 * y[i];
832   }
833    resultmatrix [1] = acc2;
834   return;
835  }
```

3.6.7.2 Regression *R* squared: RSquaredCalc

Description:This function calculates the R squared coefficient.
Acceleration: No Acceleration.
Initialize the temporary variables and compute the rsquared result using the minimal hardware resources. Since this computation involves several complex mathematical operators, **#pragma HLS PIPELINE** is explicitly set to off and **#pragma HLS UNROLL** is set to use a factor of 1. R2_1 computes the average fitted values and R2_2 computes the sum of squared residuals (rss) and the total sum of squares (tss).

```
837  real RSquaredCalc(real *coeff, real *x, real *y, int ndim)
838  {
839   real r_value = 0;
840   real predict [1000] = {0};
841   real rss = 0;
842   real tss = 0;
843   real y_mean = 0;
```

```
R2_1:
  for (int i = 0; i < ndim; i++)
  {
#pragma HLS pipeline off
#pragma HLS unroll factor = 1
#pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
    y_mean += y[i];
  }
  y_mean = (y_mean / ndim);

R2_2:
  for (int i = 0; i < ndim; i++)
  {
#pragma HLS pipeline off
#pragma HLS unroll factor = 1
#pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
    predict[i] = (coeff[0] + (coeff[1] * x[i]));
    rss += hw_pow((predict[i] - y[i]), 2);
    tss += hw_pow((y[i] - y_mean), 2);
  }
  r_value = (1.0 - (rss / tss));

  return r_value;
```

3.6.8 Math Functions

Collection of double precision operatiors - (hw_exp, hw_log, hw_sqrt, hw_fabs, hw_pow)

When the math operators are implemented in the fpga, they use the bit-approximate HLS math library functions which do not have the same accuracy as the standard C function. To achieve the same result, these functions use a different underlying algorithm from the standard C functions. The accuracy of this is between 1-4 ULP (Unit of Least Precision). If the standard math.h is used, there can be differences between the C simulation results and the RTL co-simulation results due to the fact of having different underlying function definitions as explained above. However, if we use the Vitis HLS Math Library (hls_math.h), there will be no difference between the C simulation and the RTL co-simulation. However, as hls_math.h is not optimized to run on CPU, using the hls mathematical operators results in longer execution times during the sw_emu. For example, In hw_exp function hls::exp uses the function from hls_math.h. This function is also inlined.

```
real hw_exp(real b)
{
#pragma HLS inline
#if USE_HLS_LIB
  return hls::exp(b);
#else
  return exp(b);
#endif
}
```

3.6.9 Linear Interpolation

3.6.9.1 hw_findrange

Description:This function uses an optimized routine to find the interpolation range. The function comes in five versions, which differ in the size of the interpolation grids: new_hw_findrange_n4, hw_findrange_n8, hw_findrange_n100, hw_findrange_n200, hw_findrange_n300.
Acceleration: Unrolling, Pipelining..

```
1096 small_idx_t hw_findrange(fixed_t p, const fixed_t *src, int n_elem)
1097 {
1098 #if !_BASELINE
1099 #pragma HLS inline
1100 #if (NKM_GRID == 4)
1101   if (n_elem == 4)
1102    return hw_findrange_n4(p, src);
1103 #elif (NKM_GRID == 8)
1104   if (n_elem == 8)
1105    return hw_findrange_n8(p, src);
1106 #endif
1107 #if (NKGRID == 100)
1108   else if (n_elem == 100)
1109    return hw_findrange_n100(p, src);
1110 #elif (NKGRID == 200)
1111   else if (n_elem == 200)
1112    return hw_findrange_n200(p, src);
1113 #elif (NKGRID == 300)
1114   else if (n_elem == 300)
1115    return hw_findrange_n300(p, src);
1116 #endif
1117   else
1118    return 0;
1119 #else
1120   small_idx_t result = 1;
1121   for (signed short i = (n_elem - 1); i > 0; --i)
1122   {
1123    if (p <= src[i])
1124    {
1125     result = i - 1;
1126    }
1127   }
1128   return result;
1129 #endif
1130 }
```

Based on the selection of the NKGRID, NKM_GRID, the appropriate functions will be synthesized and the rest will be disabled. A generic function can be designed that could work efficiently for all the different grids, but that is left for future experiments.

We accelerate interpolation as follows. First, we declare the loop bounds of the individual and aggregate capital grids (namely, $\{0, N_k\}$ and $\{0, N_M\}$) as fixed constants, allowing the compiler to autonomously physically *place* the required CL resources (*space dimension*). Next, we implement a jump search algorithm to find the interpolation interval over the individual capital grid. The compiler instructs the hardware to pipeline a parallel reduce tree algorithm with three stages.

Each stage determines the index of the smallest grid value larger than the interpolation point $k'(k, \epsilon, m, A)$ by performing comparisons in parallel. The number of comparisons varies by stage and grid size and ensures that the entire grid is examined, $i = \{0, \ldots, N_k\}$. The winner of each stage determines the search area of the successive stage. Since the result of this operation is part of a pipeline where the only dependence on subsequent loop iterations is through a final accumulation, we achieve an **II** of 1.

Notice that the input to this function is of fixed point data type rather than the standard double precision. The floating point comparison is implemented using dcmp (Double precision comparator) operator which consumes significant amount of hardware resources. Therefore, we type cast the input data type of fixed point data type and use the grid of values which are in fixed point representation to perform all the 100 comparisons using icmp (Integer comparator) which consumes minimal resources.

Importantly for context, the CPU cannot physically place CL resources to make these comparisons in parallel, as its silicon is pre-manufactured and cannot be programmed. We could potentially implement the described parallel-search algorithm using multiple cores. But this design would be very inefficient, as the data transfer overhead costs would dominate the increase in performance. Conversely, our single FPGA vs. single CPU core and multi-core CPU benchmarking exercises are efficient, as they keep all CPU cores busy, minimizing data transfer overhead costs.[2]

```
1179 small_idx_t hw_findrange_n100(fixed_t p, const fixed_t *src)
1180 {
1181 #pragma HLS pipeline
1182   small_idx_t result_1 = 0;
1183   small_idx_t result_2 = 0;
1184   small_idx_t result_3 = 0;
1185   small_idx_t result = 0;
1186
1187  fr100_1:
1188   for (signed short i = 99; i > 0; i=i-20) //5 comparators
1189   {
1190   fr100_2:
1191    if (p <= src[i])
1192    {
1193     result_1 = i; //send the max idex
1194    }
1195   }
1196
1197  fr100_3:
1198   for (signed short i = 4; i > 0; i--) // 4 comparators
1199   {
1200   fr100_4:
1201    if (p <= src[result_1])
1202    {
1203     result_2 = result_1; //send the max index
1204    }
1205    result_1 = result_1 - (small_idx_t)5;
1206   }
```

[2]The C++ to CPU compiler can autonomously decide to perform these operations in parallel, but this step is not controlled by the coder.

```

fr100_5:
  for (signed short i = 5; i > 0; i--) //5 comparators
  {
  fr100_6:
    if (p <= src[result_2--])
    {
      result_3 = result_2; //send the min index
    }
  }

  result = (p==src[0]) ? (small_idx_t)0 : result_3;
  return result;
}
```

3.6.9.2 hw_rail_values

Description:This function set the values outside the range to the range values.
Acceleration: Inline..
The **#pragma HLS INLINE** synthesizes separate hardware each time the function is called.

```
void hw_rail_values(real *val, const real max, const real min)
{
#pragma HLS inline
  real src = *val;
  bool over_max = (src > max);
  bool under_min = (src < min);

hw_rail_1:
  if (over_max)
    *val = max;
  else if (under_min)
    *val = min;
  return;
}
```

3.7 FPGA Configuration & Runtime Initialization

3.7.1 Configuration File: design.cfg

Description. The Vitis allows the user to control the compiler and the linker behavior using the configuration file. More information regarding the different options can be found here.

```
#check if the platform is the latest version
platform=xilinx_aws-vu9p-f1_shell-v04261818_201920_3
debug=1
profile_kernel=data:all:all:all
save-temps=1

[hls]
pre_tcl=hls_config.tcl
```

In our baseline model, we use three kernels. Therefore, the three kernel names are defined here under the *connectivity*. We further specify the SLR names for each of these three kernels followed by the DDR port assignment. The xclbin utility provides us with the information about the DDR ports that are attached to each of the SLR. By using the respective ports, we can minimize the SLR crossings. If no details are specified in the configuration file, the compiler automatically tries to configure the ports which may not be optimal.

The following command can be executed in the terminal after setting the environment variables to get the information of the DDR ports.

```
source $AWS_FPGA_REPO_DIR/vitis_setup.sh
export PLATFORM_REPO_PATHS=$(dirname $AWS_PLATFORM)
platforminfo -$AWS_FPGA_REPO_DIR
```

```
#Enable either single kernel or three kernel
##############single kernel start ################
# [ connectivity ]
# nk=runOnfpga:1:runOnfpga_1
##############single kernel end##################

################three kernel start ################
[ connectivity ]
nk=runOnfpga:3:runOnfpga_1.runOnfpga_2.runOnfpga_3

# slr =<compute_unit_name>:<slr_ID>
slr =runOnfpga_1:SLR2
slr =runOnfpga_2:SLR1
slr =runOnfpga_3:SLR0

# [ connectivity ]
sp=runOnfpga_1.hw_agshock:DDR[1]
sp=runOnfpga_1.hw_idshock:DDR[1]
sp=runOnfpga_1.preinit:DDR[1]
sp=runOnfpga_1.results:DDR[1]
sp=runOnfpga_1.hw_iter:DDR[1]

sp=runOnfpga_2.hw_agshock:DDR[0]
sp=runOnfpga_2.hw_idshock:DDR[0]
sp=runOnfpga_2.preinit:DDR[0]
sp=runOnfpga_2.results:DDR[0]
sp=runOnfpga_2.hw_iter:DDR[0]

sp=runOnfpga_3.hw_agshock:DDR[3]
sp=runOnfpga_3.hw_idshock:DDR[3]
sp=runOnfpga_3.preinit:DDR[3]
sp=runOnfpga_3.results:DDR[3]
sp=runOnfpga_3.hw_iter:DDR[3]
################three kernel end##################
```

```
[vivado]
#prop=run.impl_1.strategy=Performance_Explore
#prop=run.impl_1.strategy=Performance_NetDelay_high
#prop=run.impl_1.strategy=Performance_WLBlockPlacementFanoutOpt
#prop=run.impl_1.strategy=Performance_WLBlockPlacement
#prop=run.impl_1.strategy=Performance_ExploreWithRemap
# prop=run.impl_1.strategy=Performance_BalanceSLRs
# prop=run.impl_1.strategy=Performance_EarlyBlockPlacement
```

```
prop=run.impl_1.strategy=Performance_ExtraTimingOpt
#prop=run.impl_1.strategy=Performance_NetDelay_low
# prop=run.impl_1.strategy=Congestion_SpreadLogic_low
#param=place.runPartPlacer=0
```

3.7.2 Configuration File: hls_config.tcl

Description. While implementing the logic, some of the mathematical operators consumes considerably large number of hardware resources. The user needs to make conscious of the number of pipelines that are to be implemented whenever it involves several mathematical operators. As discussed during hw.cpp file, the functions sim_alm_coeff, regression, RsquaredCalc, hw_sim_alm consumes several resources when it is left to compile with the default settings. Therefore, we intsruct the compiler to limit the number of hardware operators using the following directives. For example, we limit the number of calls to the regression function to 1 from sim_alm_coeff function. This implies that if the prior function is being called 3 times, the compiler will implement the logic only once but utilize it thrice.

```
config_interface -m_axi_max_widen_bitwidth 512
set_directive_allocation -limit 1 -type function sim_alm_coeff regression
set_directive_allocation -limit 1 -type function sim_alm_coeff RSquaredCalc
set_directive_allocation -limit 1 -type function sim_alm_coeff hw_log
set_directive_allocation -limit 1 -type function regression hw_pow
set_directive_allocation -limit 1 -type function RSquaredCalc hw_pow
set_directive_allocation -limit 1 -type function hw_sim_alm hw_exp
set_directive_allocation -limit 1 -type function hw_sim_alm hw_log
set_param route.enableGlobalHoldIter true
```

3.7.3 Xilinx Runtime Library: xrt.ini

Description. The Xilinx runtime (XRT) uses various parameters to control execution flow, debug, profiling, and message logging during host application and kernel execution in software emulation, hardware emulation, and system run on the acceleration board. These control parameters are optionally specified in a runtime initialization file xrt.ini. This file needs to be created manually and saved to the same directory as the host executable. The runtime library checks if xrt.ini exists in the same directory as the host executable and automatically reads the file to configure the runtime.

In our program, we place this file in the parent directory. Alternatively, the file can be placed in a different location and the following command can be used to set the directory of the xrt.ini file.

```
export XRT_INI_PATH=/path/to/xrt.ini
```

The below code snippet of the xrt.ini file shows that the profile, data transfer trace and summary are set to true.

```
#Start of Debug group
[Debug]
```

```
profile =true
timeline_trace =true
data_transfer_trace =coarse
opencl_summary=true
opencl_device_counter=true
opencl_trace=true
```

3.8 Run on the FPGA

Connect to your f1.2xlarge and execute the following commands from the terminal for setting up the Xilinx environment and to clone the project.

```
git clone https :// github.com/aws/aws-fpga.git $AWS_FPGA_REPO_DIR //AWS repo
git clone https :// github.com/AleP83/KS-FPGA.git -b "dev_accel"  //KS-FPGA Project
```

Navigate to the parent directory (KS-FPGA/baseline/codes/accel/src/fpga) within the cloned KS-FPGA folder and execute the following command to generate the computations of the baseline economy for 1200 computations.

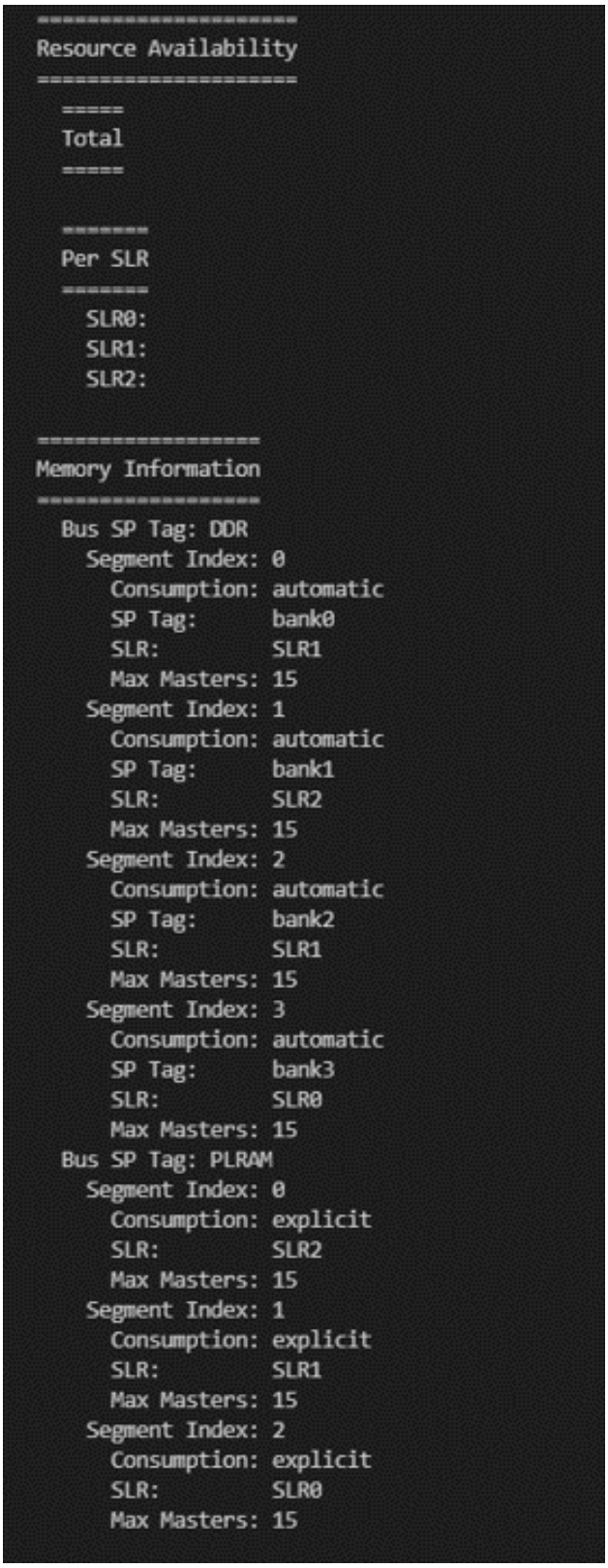
```
make results
```

Once the results are computed, execute the following command to copy all the logs, reports and summary files into a single folder (single.zip) and download this folder to your local PC to analyze the results.

```
make zip
```

Note: Make sure to terminate your F1 instance! It costs 1.65$/hr.

Figure 3.6: Information from xclbinutil



3.9 Makefile

This file is in the parent directory (KS-FPGA/baseline/codes/accel/src/fpga) within the cloned KS-FPGA project. Makefile is a tool that we use to compile source code into executable programs, run scripts, parse and combine files. It is designed to automatically update the outputs when there is a change in any of the dependencies. A simple tutorial for the Makefile can be found here.

In the below code snippet, we show the build process of the AWSXCLBIN file that can be executed on AWS f1 instance. We start by defining the variables that we use in the later section of the code.

```
41 TARGET := hw
42 MPICXX := mpic++
43 CC := g++
44 INCLUDES := -I./common -I./common/libs -I./cpu -I ./fpga -I ./ -I$(XILINX_XRT)/include/ -I$(XILINX_VIVADO)/include/
45 PLATFORM := xilinx_aws-vu9p-f1_shell-v04261818_201920_3
46 HOST_EXE := host
47 CPU_EXE := app
48 OPENMPI_EXE := openmpi_app
49 XO := ./fpga/build/runOnfpga.xo
50 XCLBIN := ./fpga/build/runOnfpga.xclbin
51 S3_BUCKET_NAME := ksfpga-$(shell aws sts get-caller-identity | grep "Account" | tr -dc '0-9')
52 S3_DCP_DIR := vitis-dcps
53 S3_LOG_DIR := vitis-logs
54 EMAIL := # enter your email address
55 SHELL := /bin/bash
56 CPU_CORES := 1 #set the number of CPU cores
```

These three flags are defined so that the host program can determine the target application. Notice that $-D$ lets us pass a particular flag during compilation. As we see that the below code is for fpga, the FPGA_FLAG is being passed while building the host program.

```
57 OPENMPI_FLAG := -D_OPENMPI_MODE
58 FPGA_FLAG := -D_FPGA_MODE
59 SERIAL_CPU_FLAG := -D_SERIAL_CPU_MODE
```

The below script is drawn from the tutorial provided by AWS. We utilize the scripts provided by AWS to generate the .AWSXCLBIN file from the .XCLBIN file.

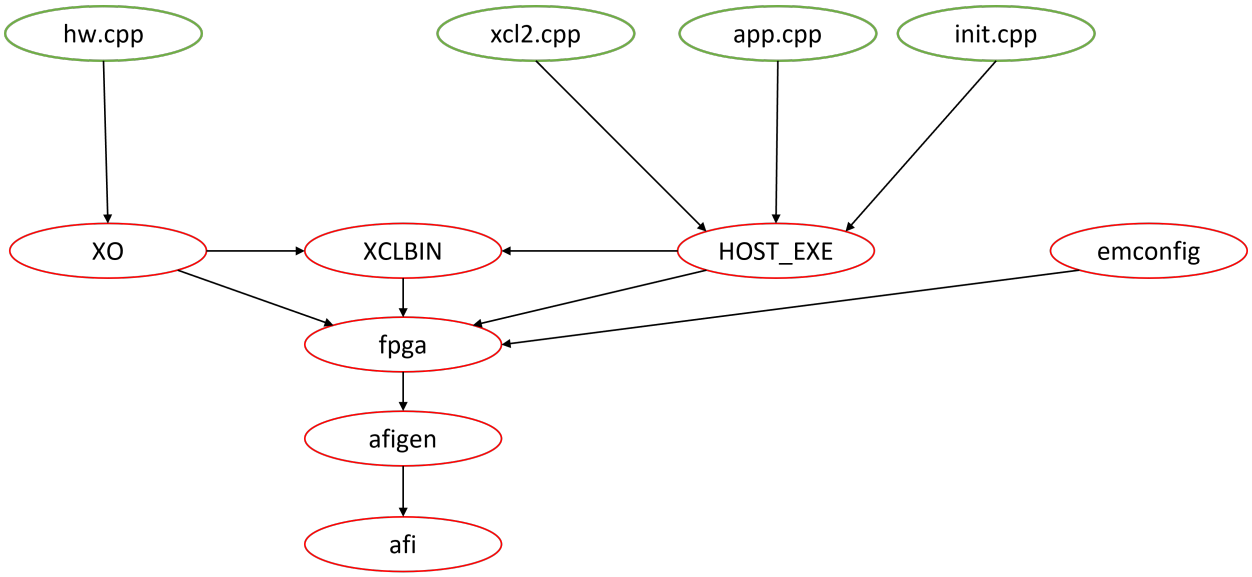
```
70 .PHONY: afi
71 afi: afigen
72  source $(AWS_FPGA_REPO_DIR)/hdk_setup.sh
73  pip install --user --upgrade boto3
74  wait_for_afi.py --afi $(shell cat *afi_id.txt | sed -n '2p' | tr -d '",' | sed 's/.*://') --notify --email $(EMAIL) &
75
76 .PHONY: afigen
77 afigen: fpga
78  aws s3 mb s3://$(S3_BUCKET_NAME) --region us-east-1
79  touch FILES_GO_HERE.txt
80  aws s3 cp FILES_GO_HERE.txt s3://$(S3_BUCKET_NAME)/$(S3_DCP_DIR)
81  touch LOGS_FILES_GO_HERE.txt
82  aws s3 cp LOGS_FILES_GO_HERE.txt s3://$(S3_BUCKET_NAME)/$(S3_LOG_DIR)
83  rm -rf to_aws
84  $(VITIS_DIR)/tools/create_vitis_afi.sh -xclbin=$(XCLBIN) -s3_bucket=$(S3_BUCKET_NAME) -s3_dcp_key=$(S3_DCP_DIR) -
        s3_logs_key=$(S3_LOG_DIR)
```

```
85
86 fpga: $(XO) $(XCLBIN) $(HOST_EXE) emconfig
```

The dependency for the following code snippet is shown in the Figure 3.7.

```
98 # Building kernel
99 $(XO): ./fpga/hw.cpp
100  v++ -I./common -I./fpga -I./  $(FPGA_FLAG) $(EGM_UNTIL_CONV_FLAG) $(KRNL_COMPILE_OPTS) -c -k runOnfpga -o'$@' '$<'
101
102 $(XCLBIN): $(XO)
103  v++ -I./common -I./fpga -I./  $(KRNL_LINK_OPTS) -l -o'$@' $(+)
104
105 # Building fpga Host for EGM until convergence
106 $(HOST_EXE): ./common/libs/xcl2.cpp ./common/app.cpp ./common/init.cpp
107  $(CC) $(FPGA_FLAG) $(EGM_UNTIL_CONV_FLAG) $(CXXFLAGS) $^ -o $@ $(CXXFLAGS2)
```

Figure 3.7: Simplified Data dependency chart for generating AWSXCLBIN



3.10 Command Guidelines

3.10.1 OpenCL Commands Description

This section provides a comprehensive list of the OpenCL commands used to design the communications between host and FPGA device(s) and the computation workflow. *Source:* Open CL Official Manual. Xilinx Documentation - UG1393 Kronos OpenCL Documentation.

3.10.1.1 Gathering information about platforms

- **Command**: **cl::Context**
- **Description:** The cl::Context API is used to create a context that contains a Xilinx device that will communicate with the host machine.
- **Command**: **cl::Platform**
- **Description:** Upon initialization, the host application needs to identify a platform composed of one or more Xilinx devices.
- **Command**: **cl::Platform::get**
- **Description:** Gets a list of available platforms.

3.10.1.2 Programming the device

- **Command**: **cl::Program::Binaries**
- **Description:**
- **Command**: **cl::Program**
- **Description:** Program interface that implements cl_program

3.10.1.3 Command Queue

- **Command**: **cl::CommandQueue**
- **Description:** The cl::CommandQueue API creates one or more command queues for each device. The FPGA can contain multiple kernels, which can be either the same or different kernels. When developing the host application, there are two main programming approaches to execute kernels on a device:
  - Single out-of-order command queue: Multiple kernel executions can be requested through the same command queue. XRT dispatches kernels as soon as possible, in any order, allowing concurrent kernel execution on the FPGA.

– Multiple in-order command queue: Each kernel execution is requested from different in-order command queues. In such cases, XRT dispatches kernels from the different command queues, improving performance by running them concurrently on the device.

The following is an example of standard API calls to create in-order and out-of-order command queues.

```
// In-order Command Queue
commands = clCreateCommandQueue(context, device_id, 0, err);
```

3.10.1.4 Kernels

- **Command**: **cl::Kernel**
- **Description:** Identifies a kernel in the program loaded into the FPGA that can be run by the host application.

3.10.1.5 Buffers

- **Command**: **cl::Buffer**
- **Description:** Interactions between the host program and hardware kernels rely on creating buffers and transferring data to and from the memory in the device. cl::Buffer constructs a buffer in a specified context.

3.10.1.6 Events

- **Command**: **cl::Event**
- **Description:** Class interface for cl_event

3.10.1.7 Memory Transfer & Kernel Computation Management

- **Command**: **cl::enqueueMigrateMemObjects**
- **Description:** Enqueues a command to indicate which device a set of memory objects should be associated with. Using this API, memory migration can be explicitly performed ahead of the dependent commands.
- **Command**: **cl::enqueueTask**
- **Description:** When the kernel is compiled to a single hardware instance (or CU) on the FPGA, the simplest method of executing the kernel is using cl::EnqueueTask which enqueues a command to execute a kernel on a device.

3.10.2 Error Management

- cl_int err
- OCL_CHECK(err, buffer_in_coeffs[d][k] = cl::Buffer(contexts[d], CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, hw_coeff_size_bytes, in_coeff[d][k].data(), err));

3.10.2.1 Computation Flow

3.10.3 Pragmas Description

This section provides a comprehensive list of the pragmas used to accelerate the code.

- **Command**: **#pragma HLS PIPELINE**
- **What it does:** The PIPELINE pragma tells the compiler to start each iteration of the loop immediately, if possible, rather than waiting for the loop body to finish before starting the next iteration of the loop. This allows multiple loop iterations to run concurrently on the same hardware, decreasing runtime. Xilinx link
- **Command**: **#pragma HLS ARRAY_PARTITION**
- **What it does:** Partitions an array into smaller arrays or individual elements. This can allow the on-chip memories to perform more reads in parallel. Xilinx link
- **Command**: **#pragma HLS UNROLL**
- **What it does:** The UNROLL pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel. Xilinx link
- **Command**: **#pragma HLS BIND_STORAGE**
- **What it does:** The BIND_STORAGE pragma assigns a variable (array, or function argument) in the code to a specific memory type in the RTL Xilinx link
- **Command**: **#pragma HLS LOOP_TRIPCOUNT**
- **What it does:** When manually applied to a loop, specifies the total number of iterations performed by a loop. This can help the tools in estimating the performance for the application. Xilinx link
- **Command**: **#pragma HLS INLINE**
- **What it does:** Removes a function as a separate entity in the hierarchy. This reduces the overhead for the function call and can allow the function to be optimized into the caller. When you inline, you will have a separate set of hardware for each place where the function is inlined. Xilinx link