

A Beginner's Guide to
Programming FPGAs for Economics:
An Introduction to Electrical Engineering Economics

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-VILLAYERDE

University of Pennsylvania, Economics
jesusfv@econ.upenn.edu

ALESSANDRO PERI

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

[[Click here for the Latest Version](#)]

Last Update: Monday 2nd September, 2024

Acknowledgements

First, we wish to thank Syed Ahmed (UPenn, Electrical and System Engineering). The material in Chapters 1 and 4 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 Xilinx, 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 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

1	Setup and Walk-through	1
1.1	Getting Started with Vitis on Amazon F1 Instance	1
1.2	Step 1: Launch the build instance	2
1.3	Step 2: Setup remote desktop	3
1.4	Step 3: Setup AWS CLI	5
1.5	Step 4: Edit Source Files in Build Instance.	5
1.6	Step 5: Build Phase	5
1.6.1	Initialize the Environment	6
1.6.2	Create a Project in Vitis HLS	6
1.6.3	C Simulation and Code Debugging	7
1.6.4	Synthesis in Vitis HLS	8
1.6.5	HLS Kernel Optimization using the <i>Vitis HLS</i> IDE	9
1.6.6	Compile the Hardware Function	9
1.7	Step 6: Runtime Phase	10
1.7.1	Set up a runtime instance	10
1.7.2	Run the application on the FPGA	10
1.8	Create an S3 bucket	10
2	Accumulator	13
2.1	Directory Structure	13
2.2	The Code	14
2.3	Setup and Launch	14
2.3.1	Compile and Execute on a CPU	14
2.3.2	Compile and Execute on an FPGA	16
2.4	Header Files	22
2.4.1	Accumulator Designs: <i>hw.cpp</i>	23
3	Krusell Smith (1998)	25
3.1	Directory Structure	25
3.2	The Code	26
3.3	Setup and Launch	26
3.3.1	Compile and Execute on a CPU	26
3.3.2	Compile and Execute on an FPGA	32
3.4	Header Files	39
3.5	Boiler-plate code: <i>app.cpp</i>	42
3.5.1	Overview	42

3.5.2	Setting up the OpenCL environment	42
3.5.3	Allocate the Buffers and Events	44
3.5.4	Set Up Kernels and Initialize Buffers	45
3.5.5	Copy Input from Host to Device	48
3.5.6	Submit Kernel for Execution	48
3.5.7	Copy the results back	49
3.5.8	Event Synchronization	49
3.5.9	Printing Results	49
3.5.10	Open MPI	50
3.6	Kernel: hw.cpp	53
3.6.1	Common HLS Optimization Pragmas	53
3.6.2	Overview	57
3.6.3	Parent Kernel Function: runOnfpga	57
3.6.4	Aggregate Law of Motion: hw_sim_alm	64
3.6.5	Individual Household Problem: hw_sim_ihp	65
3.6.6	Stochastic Simulation: hw_sim_ast	71
3.6.7	Aggregate Law of Motion: sim_alm_coeff	76
3.6.8	Math Functions	81
3.6.9	Linear Interpolation	81
3.7	FPGA Configuration & Runtime Initialization	85
3.7.1	Configuration File: design.cfg	85
3.7.2	Xilinx Runtime Library: xrt.ini	86
3.8	Makefile	88
3.9	Command Guidelines	94
3.9.1	OpenCL Commands Description	94
3.9.2	Error Management	96
3.9.3	Pragmas Description	96
4	Matrix Multiplier	97
4.1	Directory Structure	97
4.2	The code	97
4.2.1	Host.cpp: the main	98
4.2.2	MatrixMultiplication.cpp: the kernel	100
4.2.3	design.cfg: Compiler Flags	100
4.2.4	xrt.ini: Vitis Analyzer	100
4.3	CPU implementation.	101
4.4	Create a Project in Vitis	101
4.5	C Simulation and Code Debugging	102
4.6	Synthesis in Vitis HLS	102
4.6.1	Synthesis Report	103

4.6.2	Resources	103
4.6.3	Scheduler View	103
4.6.4	Data Flow	103
4.7	HLS Kernel Optimization: Loop Unrolling	103
4.7.1	Resource Profile	104
4.7.2	Full Unroll	105
4.8	HLS Kernel Optimization: Pipelining	105
4.8.1	Understanding the Initiation Interval (II)	105
4.8.2	Partitioning Arrays to Improve Pipelining	106
4.8.3	Export the Vitis Kernel	106
4.9	Run on the FPGA	107
4.10	Additional Documentation	108

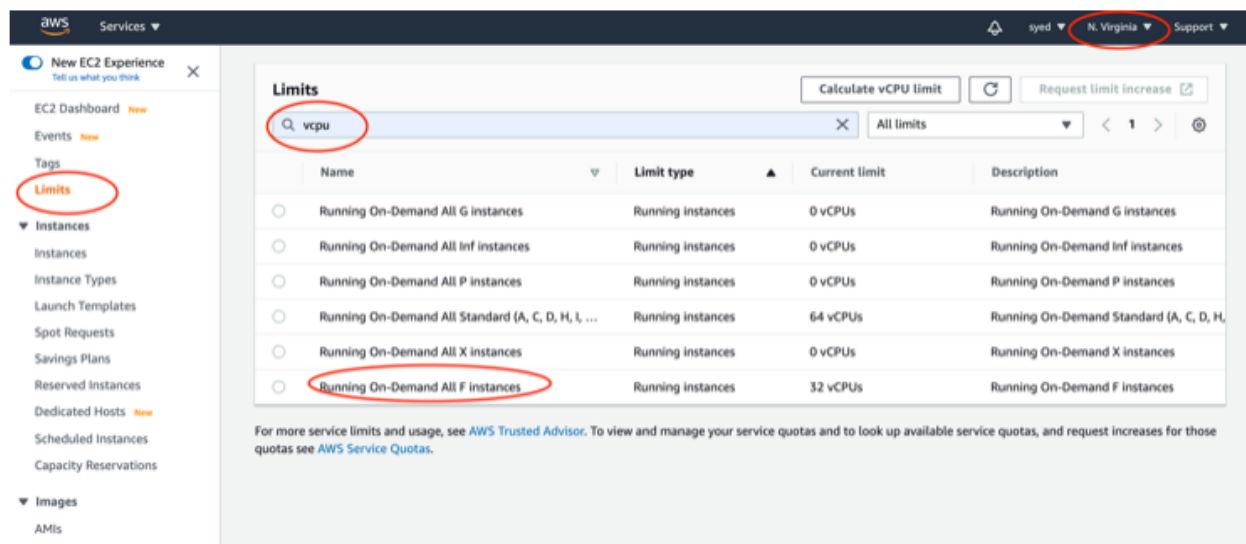
Setup and Walk-through

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](#).



The screenshot shows the AWS Management Console 'Limits' page. The left sidebar has 'Limits' highlighted. The search bar contains 'vcpu'. The table lists various instance types and their vCPU limits. The 'Running On-Demand All F instances' row is highlighted with a red circle.

Name	Limit type	Current limit	Description
Running On-Demand All G instances	Running instances	0 vCPUs	Running On-Demand G instances
Running On-Demand All Inf instances	Running instances	0 vCPUs	Running On-Demand Inf instances
Running On-Demand All P instances	Running instances	0 vCPUs	Running On-Demand P instances
Running On-Demand All Standard (A, C, D, H, I, ...)	Running instances	64 vCPUs	Running On-Demand Standard (A, C, D, H, I, ...)
Running On-Demand All X instances	Running instances	0 vCPUs	Running On-Demand X instances
Running On-Demand All F instances	Running instances	32 vCPUs	Running On-Demand F instances

2. Read about Vitis from [here](#).

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

In this tutorial, we will mostly 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. You may choose us-east-1 (N. Virginia) as the instance region. To explore availability in other regions, please visit this [link](#).

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:

```
1 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:

```
1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Effect": "Allow",
6       "Action": "s3:GetObject",
7       "Resource": "arn:aws:s3:::dcv-license.us-east-1/*"
8     }
9   ]
10 }
```

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](#).

```
1 sudo yum update
2 sudo yum install kernel-devel
3 sudo yum groupinstall "GNOME Desktop"
4 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](#)).

```
1 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-e17.tgz
tar xvf nice-dcv-2019.0-7318-e17.tgz
cd nice-dcv-2019.0-7318-e17
sudo yum install nice-dcv-server-2019.0.7318-1.e17.x86_64.rpm
sudo yum install nice-xdcv-2019.0.224-1.e17.x86_64.rpm
cd ~

sudo systemctl enable dcvserver
sudo systemctl start dcvserver
```

5. Setup a password

```
1 sudo passwd centos
```

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

```
1 sudo systemctl stop firewalld
2 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..

```
1 dcw create-session --type virtual --user centos centos
```

8. Connect to the DCV Remote Desktop session

- Download and install the [DCV Client](#) in your computer².
- 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](#), log in, 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:

```
1 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](#).

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

- **Synthesis of the Code**, which create the AFI executable which you can run on the f1 instance

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:

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

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

2. Clone your git repository using the following command:

```
1 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:

```
1 dcvm create-session --type virtual --user centos centos
2
```

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 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 with the `#pragma` command. Examples:

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

The different `#pragma` that you can use are listed in the [Vitis HLS User Guide](#). (If this link does not work, use Chrome).

3. Re-compile it;
4. Once you are 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. More information about this [here](#). 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.awsxcclbin` 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

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

You should see the following files generated when you ran:

```
1 profile_summary.csv
2 timeline_trace .csv
3 xclbin .run_summary
```

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

1.8 Create an S3 bucket

To facilitate file transfer from the build instance (**z1d.2xlarge**) to the run instance (**f1.2xlarge**), consider creating S3 buckets. This setup only needs to be performed once.

Note: Our replication code automatically generates two designated buckets, **fpga-econ-ks** and **fpga-econ-acc**, for reproducing results based on Krusell and Smith (1998) and accumulator algorithms. Accordingly, you can skip this section if you are not interested in experimenting on your own.

1. Log into your AWS account:

- Visit the [AWS Management Console](#).
- Click on the “Sign in to the Console” button and enter your AWS account credentials.

2. Navigate to the Home Console:

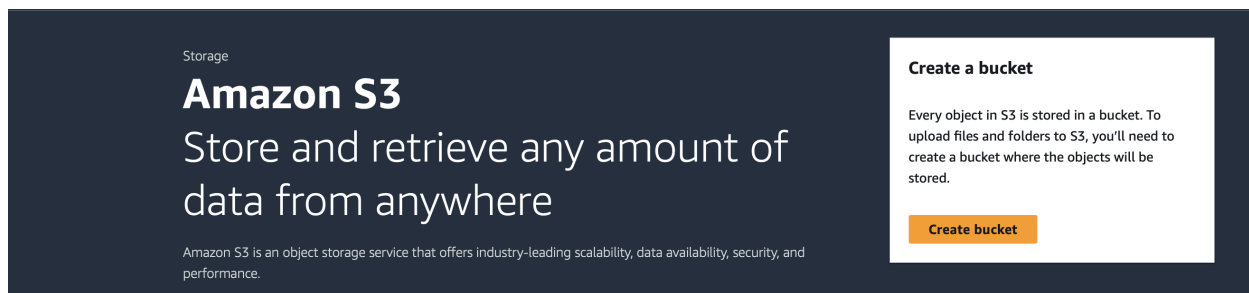
- After logging in, you should be on the AWS Management Console home page.
- If not, you can click on the “AWS” logo at the top-left corner to go back to the home page.

3. Select S3:

- In the AWS Management Console, use the search bar at the top and type “S3” to find and select the Amazon S3 service.
- Alternatively, you can navigate to the “Storage” section and click on “S3.”

4. Create a Bucket:

- In the S3 console, click on the “Create bucket” button.



- Follow the prompts to configure your new S3 bucket.
 - Provide a unique name for your bucket (in this case **fpga-econ**) and choose a region (in this case US East, North Virginia)
- IMPORTANT:** Make sure that the S3 bucket is in the same region as your AWS instance (e.g. US East, North Virginia) or you will receive an error.

A screenshot of the 'Create bucket' configuration page in the AWS console. The title is 'Create bucket' with an 'Info' link. Below the title, it says 'Buckets are containers for data stored in S3. [Learn more](#)'. The main section is 'General configuration'. It contains a 'Bucket name' field with the value 'fpga-econ' and a note: 'Bucket name must be unique within the global namespace and follow the bucket naming rules. [See rules for bucket naming](#)'. Below that is an 'AWS Region' dropdown menu showing 'US East (N. Virginia) us-east-1'. At the bottom, there is a section for 'Copy settings from existing bucket - optional' with the text 'Only the bucket settings in the following configuration are copied.' and a 'Choose bucket' button.

Figure 1.1: Create bucket (Step 1)

- You have the option to configure additional settings, but we recommend keeping the rest of the configuration as it is.
- Review your settings, and if everything looks good, click the “Create bucket” button.

Henceforth, we will utilize the **fpga-econ** s3 bucket to conveniently store our files, ensuring seamless access across different AWS instances.

5. Set Up the Directory Structure in Your S3 Bucket. You can add folders to your S3 bucket by following these steps.

- Navigate to Amazon S3 > Buckets
- Select your specific bucket (**fpga-econ**)

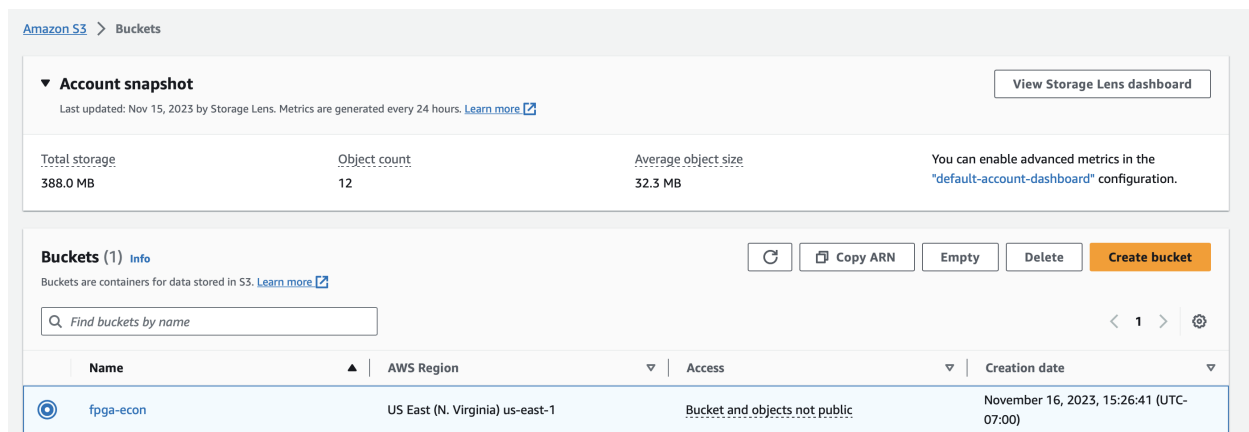


Figure 1.2: Create bucket (Step 2)

- Click the “Create folder” button.

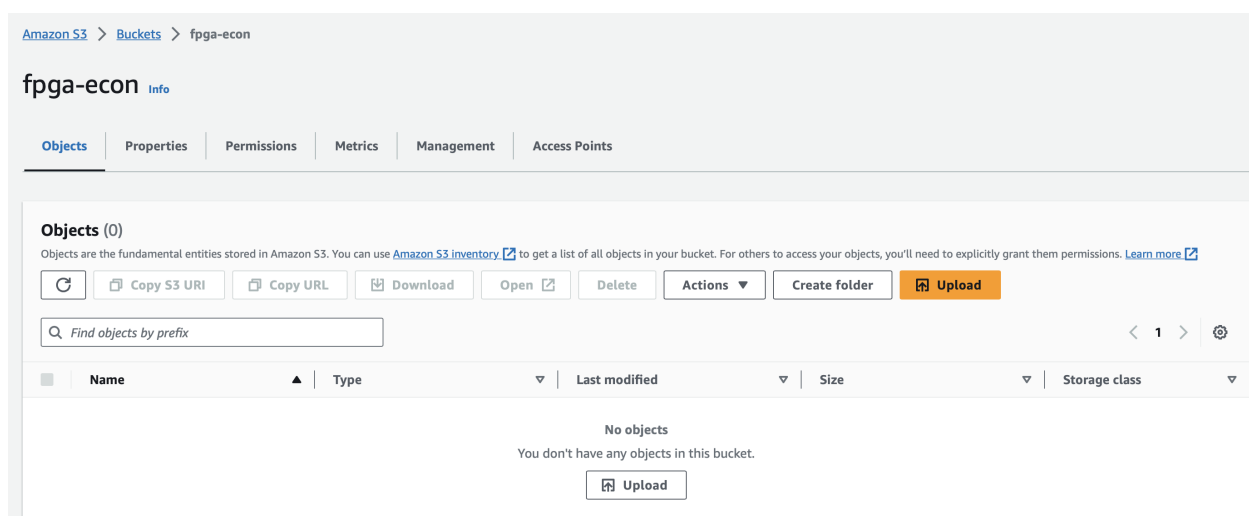


Figure 1.3: Create folders

Accumulator

This section describes the FPGA implementation of the accumulator described in Cheela et al. (2024).

2.1 Directory Structure

The code for the accumulator algorithm is contained in the directory `./V_accumulator` of our GitHub repository <https://github.com/AleP83/FPGA-Econ.git>.

The directory is structured into four folders. The folder `common` contains host code, supporting libraries, input files, and utility scripts. The folder `fpga` holds the kernel for execution on both FPGA and CPU. Results are stored in the folder `results`. For convenience, the directory `executables` stores the executables for CPU and FPGA acceleration.

```
common
  app.cpp
  app.h
  definitions.h
  dev_options.h
  init.cpp
  init.h
  libs
    ap_common.h
    ap_decl.h
    ap_fixed.h
    ap_fixed_base.h
    ap_fixed_ref.h
    ap_fixed_special.h
    ap_int.h
    ap_int_base.h
    ap_int_ref.h
    ap_int_special.h
    etc
    xcl2.cpp
    xcl2.hpp
    xcl2.mk
  stopwatch.h
  util
```

```
generate_fpga_results.sh
executables
  cpu
    app
  fpga
    fpga_afi
    host_executables
fpga
  design.cfg
  hw.cpp
  hw.h
  hw_base.cpp
  hw_opt.cpp
  hw_pipeline.cpp
  hw_unroll.cpp
results
  cpu
    final_values
  fpga
    final_values
hls_config.tcl
Makefile
README.md
xrt.ini
```

2.2 The Code

- **Makefile.** Run the [Makefile](#) to execute the application. The Makefile has 2 main targets that allow you to choose the execution mode:

- Execution on CPU: [make cpu_to_s3](#);
- Execution on FPGA: [make fpga](#).

There are other auxiliary targets. Execute [make help](#) to learn more about them. See section [2.3](#) 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 CPU/FPGA hardware execution, fetches back the result from the kernel.
- **Kernel.** The [/fpga/hw.cpp](#) contains the Vitis kernel for FPGA and CPU execution.
- **Results.** Results are stored in [/results](#).
- **Header Files.** Header files and helper functions are contained in the following directory
 - [/common](#): boiler-plate code shared by FPGA and CPU
 - [/common/libs](#): libraries for FPGA software emulation
 - [/fpga](#): kernel files
- **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**.

2.3 Setup and Launch

This section summarizes the steps required to compile and run the application under the different acceleration modes provided in the [Makefile](#).

2.3.1 Compile and Execute on a CPU

These steps describe how to compile, execute, and store the results of the accumulator algorithm on the [S3](#)-bucket named [fpga-econ-acc](#). All these steps are conducted on a build instance [z1d.2xlarge](#) (but you can use also a more inexpensive instance, e.g. [m5n.large](#)).

- **Launch the Instance.** Log into a build instance, [z1d.2xlarge](#). To set up and launch the instance, follow the instructions in [documents/FPGA-design.pdf](#). *Note:* If you are using an [m5n.large](#) instance, use instructions in [documents/CPU-run.pdf](#).
- **Install the Packages.** Initiate a terminal session on the AWS instance and run the subsequent script to install the utilities `git`, `make`, `tmux` and `wget`:

```
sudo yum install git -y
sudo yum install make -y
sudo yum install tmux -y
sudo yum install wget -y
```

- **Clone the GitHub repositories.** Clone our GitHub repository into a directory of your preference (e.g., `/home/centos/`).

```
git clone https://github.com/AleP83/FPGA-Econ.git
```

Note: If you are using an [m5n.large](#) instance the home directory is `/home/ec2-user/`.

- **Set the AWS Credentials.** Configure your AWS credentials by executing the following command in the terminal:

```
aws configure
```

Follow the steps here:

```
$ aws configure
AWS Access Key ID [***** xxxx]: <Your AWS Access Key ID>
AWS Secret Access Key [***** xxxx]: <Your AWS Secret Access Key>
Default region name: us-west-2
Default output format [None]: json
```

For more information visit this [link](#).

- **Compile, run and store results on S3-bucket.** Go to the directory [/V_accumulator/code](#). From there, you can use the following terminal instructions to compile, run, and automatically store the results of the accumulator algorithm on an AWS S3 bucket:

- **Modify the Makefile.** Update settings in the [code/Makefile](#) as follows:

- **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing [S3-NAME-GOES-HERE](#)

```
S3_EXE_BUCKET_NAME := S3-NAME-GOES-HERE
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is **us-west-2**):

```
AWS_REGION := us-west-2
```

- Run, compile and store the results on the S3 bucket:

```
make cpu_to_s3
```

Note: The script *automatically* creates an S3 bucket called `$S3_EXE_BUCKET_NAME`.

- **Compile and run.** If you just want to compile and run (without storing the results on an S3 bucket), then follow these steps. Go to the directory `/V_accumulator/code`. From there, use the following terminal instructions to compile and run the application:

```
make cpu
./ app
```

2.3.2 Compile and Execute on an FPGA

The following steps describe how to:

1. Synthesize the FPGA image on a build instance, **z1d.2xlarge**;
2. Execute it on an FPGA instance, e.g. **f1.2xlarge**;
3. Store the results of the accumulator algorithm in the **S3**-bucket named **fpga-econ-acc**.

Step 1: Synthesize the FPGA image

- **Launch the Instance.** Log into the AWS build instance **z1d.2xlarge**. To launch the instance, follow the instructions in [documents/FPGA-design.pdf](#). To set up the instance for development purposes—using for example NICE DCV for analysing the hardware design—follow the instructions in Section [1.2](#).
- **Clone the GitHub repositories.** Open the terminal. Then, clone the AWS repository and our GitHub repository into a directory of your preference (e.g., `/home/centos/`):

```
git clone https://github.com/aws/aws-fpga.git $AWS_FPGA_REPO_DIR
git clone https://github.com/AleP83/FPGA-Econ.git
```

- **Set the AWS credentials.** Configure your AWS credentials by executing the following command in the terminal:

```
aws configure
```

Follow the steps here:


```
$ aws configure
AWS Access Key ID [***** xxxx]: <Your AWS Access Key ID>
AWS Secret Access Key [***** xxxx]: <Your AWS Secret Access Key>
Default region name: us-west-2
Default output format [None]: json
```

For more information visit this [link](#).

- **Modify the Makefile.** Update settings in the `/V_accumulator/code/Makefile` as follows:

- **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing `S3-NAME-GOES-HERE`

```
S3_EXE_BUCKET_NAME := S3-NAME-GOES-HERE
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is `us-west-2`):

```
AWS_REGION := us-west-2
```

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

1. Execute Software emulation

- **Description.** The main goal of software emulation (`sw_emu`) is to ensure functional correctness of the host program and kernels (including the debugging of OpenCL instructions). 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 `/V_accumulator/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
```

Once you are happy with the performance of your FPGA design you can go move to the next step: the synthesis of the FPGA on hardware.

2. **Create all FPGA images.** To ensure your terminal session remains active throughout the potentially lengthy synthesis process, initiate a terminal multiplexer session:

```
tmux
```

The `tmux` command allows you to detach and reattach to terminal sessions without interruption. For example, to resume a `tmux` session with index 0, use the following command:

```
tmux attach -t 0
```

For detailed instructions on how to use `tmux`, see this [guide](#).

Create the FPGA Image: System Hardware Target

- **Description.** When we set as build target the hardware, HLS `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 on a build instance ([z1d.2xlarge](#)).** To initiate the synthesis of the FPGA circuit, navigate to the directory `V_accumulator/code` from within the `tmux` terminal window. Therein, execute the following instructions to generate the host and fpga target files on the build instance ([z1d.2xlarge](#)); and subsequently, upload the resulting executables to the AWS bucket:

```
make clean
unset XCL_EMULATION_MODE
//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(s)
make afi FPGA_BIN=afi_optimized HOST_BIN=host
```

This command generates the FPGA image for the accumulator described in the main paper. If you are interested in generating an FPGA circuit without any HLS optimizations (dusty deck), comment out the `#pragmas` (or copy the file `hw_base.cpp` in `hw.cpp`) and execute

```
//build the target(s)
make afi FPGA_BIN=afi_base HOST_BIN=host_base
```

Output: The command `make afi` automatically saves FPGA images and host binaries in the S3 bucket `$S3_EXE_BUCKET_NAME`. This process organizes the files in the folder `s3://$S3_EXE_BUCKET_NAME/executables/fpga/` as follows:

- `./fpga_afi/<fpga_bin>`: stores the FPGA images
- `./host_executables/<host_bin>`: stores the host binaries that call the FPGA images

```
\ texttt {\$S3\_EXE\_BUCKET\_NAME}/
  executables /
    fpga/
      fpga_afi /
        afi_base .awsxclbin
        afi_optimized .awsxclbin
      host_executables /
        host
        host_base
```

Remark: Once you are done with the creation of the FPGA images, delete all S3 buckets, except the one named `$S3_EXE_BUCKET_NAME`. For more information on how to delete S3 buckets, follow this [link](#).

Step 2: Execute on an AWS FPGA instance (**f1.2xlarge**)

- **Launch the Instance.** Log into the AWS instance **f1.2xlarge**. To set up the instance, follow the instructions in [documents/FPGA-run.pdf](#).
- **Clone the GitHub repositories.** Open the terminal. Then, clone our GitHub repository into a directory of your preference (e.g., `/home/centos/`):

```
git clone https://github.com/AleP83/FPGA-Econ.git
```

- **Set the AWS credentials.** Configure your AWS credentials by executing the following command in the terminal:

```
aws configure
```

Follow the steps here:

```
$ aws configure
AWS Access Key ID [***** xxxx]: <Your AWS Access Key ID>
AWS Secret Access Key [***** xxxx]: <Your AWS Secret Access Key>
Default region name: us-west-2
Default output format: json
```

For more information visit this [link](#).

- **Modify the Makefile.** Update settings in the `V_accumulator/code/Makefile` as follows:

- **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing `S3-NAME-GOES-HERE`

```
S3_EXE_BUCKET_NAME := S3-NAME-GOES-HERE
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is `us-west-2`):

```
AWS_REGION := us-west-2
```

AWS Region and Bucket name should coincide with the ones used in the synthesis stage.

- **Modify Shell Script for FPGA Results.** Update settings in the `V_accumulator/code/common/util/generate_fpga_results.sh` as follows:

- **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing `S3-NAME-GOES-HERE`

```
S3_EXE_BUCKET_NAME="S3-NAME-GOES-HERE"
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is `us-west-2`):

```
AWS_REGION="us-west-2"
```

AWS Region and Bucket name should coincide with the ones used in the synthesis stage.

- **Initiate `tmux` terminal session.** To ensure your terminal session remains active throughout the execution, initiate a terminal multiplexer session:

```
tmux
```

The `tmux` command allows you to detach and reattach to terminal sessions without interruption. For example, to resume a `tmux` session with index 0, use the following command:

```
tmux attach -t 0
```

For detailed instructions on how to use `tmux`, see this [guide](#).

- **Execute application on an F1 instance.** Navigate to the `/V_accumulator/code` folder, and run the following commands to:

- Copy the executables from AWS S3 folder to the current AWS instance;
- Execute all the relevant exercises
- Transfer the generated results into the S3 folder.

In particular:

1. To execute both FPGA images (baseline and optimized) on the **f1.2xlarge** instance type on the **tmux** terminal:

```
make fpga_results TABLE=ALL USE_AWS_S3_EXE=yes
```

2. To execute only the FPGA image with optimized acceleration:

```
make fpga_results TABLE=OPT USE_AWS_S3_EXE=yes
```

3. To execute only the FPGA image with no optimizations:

```
make fpga_results TABLE=BASE USE_AWS_S3_EXE=yes
```

Output. The command **make fpga_results** automatically saves the results in the S3 bucket **\$S3_EXE_BUCKET_NAME** under the folder **s3://\$S3_EXE_BUCKET_NAME/results/fpga/**:

```
$S3_EXE_BUCKET_NAME/
  results /
    fpga/
      *.txt
      *.csv
      *.run_summary
```

Remark: Make sure to terminate your F1 instance! Even the smaller one (**z1d.2xlarge**) costs 1.65\$/hr.

Step 3: Transfer the results to your local folder

The S3 bucket named **\$S3_EXE_BUCKET_NAME** contains the results of all CPU-C and FPGA-C model estimations. To download these results to your local machine, run the following file after making these changes:

- **Launch the Instance.** Log into an inexpensive AWS instance, say **m5n.large**.
- **Download S3 bucket in AWS instance.** Copy the S3 bucket into a directory of your choice within your AWS instance.

```
aws s3 cp --recursive s3 :// S3_EXE_BUCKET_NAME/ ./s3-bucket/
```

- **Compress the results.** Compress the bucket results using **tar**

```
tar -czvf s3-bucket-$(date +%Y-%m-%d).tar.gz s3-bucket/
```

- **Copy the results in your local machine.** Navigate into your local machine to a directory of your choice and execute the following commands from the terminal:

```
instance_name="35-91-136-136"
key_directory="<Your AWS Access Key ID>"
region="<Your region>"
scp -i "${key_directory}" ec2-user@ec2-$instance_name.$region.compute.amazonaws.com:/home/ec2-user/s3-bucket-*.tar.gz ./
```

Clean AWS account

Once you are done with the AWS estimation, terminate all instances, delete all attached volumes and S3 buckets to avoid unintended charges.

2.4 Header Files

File: `/code/common/definitions.h`

Description: This is the main header files. It defines and initializes essential components such as variables and structures, model and simulation parameters, number of states, the tolerance for convergence, iteration counts, file paths, and more.

Note. The file describes the main structures:

- `preinit_t`: stores the array for which we want to compute the accumulation;
- `out_t`: stores the sum of the array elements;

Note: The user can change here the size of the array `J`.

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.

Note: Users have the flexibility to switch between floating-point and fixed-precision representations by adjusting the `FIXED_ACC` macro.

File: `/code/common/app.h`

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

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 required by the kernel. In particular it declares:

- the kernel function `runOnfpga`;
- the accumulation loop `hw_loop`;
- the initialization of variables in local memory `hw_top_init`.

Files: `/code/cpu/stopwatch.h`

Description: This header file contains the class definition for the stopwatch timer which is used for measuring all latencies.

2.4.1 Accumulator Designs: `hw.cpp`

Users can customize the `hw_loop` function in the `hw.cpp` file to explore the designs discussed in Cheela et al. (2024).

Krusell Smith (1998)

This section describes the FPGA acceleration of the Krusell and Smith (1998) algorithm in Cheela et al. (2024). Code and supplementary materials are available in the `./code` directory of our GitHub repository <https://github.com/AleP83/FPGA-Econ.git>.

3.1 Directory Structure

The directory is structured into four folders. The folder `common` contains host code, supporting libraries, input files, and utility scripts. The folder `fpga` holds the kernel for execution on both FPGA and CPU. Results are stored in the folder `results`. For convenience, the directory `executables` stores the executables for CPU and FPGA acceleration.

1 <code>common</code>	26 <code>idshock.txt</code>
2 <code>app.cpp</code>	27 <code>stopwatch.h</code>
3 <code>app.h</code>	28 <code>util</code>
4 <code>cons.h</code>	29 <code>OpenMPI_install.sh</code>
5 <code>definitions.h</code>	30 <code>generate_fpga_results.sh</code>
6 <code>dev_options.h</code>	31 <code>generate_cpu_results.sh</code>
7 <code>init.cpp</code>	32 <code>input_pack.py</code>
8 <code>init.h</code>	33 <code>make_afi_public.sh</code>
9 <code>libs</code>	34 <code>executables</code>
10 <code>ap_common.h</code>	35 <code>cpu</code>
11 <code>ap_decl.h</code>	36 <code>fpga</code>
12 <code>ap_fixed.h</code>	37 <code>fpga_afi</code>
13 <code>ap_fixed_base.h</code>	38 <code>host_executables</code>
14 <code>ap_fixed_ref.h</code>	39 <code>fpga</code>
15 <code>ap_fixed_special.h</code>	40 <code>design.cfg</code>
16 <code>ap_int.h</code>	41 <code>hls_config.tcl</code>
17 <code>ap_int_base.h</code>	42 <code>hw.cpp</code>
18 <code>ap_int_ref.h</code>	43 <code>hw.h</code>
19 <code>ap_int_special.h</code>	44 <code>results</code>
20 <code>etc</code>	45 <code>cpu</code>
21 <code>xcl2.cpp</code>	46 <code>final_values</code>
22 <code>xcl2.hpp</code>	47 <code>fpga</code>
23 <code>xcl2.mk</code>	48 <code>final_values</code>
24 <code>shocks</code>	49 <code>Makefile</code>
25 <code>agshock.txt</code>	50 <code>xrt.ini</code>

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_to_s3](#),
- Parallel execution on CPU using Open MPI: [make openmpi_to_s3](#),
- Execution on FPGA: [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 CPU/FPGA hardware execution, fetches back the result from the kernel.
- **Kernel.** The [/fpga/hw.cpp](#) contains the Vitis kernel for FPGA and CPU execution.
- **Results.** Results are stored in [/results](#).
- **Header Files.** Header files and helper functions are contained in the following directory
 - [/common](#): : boiler-plate code shared by FPGA and CPU
 - [/common/libs](#): libraries for FPGA software emulation
 - [/fpga](#): kernel files shared by FPGA and CPU
- **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 Compile and Execute on a CPU

These steps describe how to compile, execute, and store the results of the KS algorithm on an [S3](#)-bucket.

3.3.1.1 Step 1: Compile all the binaries

- **Launch the Instance.** Log into the AWS instance [m5n.large](#). To set up and launch the instance, follow the instructions in [documents/CPU-run.pdf](#).
- **Install the Packages.** Initiate a terminal session on the AWS instance and run the subsequent script to install the utilities `git`, `make`, `tmux` and `wget`:

```
sudo yum install git -y
sudo yum install make -y
sudo yum install tmux -y
sudo yum install wget -y
```

- **Clone the GitHub repositories.** Clone our GitHub repository into a directory of your preference (e.g., `/home/ec2-user`):

```
git clone https://github.com/AleP83/FPGA-Econ.git
```

- **Set the AWS Credentials.** Configure your AWS credentials by executing the following command in the terminal:

```
aws configure
```

Follow the steps here:

```
$ aws configure
AWS Access Key ID [***** xxxx]: <Your AWS Access Key ID>
AWS Secret Access Key [***** xxxx]: <Your AWS Secret Access Key>
Default region name: us-west-2
Default output format [None]: json
```

For more information visit this [link](#).

- **Install OpenMPI.** Run the following script from the terminal:

```
sh code/common/util/OpenMPI_install.sh
```

Note: Installing Open-MPI may take some time (10-15 minutes).

- **Set the OpenMPI environment.** If you are compiling or building for parallel execution, execute 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
```

- **Modify the Makefile.** Update settings in the [code/Makefile](#) as follows:

- **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing [S3-NAME-GOES-HERE](#)

```
S3_EXE_BUCKET_NAME := S3-NAME-GOES-HERE
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is **us-west-2**):

```
AWS_REGION := us-west-2
```

- **Modify the Main.** Open **/code/common/app.cpp** and set the number of models **N_MODEL** you want to compute (1,200 in our benchmark specification):

```
#define N_MODEL 1200 // total number of models
```

- **Set the Grid Sizes.** Open **/code/common/definitions.h** and set the grid sizes:

```
#define NKGRID 100 // grid points on individual capital grid
#define NKM_GRID 4 // grid points on aggregate capital grid
```

The benchmark code is set to allocate **NKGRID=100**, **NKM_GRID=4**.

- **Set the Software Design.** Open **/code/common/dev_options.h** and select the interpolation-range search algorithm:

```
// Set only one of the following macros to 1, keeping the rest to zero.
#define _LINEAR_SEARCH 0
#define _BINARY_SEARCH 0
#define _CUSTOM_BINARY_SEARCH 1
```

The benchmark code is set to implement the jump-search algorithm **_CUSTOM_BINARY_SEARCH 1**.

- **Compile the binary.** After modifying the files, navigate to the **code/** directory using the terminal. Then, compile the application for CPU execution using the following command:

- For building binaries for sequential execution on single-core instance:

```
make cpu_to_s3 CPU_EXE=<# Economies>_<# indiv cap.>_<# agg cap.>
```

For example, compile the benchmark model as follows:

```
make cpu_to_s3 CPU_EXE=1200_100k_4km
```

- For building binaries for parallel execution on multi-core instance:

```
make openmpi_to_s3 OPENMPI_EXE=mpi_<# Economies>_<# indiv capital>_<# agg capital>
```

For example, compile the benchmark model as follows:

```
make openmpi_to_s3 OPENMPI_EXE=mpi_1200_100k_4km
```

- **Compile all binaries.** See the accompanying [README .pdf](#) for detailed instructions on how to compile binaries for all of the combinations, $NKGRID \in \{100, 200, 300\}$, $NKM_GRID \in \{4, 8\}$, and search algorithms $\in \{linear, binary, custom_binary\}$, required to replicate the results in the paper.

```
make cpu_to_s3 CPU_EXE=1200_100k_4km
make cpu_to_s3 CPU_EXE=1200_200k_4km
make cpu_to_s3 CPU_EXE=1200_300k_4km
make cpu_to_s3 CPU_EXE=1200_100k_8km
make cpu_to_s3 CPU_EXE=1200_200k_8km
make cpu_to_s3 CPU_EXE=1200_300k_8km
make cpu_to_s3 CPU_EXE=1200_linear
make cpu_to_s3 CPU_EXE=1200_binary
make openmpi_to_s3 OPENMPI_EXE=mpi_1200_100k_4km
```

Output: The `make cpu_to_s3` and `make openmpi_to_s3` commands will save the binaries in your S3 bucket, identified as `$S3_EXE_BUCKET_NAME`, under the folder

`s3://$S3_EXE_BUCKET_NAME/executables/cpu/`:

```
$S3_EXE_BUCKET_NAME/
  executables /
    cpu/
      1200_100k_4km
      1200_200k_4km
      1200_300k_4km
      1200_100k_8km
      1200_200k_8km
      1200_300k_8km
      1200_linear
      1200_binary
      mpi_1200_100k_4km
```

3.3.1.2 Step 2: Execute the binaries on AWS

1. **Launch the Instance.** Log into the appropriate AWS instance: `m5n.large`, `m5n.4xlarge`, or `m5n.24xlarge`. To set up and launch the instance, follow the instructions in [documents/CPU-run.pdf](#).
2. **Install the Packages.** Initiate a terminal session on the AWS instance and run the subsequent script to install the utilities `git`, `make`, `tmux` and `wget`:

```
sudo yum install git -y
sudo yum install make -y
```

```
sudo yum install tmux -y
sudo yum install wget -y
```

3. **Clone the GitHub repositories.** Clone our GitHub repository into a directory of your preference (e.g. /home/ec2-user):

```
git clone https://github.com/AleP83/FPGA-Econ.git
```

4. **Set the AWS credentials.** Configure your AWS credentials by executing the following command in the terminal:

```
aws configure
```

Follow the steps here:

```
$ aws configure
AWS Access Key ID [***** xxxx]: <Your AWS Access Key ID>
AWS Secret Access Key [***** xxxx]: <Your AWS Secret Access Key>
Default region name: us-west-2
Default output format [None]: json
```

For more information visit this [link](#).

5. **Modify the Makefile.** Update settings in the `code/Makefile` as follows:

- **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing `S3-NAME-GOES-HERE`

```
1 S3_EXE_BUCKET_NAME := S3-NAME-GOES-HERE
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is `us-west-2`):

```
1 AWS_REGION := us-west-2
```

6. **Modify Shell Script for CPU Results.** Update settings in the `code/common/util/generate_cpu_results.sh` as follows:

- **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing `S3-NAME-GOES-HERE`

```
1 S3_EXE_BUCKET_NAME="S3-NAME-GOES-HERE"
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is **us-west-2**):

```
1 AWS_REGION="us-west-2"
```

AWS Region and Bucket name should coincide with the ones used in the compiling stage.

7. **Initiate `tmux` terminal session.** To ensure your terminal session remains active throughout the potentially lengthy execution, initiate a terminal multiplexer session:

```
tmux
```

The `tmux` command allows you to detach and reattach to terminal sessions without interruption. For example, to resume a `tmux` session with index 0, use the following command:

```
tmux attach -t 0
```

For detailed instructions on how to use `tmux`, see this [guide](#).

8. **Run all binaries.** To run the binaries on the CPU, navigate to the directory **code/** from within the `tmux` terminal window. Therein, execute the binaries sequentially and copy the generated results to AWS-S3 Bucket with the following AWS instance specific commands:

- To replicate results on the **m5n.large** instance execute on the `tmux` terminal:

```
make cpu_results M5N=1x USE_AWS_S3_EXE=yes
```

Execution time: This step takes about one week (approximately six and a half days). To expedite the process (down to 50 hours), we provide commands to split the workload into three batches. These batches can be executed concurrently on three distinct **m5n.large** instances. This is achieved by replacing **M5N=1x** in the command with **M5N=1xBATCH1**, **M5N=1xBATCH2**, and **M5N=1xBATCH3**. For instance, to initiate the first batch, the following command can be used:

```
make cpu_results M5N=1xBATCH1 USE_AWS_S3_EXE=yes
```

- To replicate results on the **m5n.4xlarge** instance execute on the `tmux` terminal:

```
make cpu_results M5N=4x USE_AWS_S3_EXE=yes
```

Execution time: About one hour.

- To replicate results on the **m5n.24xlarge** instance execute on the `tmux` terminal:

```
make cpu_results M5N=24x USE_AWS_S3_EXE=yes
```

Execution time: About 10 minutes.

Output. The command **make cpu_results** automatically saves the results in your S3 bucket, identified as `$S3_EXE_BUCKET_NAME`, under the folder `s3://$S3_EXE_BUCKET_NAME/results/cpu/`:

```
$S3_EXE_BUCKET_NAME/
  results /
    cpu/
      *. txt
```

3.3.2 Compile and Execute on an FPGA

The following steps describe how to:

1. Synthesize the FPGA image on a build instance, **z1d.2xlarge**;
2. Execute it on the the appropriate AWS instance: **f1.2xlarge**, **f1.4xlarge**, or **f1.16xlarge**;
3. Store the results in the **S3**-bucket named **fpga-econ-ks**.

3.3.2.1 Step 1: Synthesize the FPGA image

- **Launch the Instance.** Log into the AWS build instance: **z1d.2xlarge**. To launch the instance, follow the instructions in [documents/FPGA-design.pdf](#). To set up the instance for development purposes—using for example NICE DCV for analysing the hardware design—follow the instructions in Section [1.2](#).
- **Clone the GitHub repositories..** Open the terminal. Then, clone the AWS repository and our GitHub repository into a directory of your preference (e.g., `/home/centos/`):

```
git clone https://github.com/aws/aws-fpga.git $AWS_FPGA_REPO_DIR
git clone https://github.com/AleP83/FPGA-Econ.git
```

- **Set the AWS credentials.** Configure your AWS credentials by executing the following command in the terminal:

```
aws configure
```

Follow the steps here:

```
$ aws configure
AWS Access Key ID [*****xxxx]: <Your AWS Access Key ID>
AWS Secret Access Key [*****xxxx]: <Your AWS Secret Access Key>
Default region name: us-west-2
Default output format [None]: json
```

For more information visit this [link](#).

- **Modify the Makefile.** Update settings in the [code/Makefile](#) as follows:

- **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing **S3-NAME-GOES-HERE**

```
S3_EXE_BUCKET_NAME := S3-NAME-GOES-HERE
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is **us-west-2**):

```
AWS_REGION := us-west-2
```

Go to the directory **code/**, and modify the following files:

- Modify the Main.** Open **/code/common/app.cpp** and set the number of models **N_MODEL** you want to compute (1,200 in our benchmark specification):

```
#define N_MODEL 1200 // total number of models
```

- Set the Grid Sizes.** Open **/code/common/definitions.h** and set the grid size:

```
#define NKGRID 100 // grid points on individual capital grid
#define NKM_GRID 4 // grid points on aggregate capital grid
```

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

- Set the Hardware Design.** Open **/code/common/dev_options.h** and select the FPGA design:

```
#define _BASELINE 0 // Design with no HLS acceleration.
#define _PIPELINE 0 // Design with only PIPELINE acceleration
#define _WITHIN_ECONOMY 0 // Single-Kernel Design
#define _ACROSS_ECONOMY 1 // Three-kernel Design (Benchmark)
```

These macros select the following FPGA designs: **_BASELINE** selects an FPGA image without optimizations; **_PIPELINE** selects an FPGA image with only pipeline optimization; **_WITHIN_ECONOMY** selects the single-kernel design; and **_ACROSS_ECONOMY** selects the three-kernel design.

- Set the Hardware Design Specs.** Open **/code/fpga/design.cfg** and select the single vs three-kernel design by appropriately commenting out the code you do not need. For example, the listing below executes the three-kernel design by commenting out (using #) the single-kernel design:

```
#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=runOnfpga_1:SLR2
slr=runOnfpga_2:SLR1
slr=runOnfpga_3:SLR0
sp=runOnfpga_1.m_axi_gmem0:DDR[1]
sp=runOnfpga_2.m_axi_gmem0:DDR[0]
sp=runOnfpga_3.m_axi_gmem0:DDR[3]
```

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

1. Execute Software emulation

- **Description.** The main goal of software emulation (sw_emu) is to ensure functional correctness of the host program and kernels (including the debugging of OpenCL instructions). 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
```

Once you are happy with the performance of your FPGA design you can go move to the next step: the synthesis of the FPGA on hardware.

2. **Create all FPGA images.** To ensure your terminal session remains active throughout the potentially lengthy synthesis process, initiate a terminal multiplexer session:

```
tmux
```

The `tmux` command allows you to detach and reattach to terminal sessions without interruption. For example, to resume a `tmux` session with index 0, use the following command:

```
tmux attach -t 0
```

For detailed instructions on how to use `tmux`, see this [guide](#). **Create the FPGA Image: System Hardware Target**

- **Description.** When we set as build target the hardware, HLS `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 on a build instance ([z1d.2xlarge](#)).** To initiate the synthesis of the FPGA circuit, navigate to the directory `code/` from within the `tmux` terminal window. Therein, execute the following instructions to generate the host and fpga target files on the build instance ([z1d.2xlarge](#)); and subsequently, upload the resulting executables to the AWS bucket:

```
make clean
unset XCL_EMULATION_MODE
// 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(s)
make afi FPGA_BIN=3ker_100k_4km HOST_BIN=1200_3ker_100k_4km
```

Important: This command generates the FPGA image for the hardware design defined by modifying the files `app.cpp`, `definitions.h`, `dev_options.h`, and `design.cfg` in steps (i)-(iv) listed above. The naming convention `fpga_bin-host_bin` is used to organize the host-FPGA binaries in the S3-bucket for replicating the results in Cheela et al. (2024) but does not modify the files for you. So, if you modify the individual capital grid size to `NKGRID=200` and execute:

```
make afi FPGA_BIN=3ker_100k_4km HOST_BIN=1200_3ker_100k_4km
```

the compiler will synthesize an image with 200 points on the grid size. Accordingly:

- * For experimentation purpose, use:

```
make afi FPGA_BIN=3ker_100k_4km HOST_BIN=1200_3ker_100k_4km
```

- * For replicating the results in Cheela et al. (2024), use the instructions in [README.pdf](#) to appropriately modify the files ((steps (i)-(iv)) to match the intended design associated with the following naming conventions:

```
make afi FPGA_BIN=3ker_100k_4km HOST_BIN=1200_3ker_100k_4km
make afi FPGA_BIN=1ker_100k_4km HOST_BIN=1200_1ker_100k_4km
make afi FPGA_BIN=1ker_200k_4km HOST_BIN=1200_1ker_200k_4km
make afi FPGA_BIN=1ker_300k_4km HOST_BIN=1200_1ker_300k_4km
make afi FPGA_BIN=1ker_100k_8km HOST_BIN=1200_1ker_100k_8km
make afi FPGA_BIN=1ker_200k_8km HOST_BIN=1200_1ker_200k_8km
make afi FPGA_BIN=1ker_300k_8km HOST_BIN=1200_1ker_300k_8km
make afi FPGA_BIN=baseline_1ker_100k_4km HOST_BIN=120_1ker_100k_4km
make afi FPGA_BIN=pipeline_1ker_100k_4km HOST_BIN=120_1ker_100k_4km
```

Output: The command `make afi` automatically saves FPGA images and host binaries in your S3 bucket, identified as `$S3_EXE_BUCKET_NAME`. This process organizes the files in the folder `s3://$S3_EXE_BUCKET_NAME/executables/fpga/` as follows:

- `./fpga_afi/<fpga_bin>`: stores the FPGA images
- `./host_executables/<host_bin>`: stores the host binaries that call the FPGA images

```
$S3_EXE_BUCKET_NAME/
  executables /
    fpga/
      fpga_afi /
        1ker_100k_4km.awsxcclbin
        1ker_100k_8km.awsxcclbin
        1ker_200k_4km.awsxcclbin
        1ker_200k_8km.awsxcclbin
        1ker_300k_4km.awsxcclbin
        1ker_300k_8km.awsxcclbin
        3ker_100k_4km.awsxcclbin
        baseline_1ker_100k_4km.awsxcclbin
        pipeline_1ker_100k_4km.awsxcclbin
      host_executables /
        120_1ker_100k_4km
        1200_1ker_100k_4km
        1200_1ker_100k_8km
        1200_1ker_200k_4km
        1200_1ker_200k_8km
        1200_1ker_300k_4km
        1200_1ker_300k_8km
        1200_3ker_100k_4km
```

Remark: Once you are done with the creation of the FPGA images, delete all S3 buckets, except for the one you created, `$S3_EXE_BUCKET_NAME`. For more information on how to delete S3 buckets, follow this [link](#).

3.3.2.2 Step 2: Execute on an AWS FPGA instance (f1.2xlarge)

- **Launch the Instance.** Log into the appropriate AWS instance: f1.2xlarge, f1.4xlarge, or f1.16xlarge. To set up the instance, follow the instructions in [documents/FPGA-run.pdf](#).
- **Clone the GitHub repositories.** Open the terminal. Then, clone our GitHub repository into a directory of your preference (e.g., /home/centos/):

```
git clone https://github.com/AleP83/FPGA-Econ.git
```

- **Set the AWS credentials.** Configure your AWS credentials by executing the following command in the terminal:

```
aws configure
```

Follow the steps here:

```
$ aws configure
AWS Access Key ID [***** xxxx]: <Your AWS Access Key ID>
AWS Secret Access Key [***** xxxx]: <Your AWS Secret Access Key>
Default region name: us-west-2
Default output format: json
```

For more information visit this [link](#).

- **Modify the Makefile.** Update settings in the [code/Makefile](#) as follows:
 - **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing **S3-NAME-GOES-HERE**

```
S3_EXE_BUCKET_NAME := S3-NAME-GOES-HERE
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is **us-west-2**):

```
AWS_REGION := us-west-2
```

- **Modify Shell Script for FPGA Results.** Update settings in the [code/common/util/-generate_fpga_results.sh](#) as follows:

- **Set the AWS S3 Bucket Name:** Specify the S3 bucket name by replacing **S3-NAME-GOES-HERE**

```
S3_EXE_BUCKET_NAME="S3-NAME-GOES-HERE"
```

Remark: The S3 bucket name must be globally unique within AWS. If an error occurs during bucket creation, it may be due to the name being already in use by another user.

- **Select the AWS region** of the S3 bucket (default is `us-west-2`):

```
AWS_REGION="us-west-2"
```

AWS Region and Bucket name should coincide with the ones used in the synthesis stage.

- **Initiate `tmux` terminal session.** To ensure your terminal session remains active throughout the execution, initiate a terminal multiplexer session:

```
tmux
```

The `tmux` command allows you to detach and reattach to terminal sessions without interruption. For example, to resume a `tmux` session with index 0, use the following command:

```
tmux attach -t 0
```

For detailed instructions on how to use `tmux`, see this [guide](#).

- **Execute application on an F1 instance.** Navigate to the `code/` folder, and run the following commands to:
 - * Copy the executables from AWS S3 folder to the current AWS instance;
 - * Execute all the relevant exercises
 - * Transfer the generated results into the S3 folder.

To experiment with a custom FPGA image execute on the `tmux` terminal:

```
make fpga_results TABLE=3 USE_AWS_S3_EXE=yes
```

This will automatically select the FPGA image named `3ker_100k_4km` and the host code named `1200_3ker_100k_4km` previously generated. As mentioned above, this naming does not reflect the hardware design you actually selected by modifying the files `app.cpp`, `definitions.h`, `dev_options.h`, and `design.cfg` in steps (i)-(iv) listed above. For replicatin the results in Cheela et al. (2024) see our accompanying [README.pdf](#) file.

Output. The command `make fpga_results` automatically saves the results in your S3 bucket, identified as `$S3_EXE_BUCKET_NAME`, under the folder `s3://$S3_EXE_BUCKET_NAME/results/fpga/`:

```
$S3_EXE_BUCKET_NAME/
  results/
    fpga/
      *.txt
      *.csv
      *.run_summary
      *.rpt
```

```
*.txt
*.log
```

Remark: Make sure to terminate your F1 instance! Even the smaller one (**z1d.2xlarge**) costs 1.65\$/hr.

3.3.2.3 Step 3: Transfer the results to your local folder

The S3 bucket named `$S3_EXE_BUCKET_NAME` contains the results of all CPU-C and FPGA-C model estimations. To download these results to your local machine, run the following file after making these changes:

- **Launch the Instance.** Log into an inexpensive AWS instance, say **m5n.large**.
- **Download S3 bucket in AWS instance.** Copy the S3 bucket into a directory of your choice within your AWS instance.

```
aws s3 cp --recursive s3://fpga-econ-ks/ ./s3-bucket/
```

- **Compress the results.** Compress the bucket results using `tar`

```
tar -czvf s3-bucket-$(date +%Y-%m-%d).tar.gz s3-bucket/
```

- **Copy the results in your local machine.** Navigate into your local machine to a directory of your choice and execute the following commands from the terminal:

```
instance_name="35-91-136-136"
key_directory="<Your AWS Access Key ID>"
region="<Your region>"
scp -i "${key_directory}" ec2-user@ec2-$instance_name.$region.compute.amazonaws.com:/
home/ec2-user/s3-bucket-*.tar.gz ./
```

3.3.2.4 Clean AWS account

Once you are done with the AWS estimation, terminate all instances, delete all attached volumes and S3 buckets to avoid unintended charges.

3.4 Header Files

File: `/code/common/definitions.h`

Description: This is the main header files. It defines and initializes essential components such as variables and structures, model and simulation parameters, number of states, the tolerance for convergence, iteration counts, file paths, and more.

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;
- `vars_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, r_2 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 CPU and 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/common/init.h`

Description: This header file declares the functions used in `init.cpp`

Files: `/code/common/stopwatch.h`

Description: This header file contains the class definition for the stopwatch timer which is used for measuring all latencies.

3.5 Boiler-plate code: app.cpp

The file `/common/app.cpp` is the main file, containing all boiler-plate code required to communicate with CPU and FPGA. 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`)

```
1 #ifdef _OPENMPI_MODE
2     #define OMPI_MODE 1 // 1 ON, 0 OFF
3 #elif _FPGA_MODE
4     #define FPGA_MODE 1 // 1 ON, 0 OFF
5 #elif _SERIAL_CPU_MODE
6     #define SERIAL_CPU_MODE 1 // 1 ON, 0 OFF
7 #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 must initialize the standard OpenCL structures: target platform, devices, context, command queue, and program. *Note:* While users have the option to follow the native OpenCL C API, this tutorial leverages the OpenCL C++ wrapper API, which is supported by XRT and utilized in many of the [Vitis Examples](#). For additional details about this C++ wrapper API, please consult the following [link](#). For the CPU implementation, we exclusively utilize the C programming language, except for the object-oriented class defined in the `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.

```
456 cl_int err = CL_SUCCESS;
```

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

```
457 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.

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

and count them.

```
462 auto device_count = devices.size();
463 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.

```
468 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.

```
469 vector<cl::Program> programs(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 `NUM_KERNELS` for each device based on the config knob settings in *dev_options.h*

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

Create one command queue vector for each of the FPGA devices

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

Attribute device name to each FPGA device

```
472 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.

```
474 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*.

```
475 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

```
479 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.

```
482 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

```
484 const size_t hw_preinit_size_bytes = sizeof( preinit_t );
485 const size_t hw_out_size_bytes = sizeof( out_t );
486 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.

```
492 vector<vector< preinit_t > > hw_preinit(NUM_DEVICES, vector<preinit_t>(NUM_KERNELS));
493 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.

```

494 vector<vector<vector<int, aligned_allocator<int>>>> hw_iter(NUM_DEVICES, vector<vector<int, aligned_allocator<int>>>>(
    NUM_KERNELS, vector<int, aligned_allocator<int>>>(hw_iter_size)));

```

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.

```

497 vector<vector<cl::Buffer>> buffer_agshock(device_count, vector<cl::Buffer>(NUM_KERNELS));
498 vector<vector<cl::Buffer>> buffer_idshock(device_count, vector<cl::Buffer>(NUM_KERNELS));
499 vector<vector<cl::Buffer>> buffer_preinit(device_count, vector<cl::Buffer>(NUM_KERNELS));
500 vector<vector<cl::Buffer>> buffer_out(device_count, vector<cl::Buffer>(NUM_KERNELS));
501 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.

```

505 vector<vector<vector<cl::Event>>> memory_read_events(NUM_DEVICES, vector<vector<cl::Event>>(NUM_KERNELS, std::
    vector<cl::Event>(1)));
506 vector<vector<vector<cl::Event>>> task_events(NUM_DEVICES, vector<vector<cl::Event>>(NUM_KERNELS, std::vector<cl::
    Event>(1)));
507 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.

```

518 OCL_CHECK(err, contexts[d] = cl::Context(devices[d], props, nullptr, nullptr, &err));
519 OCL_CHECK(err, queues[d] = cl::CommandQueue(contexts[d], devices[d], CL_QUEUE_PROFILING_ENABLE |
    CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err));
520 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.

```

523 fileBuf[d] = xcl::read_binary_file(binaryFile);
524 bins[d].push_back({fileBuf[d].data(), fileBuf[d].size()});
525 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.

```

527 for (int k = 0; k < NUM_KERNELS; k++) {
528     if ( k%5 == 0 ){
529         OCL_CHECK(err, kernels[d][k] = cl :: Kernel(programs[d], "runOnfpga:{runOnfpga_1}", &err));
530     }
531     if ( k%5 == 1 ){
532         OCL_CHECK(err, kernels[d][k] = cl :: Kernel(programs[d], "runOnfpga:{runOnfpga_2}", &err));
533     }
534     if ( k%5 == 2 ){
535         OCL_CHECK(err, kernels[d][k] = cl :: Kernel(programs[d], "runOnfpga:{runOnfpga_3}", &err));
536     }
537 }

```

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:

```

539 std::cout << "Creating Buffers[" << d << "]" [" << k << "]" << std::endl;
540 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));
541 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));
542 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));
543 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));
544 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.

```

550 for (int d = 0; d < NUM_DEVICES; d++) {
551     for (int k = 0; k < NUM_KERNELS; k++) {
552         OCL_CHECK(err, kernels[d][k].setArg(0, buffer_agshock[d][k]));
553         OCL_CHECK(err, kernels[d][k].setArg(1, buffer_idshock[d][k]));
554         OCL_CHECK(err, kernels[d][k].setArg(2, buffer_preinit[d][k]));
555         OCL_CHECK(err, kernels[d][k].setArg(3, buffer_out[d][k]));
556         OCL_CHECK(err, kernels[d][k].setArg(4, buffer_hw_iter[d][k]));
557         std::cout << "Completed Setting Arguments" << std::endl;
558         agshock_ptr[d][k] = (unsigned char *) queues[d].enqueueMapBuffer(buffer_agshock[d][k], CL_TRUE, CL_MAP_WRITE, 0,
            AGSHOCK_ARR_SIZE);
559         idshock_ptr[d][k] = (unsigned char *) queues[d].enqueueMapBuffer(buffer_idshock[d][k], CL_TRUE, CL_MAP_WRITE, 0,
            IDSHOCK_ARR_SIZE);
560     }
561 }

```

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.

```
572 env_t env[NUM_DEVICES][NUM_KERNELS];
573 input_t in[NUM_DEVICES][NUM_KERNELS];
574 vars_t vars[NUM_DEVICES][NUM_KERNELS];
```

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

```
584 init_all (&env[d][k], &in[d][k], &vars[d][k]);
585
586 for (int i=0; i<NSTATES; i++){
587     hw_preinit[d][k].kprime[i] = vars[d][k].kprime_a[i];
588 }
589
590 for (int i=0; i<NSTATES; i++){
591     hw_preinit[d][k].wealth[i] = env[d][k].wealth[i];
592 }

```

```
602 memcpy(agshock_ptr[d][k], in[d][k].agshock, AGSHOCK_ARR_SIZE);
603 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](#).

```
615 printf("Migrating buffers to kernel\n");
616 if (i == 0){
617     OCL_CHECK(err,
618         err = queues[d].enqueueMigrateMemObjects( {
619             buffer_agshock[d][k], buffer_idshock[d][k], buffer_preinit[d][k] },
620             0 /* 0 means from host*/, nullptr, &memory_read_events[d][k][0]));
621 }
622 else {
623     OCL_CHECK(err,
624         err = queues[d].enqueueMigrateMemObjects( {
625             buffer_agshock[d][k], buffer_idshock[d][k], buffer_preinit[d][k] },
626             0 /* 0 means from host*/, &memory_write_events[d][k], &memory_read_events[d][k][0]));
627 }
```

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.

```
637 printf("Enqueing Task\n");
638 OCL_CHECK(err,
639 err = queues[d].enqueueTask(kernels[d][k], &memory_read_events[d][k],
640 &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.

```
650 printf("Migrating buffers from kernel\n");
651 OCL_CHECK(err,
652 err = queues[d].enqueueMigrateMemObjects( {buffer_out[d][k], buffer_hw_iter[d][k]},
653 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.

```
665 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.

```

679 for (int d = 0; d < NUM_DEVICES; d++) {
680     for (int k=0; k < NUM_KERNELS; k++){
681
682         FILE * cfile ;
683         char FileName[512];
684         printf ("Migrating buffers from kernel\n"); //add kgrid, km grid to file names
685         sprintf (FileName, "%sfpga_nkM%d-nk%d_i%d_d%d_k%d.txt", KP_OUT_FILE, NKM_GRID, NKGRID, i, d, k);
686         cfile = fopen(FileName, "w");
687         for (int i=0; i<NSTATES; i++){
688             fprintf ( cfile , "%15lf \n", hw_out[d][k].kprime[i] );
689         }
690         fclose ( cfile );
691     .
692     .
693     .
694 }
695 }

```

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

```

728 for (int d=0; d<NUM_DEVICES; d++){
729     for (int k = 0; k < NUM_KERNELS; k++) {
730         printf ("i=%d d=%d k=%d Bad Coeff 0: %15lf\n", i, d, k, hw_out[d][k].coeff [0]);
731         printf ("i=%d d=%d k=%d Bad Coeff 1: %15lf\n", i, d, k, hw_out[d][k].coeff [1]);
732         printf ("i=%d d=%d k=%d Bad R2: %15lf\n", i, d, k, hw_out[d][k].r2 [0]);
733         printf ("i=%d d=%d k=%d Good Coeff 0: %15lf\n", i, d, k, hw_out[d][k].coeff [2]);
734         printf ("i=%d d=%d k=%d Good Coeff 1: %15lf\n", i, d, k, hw_out[d][k].coeff [3]);
735         printf ("i=%d d=%d k=%d Good R2: %15lf\n", i, d, k, hw_out[d][k].r2 [1]);
736         printf ("i=%d d=%d k=%d Total EGM iter: %d\n", i, d, k, total_egm_iter [d][k]);
737         printf ("i=%d d=%d k=%d Total Main loop iter : %d\n", i, d, k, hw_iter[d][k][0]);
738     }
739 }

```

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 be 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.

```

744 for (int d=0; d<NUM_DEVICES; d++){
745     for (int k = 0; k < NUM_KERNELS; k++) {
746         free_all (&in[d][k]);
747     }
748 }

```

3.5.10 Open MPI

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

Begin by initializing the MPI environment.

```

64 mpi_enabled = MPI_Init(NULL, NULL);

```

Collect the number of processes (available cores).

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

Collect the rank of the processes.

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

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

```
91 MPI_Barrier(MPI_COMM_WORLD);
```

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

```
93 // Range of tasks per processor .
94 int i_min_task_id, i_max_task_id;
95
96 // Define the Block to be assigned to each task
97 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.

```
107 for(int i = i_min_task_id; i <= i_max_task_id; i++) {
108 .
109 .
110 .
111 env_t env;
112 input_t in;
113 vars_t vars;
114 out_t out;
115 int hw_iter[500];
116
117 init_all (&env, &in, &vars);
118 .
119 .
120 .
121 runOnFpga(in.agshock, in.idshock, &hw_preinit, &out, hw_iter);
122 }
```

Save the results of each of the computed model.

```
155 FILE * cfile ;
156 char FileName[512];
157 printf ("Migrating buffers from kernel\n");
158 sprintf (FileName, "%scpu-core-%d_of_%d_nKM%d-nk%d.txt", OPENMPI_KP_OUT_FILE, id_task, n_tasks, NKM_GRID, NKGRID);
159 cfile = fopen(FileName, "w");
160 for(int i=0; i<NSTATES; i++){
161     fprintf ( cfile , "%.15lf \n", out.kprime[i]);
162 }
163 fclose ( cfile );
164 .
165 .
166 .
```

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

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

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

```
218 MPI_Finalize();
```

3.6 Kernel: hw.cpp

The file `/common/hw.cpp` contains the design of the kernel. The kernel defined here is common for the CPU and the FPGA. The FPGA-specific `#PRAGMAS` will be ignored in the CPU case during compilation.

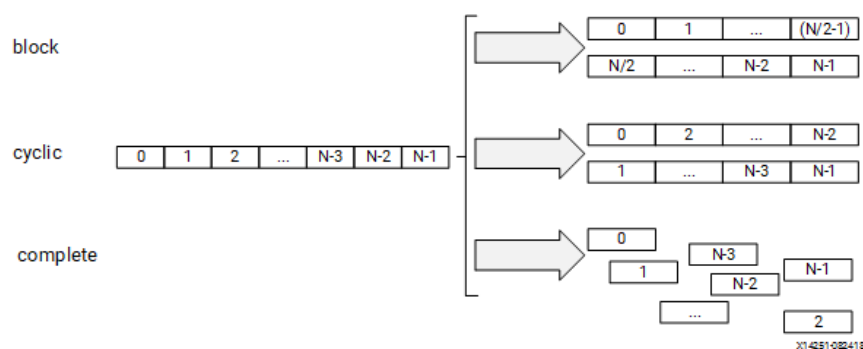
3.6.1 Common HLS Optimization Pragmas

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

3.6.1.1 `#pragma HLS ARRAY_PARTITION`

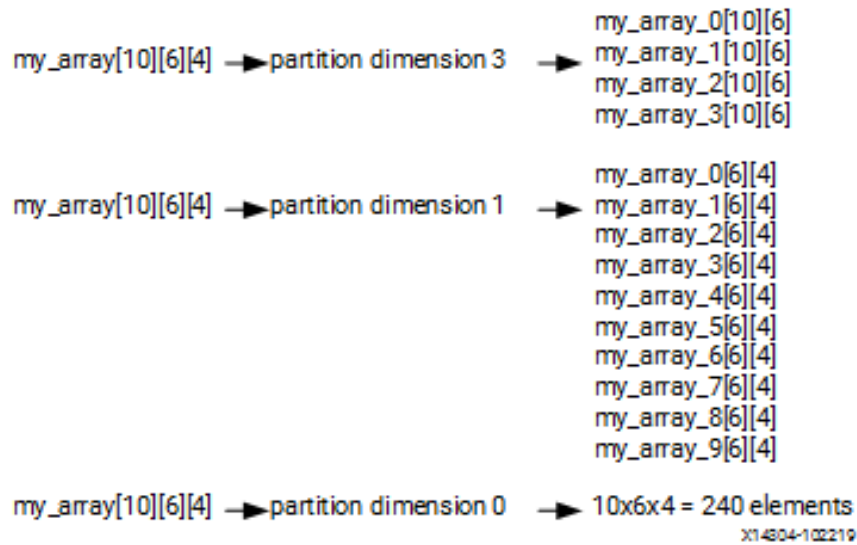
Each memory block (BRAM, URAM) consists of a limited number of memory ports to read or write from the memory. For example a BRAM block usually consist of 2 ports. When data is stored in a BRAM in a contiguous manner, we can only read a maximum of 2 elements in the same clock cycle for a dual port BRAM 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 compiler 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](#).

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](#).

3.6.1.2 `#pragma HLS UNROLL`

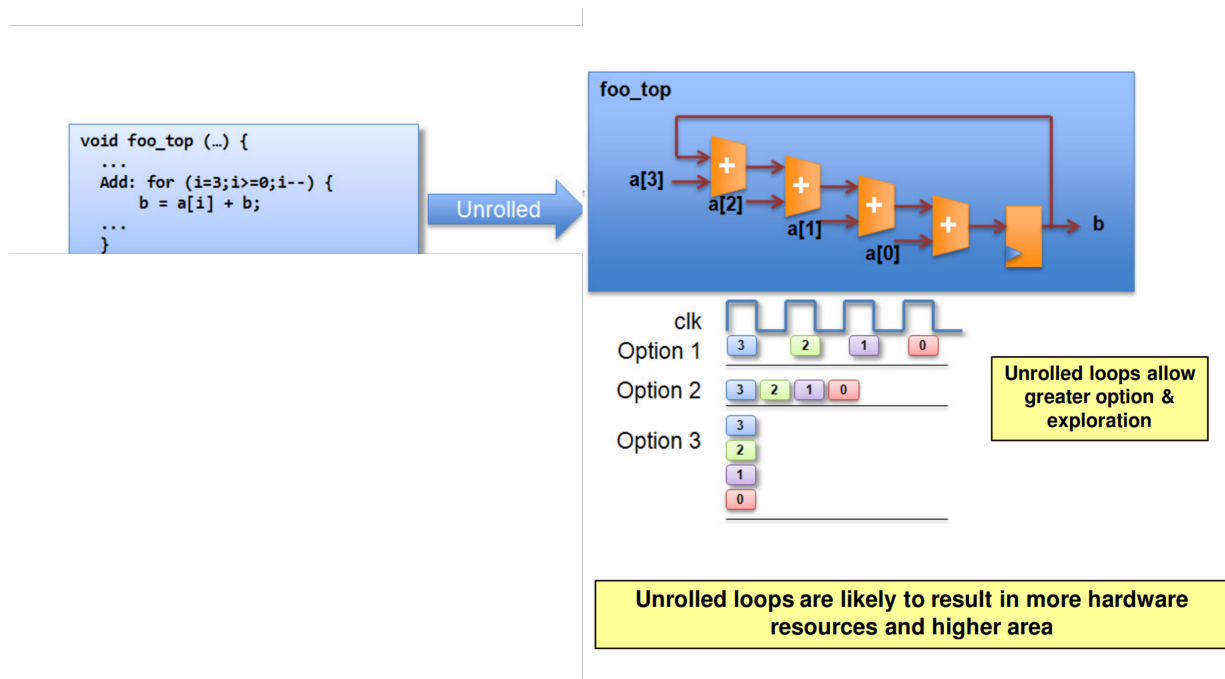
In order to make use of the fpga resources, the designer can spatially unroll loops to create multiple independent operations rather than a single collection of operations. The `#pragma HLS UNROLL` transforms loops by creating in hardware multiple copies of a loop body such that they can all occur in 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.

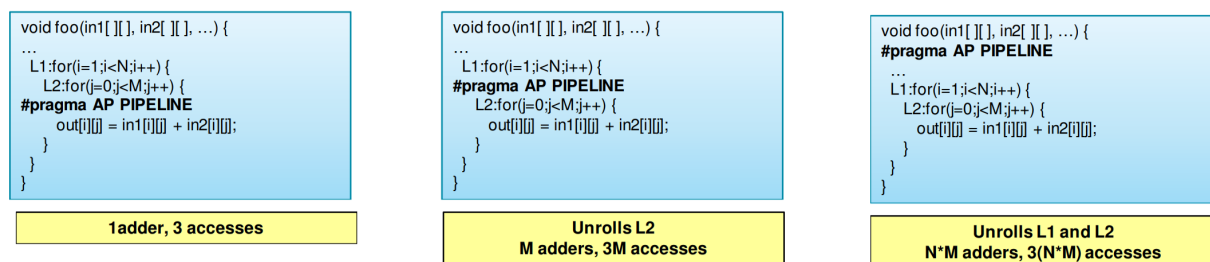
The Figure 3.4 shows a case where placing the pipeline pragma at different loop locations results in 3 different unrolling of the inner loops along with the increased hardware resources and memory accesses. The user needs to make a conscious choice about the placement of the pipeline pragma. If the data 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.

Figure 3.3: Impact of various factors of loop Unrolling

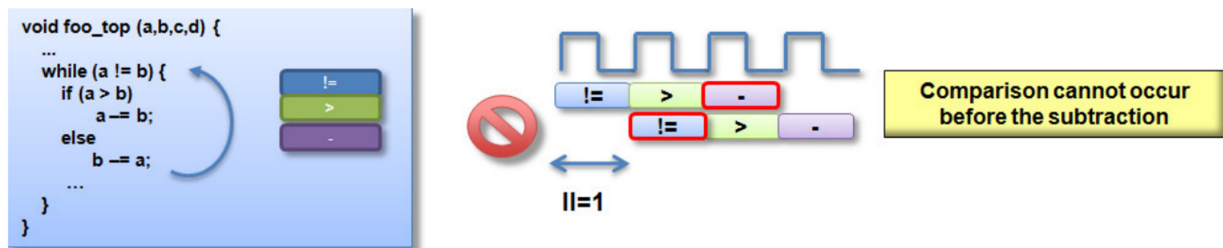


Note: This figure shows how the unrolling by different factors decreasing the overall latency of the loop while increasing the hardware resources.

Figure 3.4: Impact of Pipeline pragma at different levels



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 3.5: Data dependency preventing $\Pi=1$ 

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

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 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.

```

1 void foo (num_samples, ...) {
2     int i;
3     ...
4     loop_1: for (i=0; i< num_samples; i++) {
5         #pragma HLS loop_tripcount min=12 max=16
6         ...
7         result = a + b;
8     }
9 }
```

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

In this chapter, we explain the code snippets corresponding to the FPGA, the CPU execution is a very similar version of it. You can refer to the github repository for the exact line numbers that correspond to the CPU. 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_n8`, `hw_findrange_n100`, `hw_findrange_n200`, `hw_findrange_n300`, `regression`, `RSqauredCalc`.

3.6.3 Parent Kernel Function: runOnfpga

`runOnfpga` is the parent kernel function which:

1. acts as a interface between host (CPU) and device (Either CPU or FPGA)
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.

```

97 void runOnfpga(
98     const unsigned char *hw_agshock,
99     const unsigned char *hw_idshock,
100     preinit_t *preinit ,
101     out_t *results ,
102     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.

```

11 /** Static on-PL memories */
12 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 file `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.

```

110 unsigned char agshock[AGSHOCK_ARR_SIZE];
111 #if PACK_IDS
112     ap_uint<72> idshock[NEW_IDSHOCK_SIZE] = {0};
113 #else
114     unsigned char idshock[IDSHOCK_ARR_SIZE] = {0};
115 #endif
116
117 real st_kcross[N_AGENTS];
118 real st_kprimes[NUM_KPRIMES][NSTATES];
119 real kmts[SIM_STEPS];

```

```

120 real r2[NSTATES_AG];
121 real kmprime[NSTATES_AG * NKM_GRID];
122 real coeff[NCOEFF] = {0, 1, 0, 1};
123 real metric_coeff = 1000; // some large number

125 #if PACK_IDS
126 #pragma HLS bind_storage variable = idshock type = RAM_1P impl = URAM
127 #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. Note that for all the arrays, the size needs to be specified for it to be synthesised.

```

128 #pragma HLS array_partition variable = kmprime complete
129 #pragma HLS array_partition variable = k_grid complete dim = 1
130 #pragma HLS array_partition variable = km_grid 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. To perform this step efficiently, 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.

```

134 for (int i = 0; i < AGSHOCK_ARR_SIZE; i++)
135 {
136     agshock[i] = hw_agshock[i];
137 }

```

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.

```

139 #if PACK_IDS
140 // use URAM to store the idshocks
141 // 8 idshocks are packed into 1 byte-> (1,100 * 10,000 / 8) = 1,375,000 bytes
142 // copy to data variable of size 64 bits . Hence, 8 input bytes are copied to one element
143 main_2: // loop over each of the 1,100 time step. (10,000 / 8) = 1250
144 for (int i = 0, j = 0; i < IDSHOCK_ARR_SIZE; i = i + 1250)
145 {
146 main_2_2: // for each time step, copy 8 bytes into a single element of size 64 bits
147 for (int k = 0; k < 1250; j++)
148 {
149 // handle edge case where last 2 bytes are remaining since 1,250 is not divisible by 8
150 if (k == 1248)
151 {
152 idshock[j] = (hw_idshock[i + k + 1] << 8) | (hw_idshock[i + k]);
153 k = k + 2;
154 }
155 else
156 {
157 idshock[j] = (((ap_uint<72>)hw_idshock[i + k + 7] << 56) |
158 ((ap_uint<72>)hw_idshock[i + k + 6] << 48) |
159 ((ap_uint<72>)hw_idshock[i + k + 5] << 40) |
160 ((ap_uint<72>)hw_idshock[i + k + 4] << 32) |
161 ((ap_uint<72>)hw_idshock[i + k + 3] << 24) |
162 ((ap_uint<72>)hw_idshock[i + k + 2] << 16) |
163 ((ap_uint<72>)hw_idshock[i + k + 1] << 8) |
164 ((ap_uint<72>)hw_idshock[i + k + 0]));
165 k = k + 8;
166 }
167 }
168 }
169
170 #else
171 // use BRAM to store the idshocks
172 main_2:
173 for (int i = 0; i < IDSHOCK_ARR_SIZE; i++)
174 {
175 idshock[i] = hw_idshock[i];
176 }
177 #endif

```

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

```

180 hw_top_init(st_kprimes, st_kcross);

```

The objects **kprime** and **kcross** are burst copied from the global memory.

```

16 void hw_top_init(
17     real st_kprimes[NUM_KPRIMES][NSTATES], real st_kcross[N_AGENTS]
18 #ifndef _FPGA_MODE
19     , preinit_t preinit
20 #endif
21 )
22 {
23     init_1 :
24     for (int j = 0; j < NSTATES; ++j)
25     {
26 #ifdef _FPGA_MODE

```

```

27     real val = kp_in[j];
28 #else
29     real val = preinit.kprime[j];
30 #endif
31     for (int k = 0; k < NUM_KPRIMES; ++k)
32     {
33         st_kprimes[k][j] = val;
34     }
35 }
36
37 init_2 :
38 for (int j = 0; j < N_AGENTS; ++j)
39 {
40     st_kcross[j] = env__kss;
41 }
42
43     .

```

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.

```

65 hw_init_env();
66
67 st_env.irate_factor[0] = 0.3564000000000000;
68 st_env.irate_factor[1] = 0.3636000000000000;
69
70 st_env.wage_factor[0] = 0.6336000000000000;
71 st_env.wage_factor[1] = 0.6464000000000000;
72
73 st_env.cons2_factor[0] = 0.1500000000000000;
74 st_env.cons2_factor[1] = 1.0944444444444445;
75 st_env.cons2_factor[2] = 0.1500000000000000;
76 st_env.cons2_factor[3] = 1.1048611111111111;
77     .
78     .
79 return;

```

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

```

1027 void hw_init_env()
1028 {
1029 #pragma HLS inline
1030 st_env.alpha = env__alpha;    // 0.36 (Output capital share)
1031 st_env.beta = env__beta;      // 0.99 (Quarterly subjective discount factor)
1032 st_env.delta = env__delta;    // 0.025 (Quarterly depreciation rate)
1033 st_env.mu = env__mu;          // 0.15 (Unemployment benefits in terms of wages)
1034 st_env.l_bar = env__l_bar;    //
1035 st_env.delta_a = env__delta_a; // 0.01
1036
1037 st_env.l_bar_inv = env__l_bar_inv; // 0.9 (Time endowment)?
1038 st_env.gamma_inv = env__gamma_inv;
1039 st_env.gamma_neg = env__gamma_neg;
1040 st_env.gamma_neg_inv = env__gamma_neg_inv;
1041
1042 st_env.epsilon_u = env__epsilon_u;

```

```

1043 st_env.epsilon_e = env__epsilon_e;
1044
1045 st_env.ur[0] = env__ur_0;
1046 st_env.er[0] = (1 - st_env.ur[0]);
1047 st_env.ur[1] = env__ur_1;
1048 st_env.er[1] = (1 - st_env.ur[1]);
1049
1050 st_env.er_inv[0] = 1 / st_env.er[0];
1051 st_env.er_inv[1] = 1 / st_env.er[1];
1052
1053 // st_env.kss = hw_pow((1./st_env.beta-(1.-st_env.delta))/st_env.alpha,1./st_env.alpha-1));
1054 st_env.kss = env__kss;
1055
1056 // transition
1057 st_env.P[0] = 0.525;
1058 st_env.P[1] = 0.35;
1059 st_env.P[2] = 0.03125;
1060 st_env.P[3] = 0.09375;
1061 st_env.P[4] = 0.038889;
1062 st_env.P[5] = 0.836111;
1063 st_env.P[6] = 0.002083;
1064 st_env.P[7] = 0.122917;
1065 st_env.P[8] = 0.09375;
1066 st_env.P[9] = 0.03125;
1067 st_env.P[10] = 0.291667;
1068 st_env.P[11] = 0.583333;
1069 st_env.P[12] = 0.009115;
1070 st_env.P[13] = 0.115885;
1071 st_env.P[14] = 0.024306;
1072 st_env.P[15] = 0.850694;
1073
1074 // parms shocks
1075 st_env.epsilon[0] = st_env.epsilon_u;
1076 st_env.epsilon[1] = st_env.epsilon_e;
1077 #if AST_UNROLL
1078 for (int k = 0; k < NUM_KCROSS; ++k)
1079 {
1080 #pragma HLS pipeline off
1081 st_env.epsilon2[k][0] = 0;
1082 st_env.epsilon2[k][1] = 1;
1083 }
1084 #else
1085 st_env.epsilon2[0] = 0;
1086 st_env.epsilon2[1] = 1;
1087 #endif
1088
1089 st_env.ag[0] = 1 - st_env.delta_a;
1090 st_env.ag[1] = 1 + st_env.delta_a;
1091 st_env.ag2[0] = 0;
1092 st_env.ag2[1] = 1;
1093 return;
1094 }

```

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.

```
190 int hw_main_iter = 0; // total number of ihp calls
191 int curr_ihp_iter = 0; // number of ihp iterations in each ihp call
192 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.

```
198 while ( metric_coeff > TOLL_COEFF)
199 {
200     hw_main_iter++;
201     hw_sim_alm(kmprime, coeff); // step 1
202
203     curr_ihp_iter = 0;
204     hw_sim_ihp(st_kprimes, kmprime, curr_ihp_iter); // step 2
205     hw_ihp_iter[hw_main_iter] = curr_ihp_iter; // start from 1st element of hw_ihp_iter
206
207     real kcross_l[N_AGENTS];
208     kc_t kcross_mean = 0;
209
210     ast_kcross :
211     for (int is = 0; is < N_AGENTS; is++)
212     {
213         #if KCROSS_PIP_OFF
214             #pragma HLS pipeline off
215         #endif
216         kcross_l[is] = st_kcross[is];
217         kcross_mean += (kc_t) st_kcross[is];
218     }
219
220     hw_sim_ast(kmts, st_kprimes, kcross_l, agshock, idshock, kcross_mean); // step 3
221
222     sim_alm_coeff(kmts, coeff, &metric_coeff, r2, agshock); // step 4
223
224     if ( metric_coeff > TOLL_COEFF * 100)
225     {
226         // Replace the old with new capital distribution
227         for (int j = 0; j < N_AGENTS; j++)
228         {
229             st_kcross[j] = kcross_l[j];
230         }
231     }
232
233     #if PRINT_LOOP_CNT
234         iter_main++;
235         printf("main loop iter = %d\n", iter_main);
```

```

236 #endif
237 }

```

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.

```

275 void hw_sim_alm(real *kmprime, real *coeff)
276 {
277     small_idx_t cidx = 0;
278     real c0, c1;
279     small_idx_t kidx = 0;
280
281     alm_1:
282     for (int ia = 0; ia < NSTATES_AG; ++ia)
283     {
284         c0 = coeff[cidx];
285         c1 = coeff[cidx + 1];
286         cidx += REGRESSORS;
287     alm_2:
288         for (int ikm = 0; ikm < NKM_GRID; ++ikm)
289         {
290             #pragma HLS unroll factor = 1
291             // #pragma HLS pipeline
292             // #endif
293             // real val = hw_exp(c0
294             // + c1
295             // * st_env.log_env_km[ikm]);
296             #if defined(_SERIAL_CPU_MODE) || defined(_OPENMPI_MODE)
297                 real t_log = hw_log(st_env.km[ikm]);
298             #else
299                 real t_log = hw_log(km_grid[ikm]);
300             #endif
301             real t_mul = c1 * t_log;
302             real t_add = c0 + t_mul;
303             real val = hw_exp(t_add); // hw_exp(c0 + c1 * st_env.log_env_km[ikm])
304             hw_rail_values(&val, KM_MAX, KM_MIN); // eq 15
305             kmprime[kidx++] = val;
306         }
307     }
308     return;
309 }

```

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

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

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.

```

314 #if (NUM_KPRIMES == 8)
315 // #if (NUM_KPRIMES == 8 && _WITHIN_ECONOMY)
316 #pragma HLS array_partition variable = st_env.P complete
317 #pragma HLS array_partition variable = st_kprimes complete dim = 1
318 #pragma HLS bind_storage variable = st_kprimes type = RAM_1WNR impl = BRAM
319 #else
320 #pragma HLS array_partition variable = st_kprimes complete dim = 1
321 #pragma HLS bind_storage variable = st_kprimes type = RAM_2P impl = BRAM
322 #endif
323
324 #if AST_UNROLL
325 #pragma HLS array_partition variable = st_env.epsilon2 complete dim = 1
326 #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.

```

328 /* Lookup tables */
329 // substitute for IXV call
330 static const small_idx_t li_2d_aux_idx_base[4] = {
331     0,
332     NKGRID,
333     NKM_GRID * NSTATES_ID * NKGRID,
334     (NKGRID + NKM_GRID * NSTATES_ID * NKGRID)};
335
336 #pragma HLS array_partition variable = li_2d_aux_idx_base complete

```

```

337
338   real kprime_new[NSTATES]; // Local kprime/new copies
339   real metric = 1;
340   #if PRINT_LOOP_CNT
341     unsigned int iter_cnt = 0;
342   #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, $k'(k, \epsilon, m, A) : \mathbf{K} \times \{0, 1\}_\epsilon \times \mathbf{M} \times \mathbf{A} \rightarrow \mathbb{R}_+$. **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.

```

345 // Convergence loop: 4 x NSTATES interp over kprime[]
346 ihp_1:
347   while (metric > (real)TOLL_K) // eq 14
348   {
349     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 3.6.3.1. Hence, we use the `#pragma HLS LOOP_TRIPCOUNT` to inform the compiler about the maximum number of iterations.

```

349 #pragma HLS loop_tripcount min = 1 avg = 200 max = 2000

```

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

```

351 #if (IHP2_UNROLL==1)
352   spread_t global_spread_scalar = VERY_SMALL_SCALAR;
353   spread_t spread_scalar [2] = {VERY_SMALL_SCALAR, VERY_SMALL_SCALAR};
354 #else
355   spread_t spread_scalar = VERY_SMALL_SCALAR;
356 #endif
357
358 // Reset index values for [1600] loop
359 small_idx_t p_idx_outer = 0b0100; // 4
360 small_idx_t hundreds_cnt = NKGRID;
361 small_idx_t kp_iter_cnt = (NSTATES_ID * NKGRID);
362 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

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

3.6.5.3 IHP Iteration Step.

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

```

363  ihp_2:
364      for (small_idx_t is = 0; is < NSTATES; ++is)
365      {
366      #if SMALL_PL
367      #pragma HLS unroll factor = 1
368      #elif (IHP2_UNROLL==1)
369      #pragma HLS unroll factor = 2
370      #pragma HLS pipeline
371      #else
372      #pragma HLS pipeline
373      #endif

```

takes as given:

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

and uses Equation (3.1) to update the guess

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

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

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

1. Index Handling (Technical).

```

375  pidx_t p_idx_inner = 0; // IIDP x IAP
376  real kmp, temp_base;
377  emu_s_t emu_s = 0.;
378  real kp = st_kprimes[0][ is ];
379
380  // Index handling
381  if (++kp_iter_cnt >= NSTATES_ID * NKGRID)
382  {
383      kp_iter_cnt = 0;
384      kmp = kmprime[kidx++];
385      temp_base = kmp * (real)env__l_bar_inv;

```

```

386 }
387 if (++hundreds_cnt >= NKGRID)
388 {
389     hundreds_cnt = 0;
390     // (changes between 0 and 4 for every 100 iterations upto is = 800,
391     // and changes between 8 and 12 for every 100 iterations upto is = 1600)
392     p_idx_outer ^= (pid_x_t)0b0100; // (XOR at every bit) 0100 ^ 0100 = 0000 -> 0 (explicit conversion to short)
393     // decimal value
394 }
395 if (is == (NKM_GRID * NSTATES_ID * NKGRID)) // 800 ia
396     p_idx_outer |= (pid_x_t)0b1000; // (OR at every bit) 0000 | 1000 = 1000 -> 8 (explicit conversion to short)
397     // decimal value
398     p_idx_inner = 0;

```

2. Compute the conditional expectation emu_s :

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

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:

```

398 ihp_3:
399 for (int iap = 0; iap < NSTATES_AG; ++iap)
400 {
401     #if SMALL_PL
402     #pragma HLS unroll factor = 1
403     #endif
404     real temp = temp_base * st_env.er_inv[iap];
405     real temp_irate = pow(temp, env__alpha_c);
406     real irate = st_env.irate_factor[iap] * temp_irate;
407     #pragma HLS bind_op variable=irate op=fmul
408     real imrt = env__delta_c + irate;
409     real temp_wage = pow(temp, env__alpha);
410     real wage = st_env.wage_factor[iap] * temp_wage;
411     #pragma HLS bind_op variable=wage op=fmul
412     small_idx_t kpb = iap << 2;

```

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

```

413 ihp_4:
414 for (int iidp = 0; iidp < NSTATES_ID; ++iidp)
415 {

```

- (c) Use a linear interpolation scheme to determine tomorrow's individual capital-holding choice $\text{fp} = k'' = k'_i(k'_i) = (k'(k, \epsilon, m, A), \epsilon', m', A')$

```

438 small_idx_t idx_base = li_2d_aux_idx_base[p_idx_inner];
439 small_idx_t i1_min_base = idx_base + (NSTATES_ID * NKGRID * i1_min);
440 small_idx_t i1_max_base = idx_base + (NSTATES_ID * NKGRID * i1_max);
441 real tz_num = (kmp - i1_min_val);
442 real tz_den = (i1_max_val - i1_min_val);

```

```

443     real tz = tz_num / tz_den;
444     real tw_num = (kp - i2_min_val);
445     real tw_den = (i2_max_val - i2_min_val);
446     real tw = tw_num / tw_den;
447     real sub_tz = (1.0 - tz);
448     real sub_tw = (1.0 - tw);
449     real sub_tz_sub_tw = sub_tz * sub_tw;
450     real tz_tw = tz * tw;
451     real sub_tz_tw = sub_tz * tw;
452     real tz_sub_tw = tz * sub_tw;
453     #if (NUM_KPRIMES == 1)
454         real fp_1 = st_kprimes[0][i1_min_base + i2_min] * sub_tz_sub_tw;
455         real fp_2 = st_kprimes[0][i1_min_base + i2_max] * sub_tz_tw;
456         real fp_3 = st_kprimes[0][i1_max_base + i2_min] * tz_sub_tw;
457         real fp_4 = st_kprimes[0][i1_max_base + i2_max] * tz_tw;
458     #elif (NUM_KPRIMES == 4)
459         real fp_1 = st_kprimes[0][i1_min_base + i2_min] * sub_tz_sub_tw;
460         real fp_2 = st_kprimes[1][i1_min_base + i2_max] * sub_tz_tw;
461         real fp_3 = st_kprimes[2][i1_max_base + i2_min] * tz_sub_tw;
462         real fp_4 = st_kprimes[3][i1_max_base + i2_max] * tz_tw;
463     #elif (NUM_KPRIMES == 8)
464         real fp_1 = st_kprimes[kpb + 0][i1_min_base + i2_min] * sub_tz_sub_tw;
465         real fp_2 = st_kprimes[kpb + 1][i1_min_base + i2_max] * sub_tz_tw;
466         real fp_3 = st_kprimes[kpb + 2][i1_max_base + i2_min] * tz_sub_tw;
467         real fp_4 = st_kprimes[kpb + 3][i1_max_base + i2_max] * tz_tw;
468     #endif
469     real fp_5 = fp_1 + fp_2;
470     real fp_6 = fp_3 + fp_4;
471     real fp = fp_5 + fp_6;

```

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

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

```

472     real cons2_1 = imrt * kp;
473     real cons2_2 = wage * st_env.cons2_factor[p_idx_inner];
474     real cons2_temp = (cons2_1 + cons2_2);
475     real cons2 = cons2_temp - fp;
476     if (cons2 < 0) // eq 11
477         cons2 = CONS2_MIN;
478     real mu2 = (1.0/cons2);
479     // real mu2 = hw_pow(cons2, env_gamma_neg);
480     real emu_s_1 = imrt * mu2;
481     emu_s += (emu_s_t)(st_env.P[p_idx_outer + p_idx_inner] * emu_s_1);
482     ++p_idx_inner;

```

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

```

377     emu_s_t emu_s = 0.;

```

```

398     ihp_3:
399     for (int iap = 0; iap < NSTATES_AG; ++iap)
400     {

```

```

413 ihp_4:
414   for (int iidp = 0; iidp < NSTATES_ID; ++iidp)
415   {

```

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

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

```

487   real temp1_new_kp = (env__beta * (real)emu_s);
488   real temp2_new_kp = (1.0/temp1_new_kp);
489   real new_kp = init_wealth[is] - temp2_new_kp; // eq 10
490   // 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 multiplier λ to 0..

3.6.5.4 Closing the IAP Loop.

1. Update the guess.

```

509 ihp_5:
510   for (small_idx_t is = 0; is < NSTATES; ++is)
511   {
512     #pragma HLS pipeline
513     real temp1_kp = UPDATE_K * kprime_new[is]; // eq 13
514     real temp2_kp = UPDATE_K_C * st_kprimes[0][is]; // eq 13
515     real updated_kp = temp1_kp + temp2_kp; // eq 13
516     for (small_idx_t k = 0; k < NUM_KPRIMES; ++k)
517       st_kprimes[k][is] = updated_kp;
518   }

```

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

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 $\text{metric} = \rho(k'_{i+1}, k'_i)$.

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

```

527   // ~ Update metric
528   metric = (real) spread_scalar ;

```

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.

```

556 #if (PARTITION_KCROSS == 1)
557 #pragma HLS array_partition variable = st_kcross type = cyclic factor = 1
558 #elif (PARTITION_KCROSS == 4)
559 #pragma HLS array_partition variable = st_kcross type = cyclic factor = 4
560 #elif (PARTITION_KCROSS == 8)
561 #pragma HLS array_partition variable = st_kcross type = cyclic factor = 8
562 #endif
563
564 #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.

```

566 #if AST_UNROLL
567     real kprime_interp0[NUM_KCROSS][NSTATES_ID * NKGRID];
568     real kprime_interp1[NUM_KCROSS][NSTATES_ID * NKGRID];
569     #pragma HLS array_partition variable = kprime_interp0 complete dim = 1
570     #pragma HLS array_partition variable = kprime_interp1 complete dim = 1

```

```

571 #pragma HLS array_partition variable = st_env.epsilon2 complete dim = 1
572 #else
573     real kprime_interp0[NSTATES_ID * NKGRID];
574     real kprime_interp1[NSTATES_ID * NKGRID];
575 #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.

```

577 #if PACK_IDS
578     small_idx_t idshock_cnt = 64;
579     ap_uint<72> temp_ids = idshock[0];
580 #else
581     small_idx_t idshock_cnt = 8;
582 #endif

```

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

```

583 int idshock_idx = 0;
584 idx_t agshock_idx = 0;
585 shock_t curr_ids;
586 shock_t curr_ag;
587 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.

```

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

```

3.6.6.2 Loop.

For each time period $t \in \{0, \dots, 1099\}$ ¹

```

597 ast_1:
598 for (int t = 0; t < SIM_STEPS; ++t)
599 {

```

1. **Interpolation.** For each individual $j = 1, \dots, 10,000$, 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.

```

604 kmts[t] = curr_kmts;
605
606 // Read next packed agshock value when needed

```

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


```

607     if (++ags_phase >= AGS_PACK_FACTOR)
608     {
609         curr_ags = agshock[agshock_idx++];
610         ags_phase = 0;
611     }
612
613     bool p0 = (curr_ags & 0b1) ? 0b1 : 0b0;
614
615     curr_ags >>= 1;
616     real p1 = kmnts[t];
617     #if defined(_SERIAL_CPU_MODE) || defined(_OPENMPI_MODE)
618     small_idx_t i2_min = hw_findrange(p1, st_env.km, NKM_GRID);
619     small_idx_t i2_max = i2_min + 1;
620     real i2_min_val = st_env.km[i2_min];
621     real i2_max_val = st_env.km[i2_max];
622     #else
623     small_idx_t i2_min = hw_findrange((fixed_t)p1, fxd_km_grid, NKM_GRID);
624     small_idx_t i2_max = i2_min + 1;
625     real i2_min_val = km_grid[i2_min];
626     real i2_max_val = km_grid[i2_max];
627     #endif
628     real ty_num = (p1 - i2_min_val);
629     real ty_den = (i2_max_val - i2_min_val);
630     real ty = ty_num / ty_den;
631     real P = (p0 == 1) ? 0 : (1.0 - ty);
632     real Q = (p0 == 1) ? 0 : (ty);
633     real R = (p0 == 1) ? (1.0 - ty) : 0;
634     real S = (p0 == 1) ? (ty) : 0;
635     small_idx_t i1_min_base = 0; // L4D_D3 * i1.min(0)
636     small_idx_t i1_max_base = L4D_D3; // L4D_D3 * i1.max
637     small_idx_t i2_min_base = L4D_D2 * i2_min;
638     small_idx_t i2_max_base = L4D_D2 * i2_max;
639     small_idx_t i12_min_min = i1_min_base + i2_min_base;
640     small_idx_t i12_min_max = i1_min_base + i2_max_base;
641     small_idx_t i12_max_min = i1_max_base + i2_min_base;
642     small_idx_t i12_max_max = i1_max_base + i2_max_base;
643     small_idx_t kpi_idx = 0;
644

```

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, $\epsilon_{t,j} \in \{0, 1\}_\epsilon$

```

644     ast_2:
645     for (int iid = 0; iid < NSTATES_ID; ++iid)
646     {
647         small_idx_t i3_min_base = 0; // L4D_D1 * i3.min (0)
648         small_idx_t i3_max_base = L4D_D1; // L4D_D1 * i3.max (1)
649         real tz = st_env.epsilon[iid];
650

```

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

```

523     ast_3:
524     for (int ik = 0; ik < NKGRID; ++ik)
525     {
526     #pragma HLS pipeline
527     int i4_min = ik;

```

```

528     real p = (1.0 - tz);
529     real r = tz;
530

```

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

```

657     small_idx_t kp_idx_0 = i4_min + i3_min_base + i12_min_min;
658     small_idx_t kp_idx_2 = i4_min + i3_max_base + i12_min_min;
659     small_idx_t kp_idx_4 = i4_min + i3_min_base + i12_min_max;
660     small_idx_t kp_idx_6 = i4_min + i3_max_base + i12_min_max;
661     small_idx_t kp_idx_8 = i4_min + i3_min_base + i12_max_min;
662     small_idx_t kp_idx_10 = i4_min + i3_max_base + i12_max_min;
663     small_idx_t kp_idx_12 = i4_min + i3_min_base + i12_max_max;
664     small_idx_t kp_idx_14 = i4_min + i3_max_base + i12_max_max;
665     // ** LI3D
666     #if ((NUM_KPRIMES == 4) || (NUM_KPRIMES == 8))
667         real fp_1 = st_kprimes[0][kp_idx_0] * P * p;
668         real fp_2 = st_kprimes[0][kp_idx_2] * P * r;
669         real fp_3 = st_kprimes[1][kp_idx_4] * Q * p;
670         real fp_4 = st_kprimes[1][kp_idx_6] * Q * r;
671         real fp_5 = st_kprimes[2][kp_idx_8] * R * p;
672         real fp_6 = st_kprimes[2][kp_idx_10] * R * r;
673         real fp_7 = st_kprimes[3][kp_idx_12] * S * p;
674         real fp_8 = st_kprimes[3][kp_idx_14] * S * r;
675         real fp_9 = fp_1 + fp_2;
676         real fp_10 = fp_3 + fp_4;
677         real fp_11 = fp_5 + fp_6;
678         real fp_12 = fp_7 + fp_8;
679         real fp_13 = fp_9 + fp_10;
680         real fp_14 = fp_11 + fp_12;
681         real fp = fp_13 + fp_14;
682     #elif (NUM_KPRIMES == 1)
683         real fp = st_kprimes[0][kp_idx_0] * P * p +
684                 st_kprimes[0][kp_idx_2] * P * r +
685                 st_kprimes[0][kp_idx_4] * Q * p +
686                 st_kprimes[0][kp_idx_6] * Q * r +
687                 st_kprimes[0][kp_idx_8] * R * p +
688                 st_kprimes[0][kp_idx_10] * R * r +
689                 st_kprimes[0][kp_idx_12] * S * p +
690                 st_kprimes[0][kp_idx_14] * S * r;
691     #endif
692

```

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

```

692     #if AST_UNROLL
693         for (int k = 0; k < NUM_KCROSS; ++k)
694         {
695             kprime_interp0[k][kpi_idx] = fp;
696             kprime_interp1[k][kpi_idx] = fp;
697         }
698     #else
699         kprime_interp0[kpi_idx] = fp;
700         kprime_interp1[kpi_idx] = fp;
701     #endif
702     ++kpi_idx;
703

```

Initialise the aggregate capital to 0

```

706 //aggregate capital initialized to 0
707 kc_t agg_capital = 0;
708

```

Iterate over `N_AGENTS` using 8 parallel pipelines. The `#pragma HLS PIPELINE` unrolls the inner loop completely creating 8 pipelines. The `IDS_HOCKS` 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`

```

710 small_idx_t kidx = 0;
711 // Loop 1.3: AST agents interp over kprime_interp
712 // Unroll factor dictated by inner loop over k
713 #if PACK_IDS
714 idshock_cnt = 8;
715 #endif
716 ast_4:
717 for (int j = 0; j < (N_AGENTS / IDS_PACK_FACTOR) / IDS_AGG_X; j++)
718 {
719 #pragma HLS pipeline
720 #if PACK_IDS
721 if (idshock_cnt >= 8)
722 {
723 idshock_cnt = 0;
724 temp_ids = idshock[idshock_idx];
725 idshock_idx++;
726 }
727 curr_ids = temp_ids & 0xFF;
728 idshock_cnt++;
729 temp_ids >>= 8;
730 #else
731 curr_ids = idshock[idshock_idx++];
732 #endif
733

```

Initialize values for interpolation over `kprime_interp0` and `kprime_interp1` from above

```

734 ast_5:
735 for (int k = 0; k < IDS_PACK_FACTOR * IDS_AGG_X; ++k)
736 {
737 real p1b = st_kcross[kidx];
738 small_idx_t i2b_min = hw_findrange((fixed_t)st_kcross[kidx], fxd_k_grid, NKGRID);
739 small_idx_t i2b_max = i2b_min + 1;
740 real i2b_min_val = k_grid[i2b_min];
741 real i2b_max_val = k_grid[i2b_max];
742 bool p0b = (curr_ids & 0b1) ? 0b1 : 0b0;
743 curr_ids >>= 1;
744 small_idx_t i1b_min_base = 0; // NKGRID * i1b_min(0)
745 small_idx_t i1b_max_base = NKGRID; // NKGRID * i1b_max(1)
746 real bw_num = (p1b - i2b_min_val);
747 real bw_den = (i2b_max_val - i2b_min_val);
748 real bw = bw_num / bw_den;
749 real sub_bw = (1.0 - bw);
750 real bz_bw = (p0b == 1) ? bw : 0;
751 real sub_bz_sub_bw = (p0b == 1) ? 0 : sub_bw;
752 real bz_sub_bw = (p0b == 1) ? sub_bw : 0;
753 real bw_sub_bz = (p0b == 1) ? 0 : bw;
754

```

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

```

754   real fpb_1 = (kprime_interp0[k][i1b_min_base + i2b_min] * sub_bz_sub_bw);
755   real fpb_2 = (kprime_interp0[k][i1b_min_base + i2b_max] * bw_sub_bz);
756   real fpb_3 = (kprime_interp1[k][i1b_max_base + i2b_min] * bz_sub_bw);
757   real fpb_4 = (kprime_interp1[k][i1b_max_base + i2b_max] * bz_bw);
758   real fpb_5 = fpb_1 + fpb_2;
759   real fpb_6 = fpb_3 + fpb_4;
760   kc_t_fpb = kc_t(fpb_5 + fpb_6);
761   hw_fxd_rail_values(&fpb, KMAX, KMIN);
762   st_kcross[kidx] = (real)fpb;
763   agg_capital += fpb;
764   kidx++;
765

```

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

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

```

837   curr_kmts = ((real)agg_capital * N_AGENTS_INV);
838

```

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

```

838   hw_rail_values(&curr_kmts, KM_MAX, KM_MIN);
839

```

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

```

849   real coeff[NCOEFF] = {0.};
850   sim_alm_1:
851   for (small_idx_t i = 0; i < NCOEFF; i++)
852   {
853   #pragma HLS pipeline off
854       coeff[i] = coeff_updated[i];
855   }

```

Initializations

```

856 small_idx_t agshock_idx = 0;
857 small_idx_t ags_phase = AGS_PACK_FACTOR;
858 shock_t curr_ags = 0;
859 shock_t curr_shock_val = 0;
860 real coeff_new[NCOEFF] = {0.};
861 real x_good_v[1000] = {0.};
862 real y_good_v[1000] = {0.};
863 real x_bad_v[1000] = {0.};
864 real y_bad_v[1000] = {0.};
865
866 int ibad = 0;
867 int igood = 0;
868 agshock_idx = 0;
869 ags_phase = AGS_PACK_FACTOR;

sim_alm_2:
870 for (int t = 0; t < SIM_STEPS; t++)
871 {
872     #pragma HLS pipeline off
873     #pragma HLS unroll factor = 1
874     // Read new value when needed
875     if (++ags_phase >= AGS_PACK_FACTOR)
876     {
877         curr_ags = agshock[agshock_idx++];
878         ags_phase = 0;
879     }
880     curr_shock_val = curr_ags & 0b1; // take the least significant bit from the byte
881     curr_ags >>= 1; // right shift by 1
882     // Discard first 100
883     sim_alm_3:
884     if (t < NDISCARD || t > SIM_STEPS - 2)
885         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 $a_t = a_b$, $\{b_1(a_t), b_2(a_b)\}$

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

```

887 sim_alm_4:
888 if (curr_shock_val == 0)
889 {
890     y_bad_v[ibad] = hw_log(kmts[t + 1]);
891     x_bad_v[ibad] = hw_log(kmts[t]);
892     ibad++;
893 }

```

it collects

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

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

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

```

894     else
895     {
896         y_good_v[igood] = hw_log(kmts[t + 1]);
897         x_good_v[igood] = hw_log(kmts[t]);
898         igood++;
899     }

```

it collects

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

```

901     real badcoeff[2] = {0}; // initialize to prevent garbage values
902     real goodcoeff[2] = {0};
903     regression(badcoeff, x_bad_v, y_bad_v, ibad);
904     regression(goodcoeff, x_good_v, y_good_v, igood);
905     real rbad = RSquaredCalc(badcoeff, x_bad_v, y_bad_v, ibad);
906     real rgood = RSquaredCalc(goodcoeff, x_good_v, y_good_v, igood);
907     coeff_new[0] = badcoeff[0]; // bb
908     coeff_new[1] = badcoeff[1];
909     coeff_new[2] = goodcoeff[0];
910     coeff_new[3] = goodcoeff[1];
911     R2[0] = rbad;
912     R2[1] = rgood;

```

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

$$\begin{aligned} \ln m_{l+1} &= b_1(a_l) + b_2(a_l) \ln m_l + \epsilon_{l+1}, & l \in \{t \in \{100, \dots, 1100\} : a_t = a_b\} \\ \ln m_{l+1} &= b_1(a_l) + b_2(a_l) \ln m_l + \epsilon_{l+1}, & l \in \{t \in \{100, \dots, 1100\} : a_t = a_g\} \end{aligned}$$

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

```

913     // Update metric for convergence test (eq 17)
914     real norm = 0.;
915     sim_alm_5:
916     for (int ib = 0; ib < NCOEFF; ++ib)
917     {
918         #pragma HLS pipeline off
919         norm += (coeff_new[ib] - coeff[ib]) * (coeff_new[ib] - coeff[ib]);
920     }
921     *metric = hw_sqrt(norm);

```

Compute the Euclidean Norm.

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

```

922 // Update ALM coefficients vector
923 sim_alm_6:
924 for (int ib = 0; ib < NCOEFF; ++ib)
925 {
926 #pragma HLS pipeline off
927     coeff_updated[ib] = coeff_new[ib] * UPDATE_B + coeff[ib] * (1. - UPDATE_B); //
928 }

```

Update the Coefficients.

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

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.

```

932 void regression(real *resultmatrix, real *x, real *y, int ndim)
933 {
934     real twobytwo[4] = {0, 0, 0, 0};
935     RG_1:
936     for (int i = 0; i < ndim; i++)
937     {
938 #pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
939 #pragma HLS unroll factor = 1
940 #pragma HLS pipeline off
941         twobytwo[0] += 1;
942         twobytwo[1] += x[i];
943         twobytwo[2] += x[i];
944         twobytwo[3] += hw_pow(x[i], 2);
945     }
946     // get inverse
947     real a = twobytwo[0]; // switching indices and multiplying by determinant
948     real b = twobytwo[1];
949     real c = twobytwo[2];
950     real d = twobytwo[3];
951     real det = (a * d - b * c);
952
953     real inv_det = (1.0 / det);
954     real inv_d = inv_det * d;
955     real inv_b = inv_det * (b) * -1;
956     real inv_c = inv_det * (c) * -1;
957     real inv_a = inv_det * a;
958     real acc1 = resultmatrix[0];
959     real acc2 = resultmatrix[1];
960     // multiply by transpose of matrix and y

```

```

961 RG_2:
962   for (int i = 0; i < ndim; i++)
963   {
964     #pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
965     #pragma HLS unroll factor = 1
966     #pragma HLS pipeline off
967     real acc_t1 = inv_b * x[i];
968     real acc_t2 = inv_d + acc_t1;
969     acc1 += acc_t2 * y[i];
970   }
971   resultmatrix [0] = acc1;
972 RG_3:
973   for (int i = 0; i < ndim; i++)
974   {
975     #pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
976     #pragma HLS unroll factor = 1
977     #pragma HLS pipeline off
978     real acc2_t1 = inv_a * x[i];
979     real acc2_t2 = inv_c + acc2_t1;
980     acc2 += acc2_t2 * y[i];
981   }
982   resultmatrix [1] = acc2;
983   return;
984 }

```

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).

```

990 real RSquaredCalc(real *coeff, real *x, real *y, int ndim)
991 {
992   real r_value = 0;
993   real predict [1000] = {0};
994   real rss = 0;
995   real tss = 0;
996   real y_mean = 0;
997   R2_1:
998     for (int i = 0; i < ndim; i++)
999     {
1000       #pragma HLS pipeline off
1001       #pragma HLS unroll factor = 1
1002       #pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
1003       y_mean += y[i];
1004     }
1005     y_mean = (y_mean / ndim);
1006
1007   R2_2:
1008     for (int i = 0; i < ndim; i++)
1009     {

```



```

1010 #pragma HLS pipeline off
1011 #pragma HLS unroll factor = 1
1012 #pragma HLS loop_tripcount min = 100 avg = 494 max = 1000
1013     predict[i] = (coeff[0] + (coeff[1] * x[i]));
1014     rss += hw_pow((predict[i] - y[i]), 2);
1015     tss += hw_pow((y[i] - y_mean), 2);
1016 }
1017 r_value = (1.0 - (rss / tss));
1018
1019 return r_value;

```

3.6.8 Math Functions

Collection of double precision operations - ([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.

```

1096 real hw_exp(real b)
1097 {
1098 #pragma HLS inline
1099 #if USE_HLS_LIB
1100     return hls::exp(b);
1101 #else
1102     return exp(b);
1103 #endif
1104 }

```

3.6.9 Linear Interpolation

3.6.9.1 [hw_fndrange](#)

Description: This function uses an optimized routine to find the interpolation range. Based on the config knob selection in [dev_options.h](#), we choose between the 3 different algorithms [LINEAR_SEARCH](#), [BINARY_SEARCH](#) or [CUSTOM_BINARY_SEARCH](#) to implement this - The [CUSTOM_BINARY_SEARCH](#) function comes in five versions, which differ in the size of the interpolation grids: [new_hw_fndrange_n4](#), [hw_fndrange_n8](#), [hw_fndrange_n100](#),

[hw_findrange_n200](#), [hw_findrange_n300](#).

Acceleration: Unrolling, Pipelining.

```

1336 small_idx_t hw_findrange(fixed_t p, const fixed_t *src, int n_elem)
1337 {
1338 #if _LINEAR_SEARCH || _BASELINE
1339     small_idx_t result = 1;
1340     for (signed short i = (n_elem - 1); i > 0; --i)
1341     {
1342         if (p <= src[i])
1343         {
1344             result = i - 1;
1345         }
1346     }
1347     return result;
1348 #elif _BINARY_SEARCH
1349 #ifdef _FPGA_MODE
1350     printf("ERROR: No support for binary search in FPGA\n");
1351 #else
1352     small_idx_t l = 0;
1353     small_idx_t r = n_elem - 1; // last index for array with 100 elements
1354     while(l <= r){
1355
1356         small_idx_t m = (l + r) >> 1; // div by 2
1357
1358         // If x greater, ignore left half
1359         if (src[m] <= p)
1360             l = m + 1;
1361
1362         // If x is smaller, ignore right half
1363         else{
1364             r = m - 1;
1365         }
1366     }
1367
1368     r = (p==src[0]) ? 0 : r;
1369
1370
1371     return r;
1372 #endif // _FPGA_MODE
1373 #elif _CUSTOM_BINARY_SEARCH || (_PIPELINE || _WITHIN_ECONOMY || _ACROSS_ECONOMY)
1374 #pragma HLS inline
1375 #if (NKM_GRID == 4)
1376     if (n_elem == 4)
1377         return hw_findrange_n4(p, src);
1378 #elif (NKM_GRID == 8)
1379     if (n_elem == 8)
1380         return hw_findrange_n8(p, src);
1381 #endif
1382 #if (NKGRID == 100)
1383     else if (n_elem == 100)
1384         return hw_findrange_n100(p, src);
1385 #elif (NKGRID == 200)
1386     else if (n_elem == 200)
1387         return hw_findrange_n200(p, src);
1388 #elif (NKGRID == 300)
1389     else if (n_elem == 300)
1390         return hw_findrange_n300(p, src);
1391 #endif

```

```

1392     else
1393         return 0;
1394
1395 #endif
1396 }

```

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 custom 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, $i = \{0, \dots, N_k\}$. 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 Π 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

```

1207 small_idx_t hw_findrange_n100(fixed_t p, const fixed_t *src)
1208 {
1209 #pragma HLS pipeline
1210     small_idx_t result_1 = 0;
1211     small_idx_t result_2 = 0;
1212     small_idx_t result_3 = 0;

```

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

```

1213  small_idx_t  result = 0;
1214
1215  fr100_1:
1216  for (signed short i = 99; i > 0; i=i-10) //10 comparators
1217  {
1218  fr100_2:
1219    if (p <= src[i])
1220    {
1221      result_1 = i; //send the max index
1222    }
1223  }
1224
1225  fr100_3:
1226  for (signed short i = 2; i > 0; i--) // 2 comparators
1227  {
1228  fr100_4:
1229    if (p <= src[result_1])
1230    {
1231      result_2 = result_1; //send the max index
1232    }
1233    result_1 = result_1 - (small_idx_t)5;
1234  }
1235
1236  fr100_5:
1237  for (signed short i = 5; i > 0; i--) //5 comparators
1238  {
1239  fr100_6:
1240    if (p <= src[result_2--])
1241    {
1242      result_3 = result_2; //send the min index
1243    }
1244  }
1245
1246  result = (p==src[0]) ? (small_idx_t)0 : result_3;
1247  return result;
1248  }

```

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.

```

1146  void hw_rail_values(real *val, const real max, const real min)
1147  {
1148  #pragma HLS inline
1149    real src = *val;
1150    bool over_max = (src > max);
1151    bool under_min = (src < min);
1152
1153    hw_rail_1:
1154    if (over_max)
1155      *val = max;
1156    else if (under_min)
1157      *val = min;
1158    return;
1159  }

```

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](#).

```

1 #check if the platform is the latest version
2 platform=xilinx_aws-vu9p-f1_shell-v04261818_201920_3
3 debug=1
4 profile_kernel =data: all : all : all
5 save-temps=1
6
7 [hls]
8 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.

```

1 source $AWS_FPGA_REPO_DIR/vitis_setup.sh
2 export PLATFORM_REPO_PATHS=$(dirname $AWS_PLATFORM)
3 platforminfo - $AWS_FPGA_REPO_DIR
4
5
6
7
8
9
10
11 #Enable either single kernel or three kernel
12 [ connectivity ]
13 #####single kernel start#####
14 # nk=runOnfpga:1:runOnfpga_1
15 #####three kernel start#####
16 nk=runOnfpga:3:runOnfpga_1.runOnfpga_2.runOnfpga_3
17 #slr=<compute_unit_name>:<slr_ID>
18 slr=runOnfpga_1:SLR2
19 slr=runOnfpga_2:SLR1
20 slr=runOnfpga_3:SLR0
21 sp=runOnfpga_1.m_axi_gmem0:DDR[1]
22 sp=runOnfpga_2.m_axi_gmem0:DDR[0]
23 sp=runOnfpga_3.m_axi_gmem0:DDR[3]
24 #####three kernel end#####
25
26 [vivado]
27 prop=run.impl_1.strategy=Performance_ExtraTimingOpt

```

3.7.2 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.

```
1 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.

```
1 #Start of Debug group
2 [Debug]
3 profile=true
4 timeline_trace=true
5 data_transfer_trace=coarse
6 opencl_summary=true
7 opencl_device_counter=true
8 opencl_trace=true
```

Figure 3.6: Information from xclbinutil

```

=====
Resource Availability
=====

=====
Total
=====

=====
Per SLR
=====

SLR0:
SLR1:
SLR2:

=====
Memory Information
=====
=====
Bus SP Tag: DDR
Segment Index: 0
Consumption: automatic
SP Tag: bank0
SLR: SLR1
Max Masters: 15
Segment Index: 1
Consumption: automatic
SP Tag: bank1
SLR: SLR2
Max Masters: 15
Segment Index: 2
Consumption: automatic
SP Tag: bank2
SLR: SLR1
Max Masters: 15
Segment Index: 3
Consumption: automatic
SP Tag: bank3
SLR: SLR0
Max Masters: 15
Bus SP Tag: PLRAM
Segment Index: 0
Consumption: explicit
SLR: SLR2
Max Masters: 15
Segment Index: 1
Consumption: explicit
SLR: SLR1
Max Masters: 15
Segment Index: 2
Consumption: explicit
SLR: SLR0
Max Masters: 15

```

3.8 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.

```
46 TARGET := hw
47 MPICXX := mpicxx
48 CC := g++
49 FPGA_INCLUDES := -I./common -I./common/libs -I./fpga -I$(XILINX_XRT)/include -I$(XILINX_VIVADO)/include
50 CPU_INCLUDES := -I./common -I./fpga
51 OPENMPI_INCLUDES := -I./common -I./fpga
52 PLATFORM := xilinx_aws-vu9p-f1_shell-v04261818_201920_3
```

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.

```
53 OPENMPI_FLAG := -D_OPENMPI_MODE
54 FPGA_FLAG := -D_FPGA_MODE
55 SERIAL_CPU_FLAG := -D_SERIAL_CPU_MODE
```

These are the different host flags that will be used to work with FPGA, CPU, and OPENMPI later on during the compilation and linking.

```
58 CXXFLAGS := $(FPGA_INCLUDES) -Wall -O3 -g -std=c++1y -fmessage-length=0 -L$(XILINX_XRT)/lib -pthread -lOpenCL -lrt -lstdc++
59 CXXFLAGS2 := $(OPENMPI_INCLUDES) -Wall -O3 -g -std=c++1y -fmessage-length=0 -pthread -lrt -lstdc++
60 CXXFLAGS3 := $(CPU_INCLUDES) -Wall -O3 -g -std=c++1y -fmessage-length=0 -pthread -lrt -lstdc++
```

These are additional kernel flags to be passed for FPGA to keep things more organized into the folder directories that we need.

```
63 KRNL_COMPILE_OPTS := -t $(TARGET) --config ./fpga/design.cfg --log_dir ./fpga/logs --report_dir ./fpga/reports --save-temps --jobs 8 --optimize 3
64 KRNL_LINK_OPTS := -t $(TARGET) --config ./fpga/design.cfg --log_dir ./fpga/logs --report_dir ./fpga/reports --save-temps --jobs 8 --optimize 3
```

Define buckets for storing temporary and permanent results in AWS S3.

```
67 AWS_REGION := us-west-2
68 # 1. TEMPORARY BUCKET
69 S3_BUCKET_NAME := ksfpga-$(shell aws sts get-caller-identity | grep "Account" | tr -dc '0-9')
70 S3_DCP_DIR := vitis-dcps
71 S3_LOG_DIR := vitis-logs
72 S3_EXEC_DIR := executables
73 # 2. PERMANENT BUCKET
```



```

74 # AWS S3 stores the final executables and later final results
75 # fpga-econ-ks
76 #   - executables
77 #     - fpga
78 #       - fpga_afi
79 #       - host_executables
80 #     - cpu
81 # - results (will be generated by make fpga_results, make cpu_results)
82 # - fpga
83 # - cpu
84 S3_EXE_BUCKET_NAME := fpga-econ-ks
85 S3_EXE_DIR := executables
86 S3_FPGA_DIR := $(S3_EXE_DIR)/fpga
87 S3_FPGA_AFI_DIR := $(S3_FPGA_DIR)/fpga_afi
88 S3_HOST_EXEC_DIR := $(S3_FPGA_DIR)/host_executables
89 S3_CPU_DIR := $(S3_EXE_DIR)/cpu
90 S3_RESULTS_DIR := results
91 S3_RESULTS_FPGA_DIR := $(S3_RESULTS_DIR)/fpga
92 S3_RESULTS_FPGA_REPORTS_DIR := $(S3_RESULTS_FPGA_DIR)/reports

```

List the valid executable names for this project.

```

95 # List of valid executable names for replication purpose
96 # Sequential CPU: make cpu_to_s3 CPU_EXE=<one of these names>
97 VALID_CPU_BIN_NAMES := 1200_100k_4km 1200_200k_4km 1200_300k_4km 1200_100k_8km 1200_200k_8km 1200_300k_8km 120
   _100k_4km 1200_linear 1200_binary
98 # OpenMPI: make openmpi_to_s3 OPENMPI_EXE=<one of these names>
99 VALID_OPENMPI_BIN_NAMES := mpi_1200_100k_4km
100 #1200_200k_4km 1200_300k_4km 1200_100k_8km 1200_200k_8km 1200_300k_8km 120_100k_4km 1200_linear 1200_binary 1200
   _custom_binary
101 # FPGA host: make afi FPGA_BIN=<> HOST_BIN=<one of these names>
102 VALID_HOST_BIN_NAMES := 1200_1ker_100k_4km 1200_1ker_200k_4km 1200_1ker_300k_4km 1200_1ker_100k_8km 1200
   _1ker_200k_8km 1200_1ker_300k_8km 1200_3ker_100k_4km 1200_3ker_200k_4km 1200_3ker_300k_4km 1200_3ker_100k_8km
   1200_3ker_200k_8km 1200_2ker_300k_8km 120_1ker_100k_4km 120_3ker_100k_4km
103 # FPGA afi: make afi FPGA_BIN=<one of these names> HOST_BIN=<>
104 VALID_AFI_NAMES := 1ker_100k_4km 1ker_200k_4km 1ker_300k_4km 1ker_100k_8km 1ker_200k_8km 1ker_300k_8km 3
   ker_100k_4km 3ker_200k_4km 3ker_300k_4km 3ker_100k_8km 3ker_200k_8km 2ker_300k_8km baseline_1ker_100k_4km
   pipeline_1ker_100k_4km within_economy_1ker_100k_4km
105
106 # Names of the executables / binaries
107 CPU_EXE ?= app
108 OPENMPI_EXE ?= openmpi_app
109 ifndef HOST_BIN
110     HOST_EXE = host
111 else
112     HOST_EXE = $(HOST_BIN)
113 endif
114 FPGA_EXE = runOnfpga
115 XO := ./fpga/build/$(FPGA_EXE).xo
116 XCLBIN := ./fpga/build/$(FPGA_EXE).xclbin

```

Build the binaries

```

118 # 1. Sequential CPU (g++)
119 $(CPU_EXE): ./common/init.cpp ./common/app.cpp ./fpga/hw.cpp
120     $(CC) $(SERIAL_CPU_FLAG) $(CXXFLAGS3) $^ -o $@
121
122 # 2. OpenMPI (mpic++)
123 $(OPENMPI_EXE): ./common/init.cpp ./common/app.cpp ./fpga/hw.cpp
124     $(MPICXX) $(OPENMPI_FLAG) $(CXXFLAGS2) $^ -o $@

```

```

125
126 # 3. FPGA
127 # - FPGA host
128 $(HOST_EXE): ./common/libs/xcl2.cpp ./common/app.cpp ./common/init.cpp
129 $(CC) $(FPGA_FLAG) $(CXXFLAGS) $^ -o $@
130
131 # - FPGA kernel
132 $(XO): ./fpga/hw.cpp
133 v++ -I./common -I./fpga -I./ $(FPGA_FLAG) $(KRNL_COMPILE_OPTS) -c -k $(FPGA_EXE) -o'${@}' '<'
134
135 # - FPGA .xclbin
136 $(XCLBIN): $(XO)
137 v++ -I./common -I./fpga -I./ $(KRNL_LINK_OPTS) -l -o'${@}' $(+)
```

Targets to create the executable

```

141 # Build sequential CPU executable
142 cpu: $(CPU_EXE)
143
144 # Build OpenMPI multi-core CPU executable
145 openmpi: $(OPENMPI_EXE)
146
147 # Build FPGA host executable
148 exe: $(HOST_EXE)
149
150 # Build FPGA kernel and .xclbin
151 xclbin: $(XO) $(XCLBIN)
152
153 # Build FPGA host, kernel and .xclbin
154 fpga: $(XO) $(XCLBIN) $(HOST_EXE) emconfig
```

This ensures that the CPU executable is valid, builds it, and uploads it to an S3 bucket. Error Handling: It includes a check to ensure the executable name is valid, providing a clear error message if not. [.PHONY: cpu_to_s3](#) and [.PHONY: check_cpu_variables](#): These lines declare *cpu_to_s3* and *check_cpu_variables* as phony targets, meaning they are not actual files but labels for commands to run.

CPU Target

```

157 # Build sequential CPU executable and upload it on S3 bucket
158 .PHONY: cpu_to_s3
159 cpu_to_s3: check_cpu_variables create_s3_dirs $(CPU_EXE)
160 aws s3 cp $(CPU_EXE) s3://$(S3_EXE_BUCKET_NAME)/$(S3_CPU_DIR)/
161
162 .PHONY: check_cpu_variables
163 check_cpu_variables:
164 ifeq ($( filter $(CPU_EXE),$(VALID_CPU_BIN_NAMES)),)
165 $(error Invalid value for host executable for m5n.xx instance . Please specify CPU_EXE= one of: $(
    VALID_CPU_BIN_NAMES))
166 endif
```

OPENMPI Target

```

169 # Build design for parallel execution using OPENMPI and upload binary into S3 bucket
170 .PHONY: openmpi_to_s3
171 openmpi_to_s3: check_openmpi_variables create_s3_dirs $(OPENMPI_EXE)
172 aws s3 cp $(OPENMPI_EXE) s3://$(S3_EXE_BUCKET_NAME)/$(S3_CPU_DIR)/
```

```

173
174 .PHONY: check_openmpi_variables
175 check_openmpi_variables:
176 ifeq ($(filter $(OPENMPI_EXE),$(VALID_OPENMPI_BIN_NAMES)),)
177 $(error Invalid value for host executable for m5n.xx instance. Please specify OPENMPI_EXE= one of: $(
    VALID_OPENMPI_BIN_NAMES))
178 endif

```

FPGA Target

```

182 # Build FPGA host and afi executables and upload them into S3 bucket
183 .PHONY: afi
184 afi: check_afi_variables create_s3_dirs afigen
185 # wait_for_afi.py script is not working, need to be fixed by AWS
186 # . $(AWS_FPGA_REPO_DIR)/hdk_setup.sh; \
187 # wait_for_afi.py --afi $(shell cat *afi_id.txt | sed -n '2p' | tr -d ',' | sed 's /.*/') --notify --email $(EMAIL) &
188
189 .PHONY: check_afi_variables
190 check_afi_variables :
191 ifndef FPGA_BIN
192 $(error FPGA_BIN is not defined. Please specify a value for FPGA_BIN when invoking make afi.)
193 endif
194 ifndef HOST_BIN
195 $(error HOST_BIN is not defined. Please specify a value for HOST_BIN when invoking make afi.)
196 endif
197 ifeq ($(filter $(FPGA_BIN),$(VALID_AFI_NAMES)),)
198 $(error Invalid value for fpga afi name. Please use one of: $(VALID_AFI_NAMES))
199 endif
200 ifeq ($(filter $(HOST_BIN),$(VALID_HOST_BIN_NAMES)),)
201 $(error Invalid value for host executable for f1.xx instance. Please use one of: $(VALID_HOST_BIN_NAMES))
202 endif
203
204 .PHONY: afigen
205 afigen: fpga
206 @if ! aws s3 ls s3://$(S3_BUCKET_NAME) --region $(AWS_REGION) > /dev/null 2>&1; then \
207 aws s3 mb s3://$(S3_BUCKET_NAME) --region $(AWS_REGION); \
208 touch FILES_GO_HERE.txt; \
209 aws s3 cp FILES_GO_HERE.txt s3://$(S3_BUCKET_NAME)/$(S3_DCP_DIR)/; \
210 touch LOGS_FILES_GO_HERE.txt; \
211 aws s3 cp LOGS_FILES_GO_HERE.txt s3://$(S3_BUCKET_NAME)/$(S3_LOG_DIR)/; \
212 touch EXECUTABLES_GO_HERE.txt; \
213 aws s3 cp EXECUTABLES_GO_HERE.txt s3://$(S3_BUCKET_NAME)/$(S3_EXEC_DIR)/; \
214 fi
215 aws s3 ls s3://$(S3_BUCKET_NAME)/$(S3_BUCKET_PROJECT_FOLDER)
216 rm -rf to_aws
217 # Transform the .xclbin file to .awsxclbin:
218 $(VITIS_DIR)/tools/ create_vitis_afi .sh -xclbin=$(XCLBIN) -s3_bucket=$(S3_BUCKET_NAME) -s3_dcp_key=$(S3_DCP_DIR) -
    s3_logs_key=$(S3_LOG_DIR)
219 # Rename the fpga afi to the input name
220 cp runOnfpga.awsxclbin $(FPGA_BIN).awsxclbin
221 # Copy the executables host and .awsxclbin into the S3-bucket to be then executed on the f1 instance .
222 aws s3 cp $(HOST_EXE) s3://$(S3_EXE_BUCKET_NAME)/$(S3_HOST_EXEC_DIR)/
223 aws s3 cp $(FPGA_BIN).awsxclbin s3://$(S3_EXE_BUCKET_NAME)/$(S3_FPGA_AFI_DIR)/
224 # Rename the reports
225 mv fpga/reports/runOnfpga/hls_reports/runOnfpga_csynth.rpt $(FPGA_BIN)_runOnfpga_csynth.rpt
226 mv fpga/reports/runOnfpga/system_estimate_runOnfpga.txt $(FPGA_BIN)_system_estimate_runOnfpga.txt
227 mv fpga/logs/runOnfpga/runOnfpga_vitis_hls.log $(FPGA_BIN)_runOnfpga_vitis_hls.log
228 mv fpga/logs/runOnfpga/v++_log $(FPGA_BIN)_v++_log
229 # Copy the reports
230 aws s3 cp $(FPGA_BIN)_runOnfpga_csynth.rpt s3://$(S3_EXE_BUCKET_NAME)/$(S3_RESULTS_FPGA_REPORTS_DIR)/

```

```

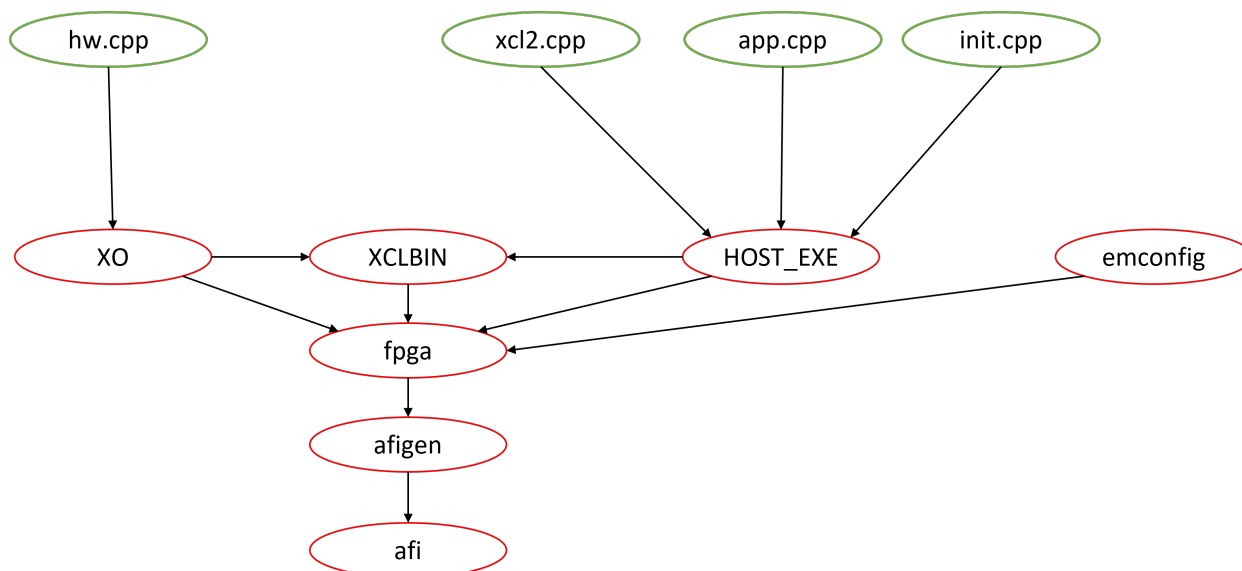
231 aws s3 cp $(FPGA_BIN)_system_estimate_runOnfpga.txt s3://$(S3_EXE_BUCKET_NAME)/$(S3_RESULTS_FPGA_REPORTS_DIR)
232 /
233 aws s3 cp $(FPGA_BIN)_runOnfpga_vitis_hls.log s3://$(S3_EXE_BUCKET_NAME)/$(S3_RESULTS_FPGA_REPORTS_DIR)/
234 aws s3 cp $(FPGA_BIN)_v++_log s3://$(S3_EXE_BUCKET_NAME)/$(S3_RESULTS_FPGA_REPORTS_DIR)/
235 .PHONY: emconfig
236 emconfig:
237 emconfigutil --platform $(PLATFORM)

```

The above 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.

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

Figure 3.7: Simplified Data dependency chart for generating AWSXCLBIN



This defines the command to run a specified script to run all the executables for CPU and FPGA for the replication of results in this paper.

```

240 # Execute CPU binaries: sequential and OpenMPI: make cpu_results M5N=<1x/1xBATCH1/1xBATCH2/1xBATCH3/4x/24x>
241     USE_AWS_S3_EXE=<yes/no>"
242 .PHONY: cpu_results
243 cpu_results :
244 sh ./common/util/generate_cpu_results.sh
245 # Execute FPGA designs: make fpga_results TABLE=<3/4/5/all> USE_AWS_S3_EXE=<yes/no>"
246 .PHONY: fpga_results
247 fpga_results :
248 sh ./common/util/generate_fpga_results.sh

```

Handy function to create S3 bucket.

```

253 # Create Permanent S3-bucket if not already created
254 .PHONY: create_s3_dirs

```

```

255 create_s3_dirs :
256 # Check if the AWS CLI is configured
257 @if ! aws configure get aws_access_key_id >/dev/null 2>&1; then \
258     echo "Error: AWS CLI is not configured. Please run 'aws configure' to set up your AWS credentials before running this
        target."; \
259     exit 1; \
260 fi
261
262 # Check if the bucket exists
263 @if ! aws s3 ls "s3://${S3_EXE_BUCKET_NAME}" --region "${AWS_REGION}"; then \
264     aws s3 mb "s3://${S3_EXE_BUCKET_NAME}" --region "${AWS_REGION}"; \
265 fi
266
267 # Create directories if they don't exist
268 aws s3api put-object --bucket "${S3_EXE_BUCKET_NAME}" --key "${S3_FPGA_AFL_DIR}/" --region "${AWS_REGION}" || true
269 aws s3api put-object --bucket "${S3_EXE_BUCKET_NAME}" --key "${S3_HOST_EXEC_DIR}/" --region "${AWS_REGION}" ||
    true
270 aws s3api put-object --bucket "${S3_EXE_BUCKET_NAME}" --key "${S3_CPU_DIR}/" --region "${AWS_REGION}" || true
271 aws s3api put-object --bucket "${S3_EXE_BUCKET_NAME}" --key "${S3_RESULTS_FPGA_REPORTS_DIR}/" --region "${
    AWS_REGION}" || true
272
273 @echo "S3-Bucket: Directories successfully created or they already exist."

```

Functions to clean the directories

```

182 # Clean
183 .PHONY: clean
184 clean :
185 $(RM) -rf *.csv *.jou *.run_summary *.dcp to_aws_x *.tar *.bin *.txt *.dSYM *.out
186 $(RM) *.o ~ rm -f $(HOST_EXE) $(CPU_EXE) $(OPENMPI_EXE)
187 $(RM) -rf *.log *.json *.xo .Xil/ .run/ runOnfpga* fpga/logs/ fpga/logs/link/* fpga/reports/* fpga/build/*
188
189 .PHONY: results_clean
190 results_clean :
191 $(RM) -rf ./ results /fpga/*.txt ./ results /fpga/*.csv ./ results /fpga/*.run_summary ./results/fpga/final_values/*.txt ./
    results /fpga/log_results/*.txt ./ results /cpu/*.txt ./ results /final_values/*.txt

```

3.9 Command Guidelines

3.9.1 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](#).

3.9.1.1 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.

3.9.1.2 Programming the device

- **Command:** [cl::Program::Binaries](#)
- **Description:**
- **Command:** [cl::Program](#)
- **Description:** Program interface that implements `cl_program`

3.9.1.3 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.

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

3.9.1.4 Kernels

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

3.9.1.5 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.

3.9.1.6 Events

- **Command:** [cl::Event](#)
- **Description:** Class interface for `cl_event`

3.9.1.7 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.

3.9.2 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));`

3.9.2.1 Computation Flow

3.9.3 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](#)

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 Source and binary files are used and distributed in respect to the terms specified in the copyright notice, *Copyright (c) 2018, Xilinx, Inc.*

4.1 Directory Structure

```
1 code/
2   Makefile
3   design.cfg
4   xrt.ini
5   common/
6       Constants.h
7       EventTimer.h
8       EventTimer.cpp
9       Utilities.cpp
10      Utilities.h
11   hls/
12       export_hls_kernel.sh
13       run_hls.tcl
14       MatrixMultiplication.h
15       MatrixMultiplication.cpp
16       Testbench.cpp
17   Host.cpp
```

4.2 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](#)

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

- `xrt.ini` defines the options necessary for *Vitis Analyzer*
- The `common` folder has header files and helper functions.
- 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.

4.2.1 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.

```

1 #include "Utilities .h"
2
3 // -----
4 // Main program
5 // -----
6 int main(int argc, char** argv)
7 {
8     // Initialize an event timer we'll use for monitoring the application
9     EventTimer timer;
10    // -----
11    // Step 1: Initialize the OpenCL environment
12    // -----
13    timer.add("OpenCL Initialization ");
14    cl_int err;
15    std::string binaryFile = argv[1];
16    unsigned fileBufSize ;
17    std::vector<cl::Device> devices = get_xilinx_devices () ;
18    devices.resize(1);
19    cl::Device device = devices[0];
20    cl::Context context(device, NULL, NULL, NULL, &err);
21    char* fileBuf = read_binary_file ( binaryFile , fileBufSize );
22    cl::Program::Binaries bins {{ fileBuf , fileBufSize }};
23    cl::Program program(context, devices, bins, NULL, &err);
24    cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE, &err);
25    cl::Kernel krnl_mmult(program, "mmult", &err);
26
27    // -----
28    // Step 2: Create buffers and initialize test values
29    // -----
30    timer.add("Allocate contiguous OpenCL buffers");
31    // Create the buffers and allocate memory
32    cl::Buffer in1_buf(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY, sizeof(matrix_type) * MATRIX_SIZE, NULL, &err);
33    cl::Buffer in2_buf(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY, sizeof(matrix_type) * MATRIX_SIZE, NULL, &err);
34    cl::Buffer out_buf_hw(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_WRITE_ONLY, sizeof(matrix_type) * MATRIX_SIZE, NULL, &err);
35
36    timer.add("Set kernel arguments");
37    // Map buffers to kernel arguments, thereby assigning them to specific device memory banks
38    krnl_mmult.setArg(0, in1_buf);
39    krnl_mmult.setArg(1, in2_buf);

```

```

40     krnl_mmult.setArg(2, out_buf_hw);
41
42     timer.add("Map buffers to userspace pointers");
43     // Map host-side buffer memory to user-space pointers
44     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) *
75         MATRIX_SIZE);
76     timer.finish();
77
78     // -----
79     // Step 4: Check Results and Release Allocated Resources
80     // -----
81     multiply_gold(in1, in2, out_sw);
82     bool match = Compare_matrices(out_sw, out_hw);
83     Destroy_matrix(out_sw);
84     delete[] fileBuf;
85     q.enqueueUnmapMemObject(in1_buf, in1);
86     q.enqueueUnmapMemObject(in2_buf, in2);
87     q.enqueueUnmapMemObject(out_buf_hw, out_hw);
88     q.finish();
89
90     std::cout << "----- Key execution times -----" << std::endl;
91     timer.print();
92
93     std::cout << "TEST " << (match ? "PASSED" : "FAILED") << std::endl;
94     return (match ? EXIT_SUCCESS : EXIT_FAILURE);
95 }

```

Listing 4.1: Host.cpp

4.2.2 MatrixMultiplication.cpp: the kernel

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

```

1 #include "MatrixMultiplication.h"
2
3 void mmult(const matrix_type Input_1[MATRIX_WIDTH * MATRIX_WIDTH],
4   const matrix_type Input_2[MATRIX_WIDTH * MATRIX_WIDTH],
5   matrix_type Output[MATRIX_WIDTH * MATRIX_WIDTH]) {
6   #pragma HLS INTERFACE m_axi port=Input_1 bundle=aximm1
7   #pragma HLS INTERFACE m_axi port=Input_2 bundle=aximm2
8   #pragma HLS INTERFACE m_axi port=Output bundle=aximm1
9   matrix_type Buffer_1[MATRIX_WIDTH][MATRIX_WIDTH];
10  matrix_type Buffer_2[MATRIX_WIDTH][MATRIX_WIDTH];
11
12  Init_loop_i: for (int i = 0; i < MATRIX_WIDTH; i++)
13    Init_loop_j: for (int j = 0; j < MATRIX_WIDTH; j++) {
14      Buffer_1[i][j] = Input_1[i * MATRIX_WIDTH + j];
15      Buffer_2[i][j] = Input_2[i * MATRIX_WIDTH + j];
16    }
17
18  Main_loop_i: for (int i = 0; i < MATRIX_WIDTH; i++)
19    Main_loop_j: for (int j = 0; j < MATRIX_WIDTH; j++) {
20      matrix_type Result = 0;
21      Main_loop_k: for (int k = 0; k < MATRIX_WIDTH; k++) {
22        Result += Buffer_1[i][k] * Buffer_2[k][j];
23      }
24      Output[i * MATRIX_WIDTH + j] = Result;
25    }
26 }

```

Listing 4.2: `MatrixMultiplication.cpp`

4.2.3 design.cfg: Compiler Flags

Defines several options for the *v++ compiler*. Learn more about it [here](#)

```

1 platform=xilinx_aws-vu9p-f1_shell-v04261818_201920_2
2 debug=1
3 profile_kernel =data: all : all : all
4 save-temps=1
5
6 [ connectivity ]
7 nk=mmult:1:mmult_1
8 sp=mmult_1.Input_1:DDR[1]
9 sp=mmult_1.Input_2:DDR[2]
10 sp=mmult_1.Output:DDR[1]

```

Listing 4.3: `design.cfg`

4.2.4 xrt.ini: Vitis Analyzer

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

```
INFO: Loading mmult.xclbin
----- Key execution times -----
OpenCL Initialization                : 83.500 ms
Allocate contiguous OpenCL buffers   : 0.043 ms
Set kernel arguments                 : 0.164 ms
Map buffers to userspace pointers    : 1.058 ms
Populating buffer inputs             : 0.119 ms
Set kernel arguments                 : 0.020 ms
Memory object migration enqueue host->device : 0.255 ms
Launch mmult kernel                  : 0.130 ms
Wait for mmult kernel to finish running : 1.385 ms
Read back computation results (implicit device->host migration) : 0.169 ms
TEST PASSED
[centos@ip-172-31-4-76 hw5]$
```

Figure 4.1: CPU Implementation

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

4.3 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.

```
1 # compile
2 source $AWS_FPGA_REPO_DIR/vitis_setup.sh
3 export PLATFORM_REPO_PATHS=$(dirname $AWS_PLATFORM)
4 make all TARGET=sw_emu
5
6 # run
7 source $AWS_FPGA_REPO_DIR/vitis_runtime_setup.sh
8 export XCL_EMULATION_MODE=sw_emu
9 ./host mmult.xclbin
```

The latency is **86.93ms** 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).

4.4 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.

4.5 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 testbench 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 4.2 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 4.2: Testbench Console

4.6 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.

Property	Value
Line Number	22
Name	mul
Opcode	fmul
Op Latency	1
RTL Name	fmul_32ns_32ns_32_2_max_dsp_1_U2
Source File	hls/MatrixMultiplication.cpp
Topo Index	74

Figure 4.3: *Scheduler View*

4.6.1 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.

4.6.2 Resources

Table 4.1 reports the resource utilization.

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

Table 4.1: Resource Utilization

4.6.3 Scheduler View

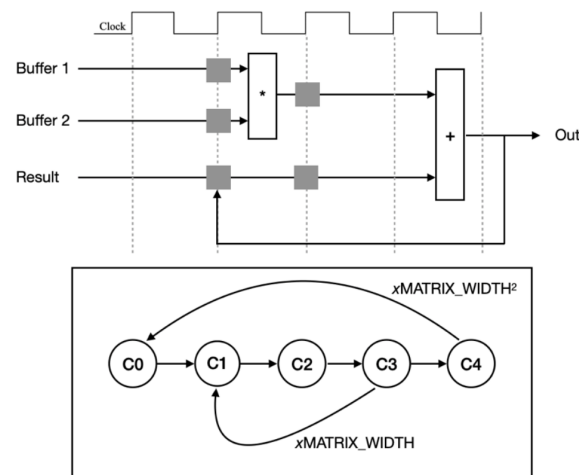
Use the *Scheduler View* under the *Analysis Perspective* to analyze how the computations are scheduled in time. From the *Scheduler View* it appears that the multiplication takes 1 cycle (Figure 4.4)

4.6.4 Data Flow

Dataflow and FSM diagram for main loop of *MatrixMultiplication.cpp*

4.7 HLS Kernel Optimization: Loop Unrolling

- Go back to the *Synthesis perspective*

Figure 4.4: *Scheduler View*

- 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 4.4: `MatrixMultiplication.cpp` with `#pragma HLS UNROLL`

For other examples see [here](#).

- 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 4.5: `MatrixMultiplication.cpp`

4.7.1 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!

4.7.2 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.

4.7.2.1 Change the clock

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

4.7.2.2 Resources

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

Table 4.2: 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.

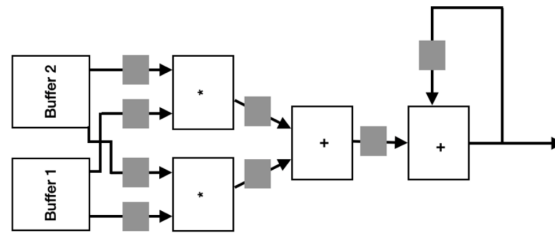
4.8 HLS Kernel Optimization: Pipelining

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.

4.8.1 Understanding the Initiation Interval (II)

Note the initiation interval is 32 for the pipelined loop `j`. To understand this result, Figure 4.5 draws a schematic for the data path of `Main_loop_j` and shows how it is connected to

Figure 4.5: *Scheduler View*

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 ports 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.

4.8.2 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](#).

4.8.3 Export the Vitis Kernel

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

```

1 #include "MatrixMultiplication.h"
2
3 void mmult(const matrix_type Input_1[MATRIX_WIDTH * MATRIX_WIDTH],
4     const matrix_type Input_2[MATRIX_WIDTH * MATRIX_WIDTH],
5     matrix_type Output[MATRIX_WIDTH * MATRIX_WIDTH]) {
6     #pragma HLS INTERFACE m_axi port=Input_1 bundle=aximm1
7     #pragma HLS INTERFACE m_axi port=Input_2 bundle=aximm2
8     #pragma HLS INTERFACE m_axi port=Output bundle=aximm1
9     matrix_type Buffer_1[MATRIX_WIDTH][MATRIX_WIDTH];
10    matrix_type Buffer_2[MATRIX_WIDTH][MATRIX_WIDTH];
11
12    #pragma HLS ARRAY_PARTITION variable=Buffer_1 complete dim=2
13    #pragma HLS ARRAY_PARTITION variable=Buffer_2 complete dim=1
14
15    Init_loop_i : for (int i = 0; i < MATRIX_WIDTH; i++)
16        Init_loop_j : for (int j = 0; j < MATRIX_WIDTH; j++) {

```

```

17   Buffer_1[i][j] = Input_1[i * MATRIX_WIDTH + j];
18   Buffer_2[i][j] = Input_2[i * MATRIX_WIDTH + j];
19   }
20
21   Main_loop_i: for (int i = 0; i < MATRIX_WIDTH; i++)
22   Main_loop_j: for (int j = 0; j < MATRIX_WIDTH; j++) {
23       #pragma HLS PIPELINE II=1
24       matrix_type Result = 0;
25       Main_loop_k: for (int k = 0; k < MATRIX_WIDTH; k++) {
26           Result += Buffer_1[i][k] * Buffer_2[k][j];
27       }
28       Output[i * MATRIX_WIDTH + j] = Result;
29   }
30 }

```

Listing 4.6: `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.

```

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

```

4.9 Run on the FPGA

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

```

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

```

You should see the following files generated when you ran:

```

1 profile_summary.csv
2 timeline_trace.csv
3 xclbin.run_summary

```

Listing 4.7: 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.

4.10 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.

