Center for Embedded and Cyber-Physical Systems
University of California, Irvine

A Scalable SystemC Model of a Checkerboard Grid of Processing Cells

Yutong Wang, Rainer Dömer

Technical Report CECS TR 22-03
September 30, 2022

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

yutongw5@uci.edu
http://www.cecs.uci.edu

A Scalable SystemC Model of a Checkerboard Grid of Processing Cells

Yutong Wang, Rainer Dömer

Technical Report CECS TR 22-03
September 30, 2022

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

yutongw5@uci.edu
http://www.cecs.uci.edu

Abstract

Ever since the introduction of embedded computers, embedded computing systems have had a great impact on our daily life and changed the way the whole society operates. Embedded computers today have gained many functionalities and massive amount of computing power. Further more, embedded computing systems can be integrated into System-on-Chip (SoC) which usually contains many processing cores and some of them capable of rendering graphics for high resolution screen. As SoCs get more and more complex, scalability becomes a series issue: it is harder and harder to fit more computing units, memories and other component in a single chip while maintaining scalable performance.

In this work [1], we introduce the "Checkerboard" Grid of Processing Cells (GPC) architecture model, which is designed to be a scalable and stable platform without sacrificing scalable performance. This work simulates and evaluates the scalability of Checkerboard SystemC model with a Mandelbrot Set Visualization application – an embarrassingly parallel program that calculates and visualizes the Mandelbrot set with given parameters.

Contents

1 Introduction **1**
- 1.1 Related Work . . . 1
- 1.2 Background . . . 2

2 Checkerboard Grid of Processing Cells Architecture **2**
- 2.1 Example Checkerboard 4x4 Model Explained . . . 3
- 2.2 Core-Memory Communication and Checkerboard Address Space . . . 10
- 2.3 Features and Functionalities of Checkerboard Model . . . 14

3 The Mandelbrot Set and Mandelbrot Visualization Application **15**
- 3.1 Definition of the Mandelbrot Set . . . 15
- 3.2 Base Model of Mandelbrot Set Visualization . . . 15
  - 3.2.1 The Base Model of Mandelbrot Set Visualization Explained . . . 17
  - 3.2.2 Functionalities of the Base Model of Mandelbrot Set Visualization . . . 22
  - 3.2.3 Limitations of the Base Model . . . 24
- 3.3 Version 1.0 of Mandelbrot Set Visualization . . . 24
  - 3.3.1 1.0 Model of Mandelbrot Set Visualization Explained . . . 25
  - 3.3.2 Functionalities of the 1.0 Model of Mandelbrot Set Visualization . . . 27
  - 3.3.3 Scalability and Timing of the 1.0 Model of Mandelbrot Set Visualization . . . 28

4 Mapping of Mandelbrot Set Visualization onto Checkerboard Model **28**
- 4.1 Example Mandelbrot on Checkerboard 4x4 Explained . . . 28
- 4.2 Functionalities of Scalable GPC Mandelbrot Model . . . 33

5 Experimental Results **35**
- 5.1 Experimental Setups . . . 35
- 5.2 Data Collected . . . 36

6 Conclusion and Future Work **41**

References **42**

A Scalable SystemC Model of a Checkerboard Grid of Processing Cells

Y. Wang, R. Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
yutongw5@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

Ever since the introduction of embedded computers, embedded computing systems have had a great impact on our daily life and changed the way the whole society operates. Embedded computers today have gained many functionalities and massive amount of computing power. Further more, embedded computing systems can be integrated into System-on-Chip (SoC) which usually contains many processing cores and some of them capable of rendering graphics for high resolution screen. As SoCs get more and more complex, scalability becomes a series issue: it is harder and harder to fit more computing units, memories and other component in a single chip while maintaining scalable performance.
In this work [1], we introduce the "Checkerboard" Grid of Processing Cells (GPC) architecture model, which is designed to be a scalable and stable platform without sacrificing scalable performance. This work simulates and evaluates the scalability of Checkerboard SystemC model with a Mandelbrot Set Visualization application – an embarrassingly parallel program that calculates and visualizes the Mandelbrot set with given parameters.

1 Introduction

The **"Checkerboard", Grid of Processing Cells (GPC)** model in this paper is a on-going System-on-Chip design developed with SystemC [2] release 2.3.3. Checkerboard is one of the many designs of Grid of Processing Cells [3] and is the model designed and evaluated in SystemC TLM-2.0 in this work [1]. Checkerboard is designed with scalability and ease-of-use in mind. In order to properly analyze the scalability of Checkerboard architecture, the software chosen to be mapped onto it is the Visualization of Mandelbrot Set [4]. This program is introduced in Section 2. Mandelbrot on Checkerboard model is compared with an ideal SystemC model (Mandelbrot Set Visualization 1.0 model) with near perfect scalability to show that the Checkerboard model, when mapped with a scalable software, is also scalable.

1.1 Related Work

Many computer architectures have been proposed and used throughout the years. The classic von-Neumann computer architecture [5] has one memory bus between memory and central processing unit (CPU). The original Harvard architecture [6] and its modern implementation, the modified Harvard architecture [7], all use single shared bus to a main memory. While modern computers typically organized as symmetric multiprocessors (SMPs) [8], there is only a single shared memory connected via a bus interface. Having a single

memory with a shared bus for CPU(s) limits the scalability of these architectures.

Knowing that the architectures with single memory bus has limited scalability, other architectures have been proposed with better scalability. The Raw architecture [9] is a 4x4 tiled architecture designed with scalability in mind. It allows application-specific resource allocation and data flow within the chip. The tiled architecture of Raw also allows it to scale with increasing silicon density [9]. Another scalable architecture is the Tile Processor [10][11]. The Tile Processor's implementations, TILE64 processor and TILEPro64 processor, are both manufactured by Tilera. Both processors show the scalability of the Tiled architecture. With each tile contains a general purpose processor, cache, and a router, TILE64 and TILEPro64 tiles are able to communicate with each other and other input/output interfaces on a very large 8x8 scale [12][13][14]. Intel's Teraflops Research Chip (codenamed Polaris), with a network-on-chip architecture, is another scalable manycore design [15]. Polaris consists of a 10x8 2D mesh network (80 Cores) with a sustained performance of 1.28 teraFLOPS, demonstrating very good scalability of the Polaris' architecture [16]. Intel's Single-Chip Cloud Computer (SCC) is another tiled architecture that communicate through architecture similar to a cloud computer in data center. The chip contains tiles in a 4x6 2D-mesh with 2 P54C Pentium cores and a router in each tile. Intel hopes to make SCC scale to 100+ cores by having each chip communicate to another chip [17][18]. KiloCore processor array, which contains 1000 independent processors and 12 memory modules on a single chip, is another scalable tiled-like architecture [19]. Their data indicates that under most conditions, the processor array has a near-optimal proportional scaling of power dissipation. The Checkerboard GPC architecture, like Raw, Tilera, Polaris, SCC and KiloCore, is also a scalable architecture. The Checkerboard SystemC model, which supports varying layout from 1x1 all the way up to 16x16, adds flexibility on the prior works. There are two more GPC variants proposed, which are hierarchical GPC, and 3D GPC [3]. In this work [1], only the Checkerboard GPC will be introduced.

1.2 Background

A scalable architecture, according to definition, is an architecture that can linearly scale up to meet increased work loads. In other words, if the work load is increased to exceed the current work capacity of the system, a scalable system should be able to scale up the system to match the increased work load. As the system gets scaled up by a factor of x times, if the system can also process x times the amount of the original work load, then this machine is said to have "linear scalability".

Linear scalability, however, is not usually the case when it comes to computer systems. One cannot simply double the amount of processing units and expect the system to work twice as fast to handle double the load. Very often there is overhead when system scales up. The scalability of that system depends on how much overhead there exist in both software and hardware. If a hardware is designed perfectly to scale linearly but the software application does not scale at all, the system then would still have no scalability.

In this work [1], to test out the scalability of Checkerboard architecture, which is a hardware design, the software must be scalable. In other words, when the hardware scales up, the software mapped can properly reflect the effect of different sizes of the Checkerboard architecture. Therefore, on the software side, the Mandelbrot Set Visualization application was chosen as a benchmark that gets mapped on to Checkerboard platform in order to show the scalability of the architecture.

2 Checkerboard Grid of Processing Cells Architecture

In this chapter, Checkerboard Grid of Processing Cells model (Checkerboard for short) is explained in detail with a top-down approach. The Grid of Processing Cells is proposed by Rainer Dömer and discussed with Arya Daroui, Vivek Govindasamy, Yutong Wang at University of California, Irvine [3]. Although Checkerboard model goes through many versions, only the latest version of the Checkerboard model will be used to

explain the functionalities of the Checkerboard model.

2.1 Example Checkerboard 4x4 Model Explained

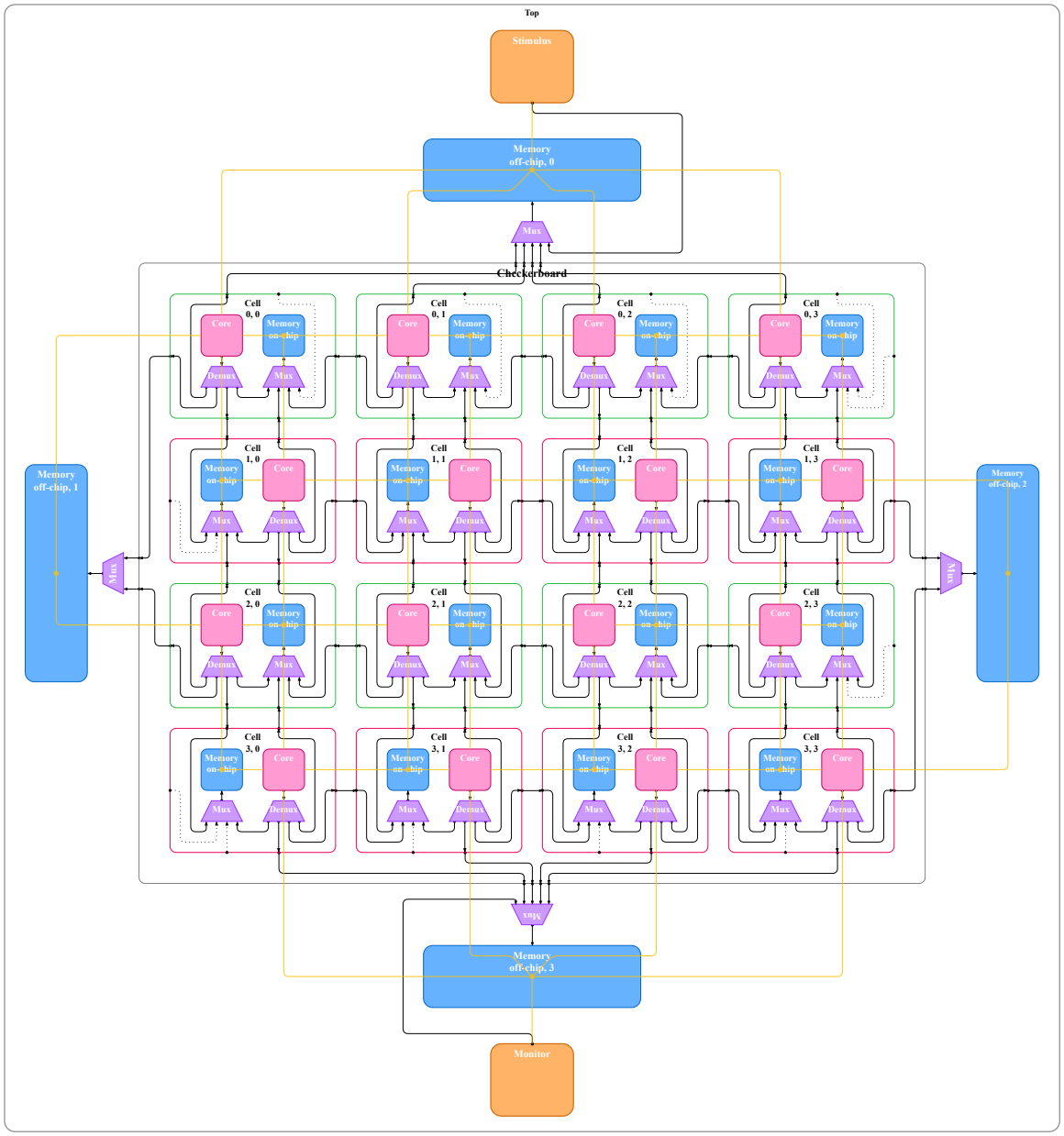

Figure 1: High-level Schematic of Checkerboard Model [20]

This section includes a high-level schematic of an example of Checkerboard Grid of Processing Cells model with a 4 by 4 setup and detailed top-down explanation of the Checkerboard model. There are other

configurations for the Checkerboard model such as 3x3 and 2x4 because the scalability of the Checkerboard model. For time and space constraints, only the 4x4 Checkerboard example will be explained in detail in this section. Some sections involve SystemC knowledge and cannot be explained in detail in this paper. All SystemC related documents can be found on SystemC official reference page at http://www.systemc.org [2].

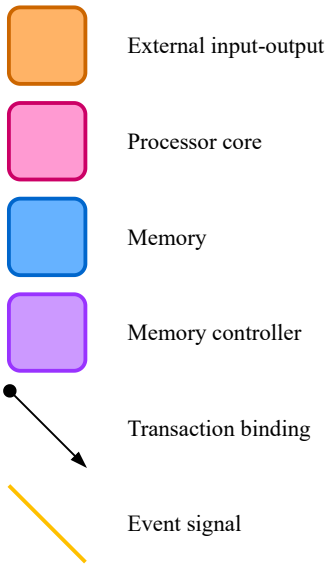

Figure 2: Legend for Schematic of Checkerboard Model [20]

Starting from the largest model, which is named "Top" in figure 1. Module Top contains every other model in the Checkerboard model, including user defined Stimulus and Monitor, off-chip memories, multiplexers (mux) for off-chip memories and the Checkerboard module itself. Inside checkerboard, there are more modules including Cores, on-chip memories, demultiplexers (demux) for Cores and multiplexers for on-chip memories. Each module is connected with TLM-2.0 style sockets which is the same socket pair introduced in section 3.3.1

The next group of modules just below the Top model are Stimulus, Monitor, off-chip memories, multiplexers for off-chip memories and the checkerboard module. All these modules can be configured in user file, the user have full control on what type of off-chip memory they can use and the number of sockets a memory can have. Same rules apply for the off-chip memory multiplexers that routes the memory read and write requests from the border units inside checkerboard. If the user does not provide their customized off-chip memory, a default one-port memory is provided with all sockets connected. A alternative way to connect the Stimulus and Monitor other than routing their requests through the off-chip multiplexers is to add another target socket to the provided off-chip memory and connect the Stimulus and Monitor. The files in Checkerboard repository does not include Stimulus and Monitor code. The user is responsible for designing and implementing these two modules.

The Stimulus and Monitor module's functionalities are completely up to the user to implement, just like the Stimulus and Monitor in section 3.2 and section 3.3. As mentioned above, this 4x4 setup is only an example, the user of the Checkerboard model can connect the Stimulus and Monitor in anyway they want, or the user can choose to completely not use these two modules and work with a bare-bones version of the

Checkerboard model.

The multiplexers on the outskirts between off-chip memories and the checkerboard module is also a configurable component in the user file, the configuration is done automatically in the Python code generator (which is introduced in Section 2.3). Each multiplexers can have multiple input sockets connected to the border cells inside checkerboard module (shown as the purple trapezoid on the outside of Checkerboard module in Figure 1). For example, the off-chip multiplexer on top would have 4 target sockets connected to 4 border cells on top inside checkerboard module and 1 extra target socket connected to the Stimulus. There is also one initiator socket connected to the target socket in the off-chip memory modules. The purpose of these off-chip multiplexers is to route incoming read and write requests from the connected cells to the connected off-chip memory (memories). The off-chip multiplexers are created with C++ template and therefore can have different number of sockets during initialization phase:

```
template <int Number_of_Sockets>
class DRAM_MemMux_N: public sc_module{
...
}
....
class TOP: public sc_module{
public:
        DRAM_MemMux_N<4> dram_memmux_up,dram_memmux_down;
        DRAM_MemMux_N<2> dram_memmux_left,dram_memmux_right;
...
}
```

In this case, because there are 4 sockets on the top and down off-chip muxes and 2 sockets on the left and right off-chip muxes, the number 4 and 2 are used here.

All four off-chip memories are TLM-2.0 style single socket memories with basic read and write functionalities, shown as blue rectangles on the outside of Checkerboard module in Figure 1. The user can specify their own off-chip memories and even memory buses in the `checkerboard_user` file if they wish. The example 1 socket memories are provided with minimum functionalities. Replacement of these default off-chip memories can be easily done by the user, but this process will not be explained in detail here, please go check the user manual of the Checkerboard Model. Each off-chip memory can be configured with a different size (this size must be smaller than the maximum size allowed for off-chip memories), also different read and write delays:

```
#ifndef OFF_CHIP_MEMORY_SIZE
#define OFF_CHIP_MEMORY_SIZE                OFF_CHIP_MEMORY_SIZE_MAX
#endif

#ifndef OFF_CHIP_MEM_READ_ACCESS_DELAY
#define OFF_CHIP_MEM_READ_ACCESS_DELAY  sc_time(0, SC_NS)
#endif
#ifndef OFF_CHIP_MEM_WRITE_ACCESS_DELAY
#define OFF_CHIP_MEM_WRITE_ACCESS_DELAY sc_time(0, SC_NS)
#endif
```

The maximum value of off-chip memory size is set to be 0x20000000 because of the way how Checkerboard's address mapping works.

As mentioned above, almost every component under module Top is configurable, the multiplexers, off-chip memory, and of course, the checkerboard module itself. The user can choose the dimensions of the

checkerboard module shown in the middle of figure 1. In this example, the dimensions are set to be 4 cells wide and 4 cells high. Every connection inside the Checkerboard module is done automatically once given the proper parameters during compilation stage. Details on how that is done is explained in later sections. The connections to the off-chip memory will also change automatically according to the height and width of the checkerboard module.

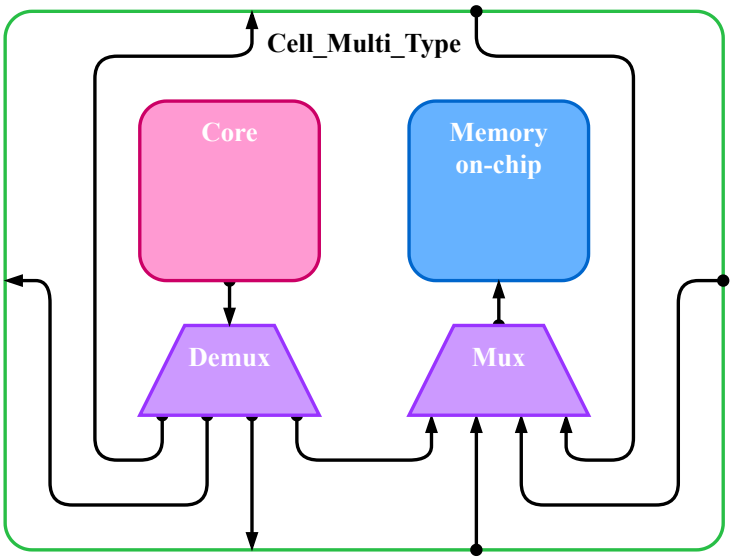

Figure 3: High-level Schematic of a Generic Multi-Type Cell Inside Checkerboard with Core & Core Demux on Left and Memory & Mem Mux on Right, figure modified from [20]

Each cell, marked as transparent box with green and red borders in Figure 1, contains 1 core module, 1 core demultiplexer (demux), 1 memory module, and 1 memory multiplexer (mux) as shown in Figure 3, . The structure Cell does not have any actual functionalities, its sole purpose it to serve as a container for the core and core demux, memory and memory mux. Having the Cell module keeps the Checkerboard clean on a hierarchal level and makes the code easer to read. Generally, the Checkerboard contains 2 high-level types of cell: one type have the Core module and Core Demux on the left side and Memory and Mem Mux on the right side, as shown in Figure 3; The other type have Core module and Core Demux on the right side and Memory and Mem Mux on the left side, shown in Figure 4. Having 2 types of major cell layouts allows each core have access to 4 adjacent memories without crossing wires (the adjacent memories include off-chip memories on the outside of Checkerboard module). It is possible to use only 1 layout (either Core left Mem right or Core right Mem left) to ensure that each core have access to 4 adjacent memories. However, it would require the wires to be crossed inside cell or between cells. This is another topic that can be researched, but it would not be included in this paper. Every connection inside the cell is done in TLM-2.0 style SystemC initiator and target socket as mentioned in previous chapters.

Inside each cell, there are always 1 Core, 1 Memory, 1 Core Demux, 1 MemMux. Each Core will always have access to four memories, either that memory is inside another neighboring cell or the off-chip memory. Within that four memories that a Core have access to, one of them is local to that core, that is, that local memory is inside the same cell as that Core. It is unspecified at this point if a local core should have higher access speed for reading and writing because Cells exists for only hierarchical purpose and does not reflect

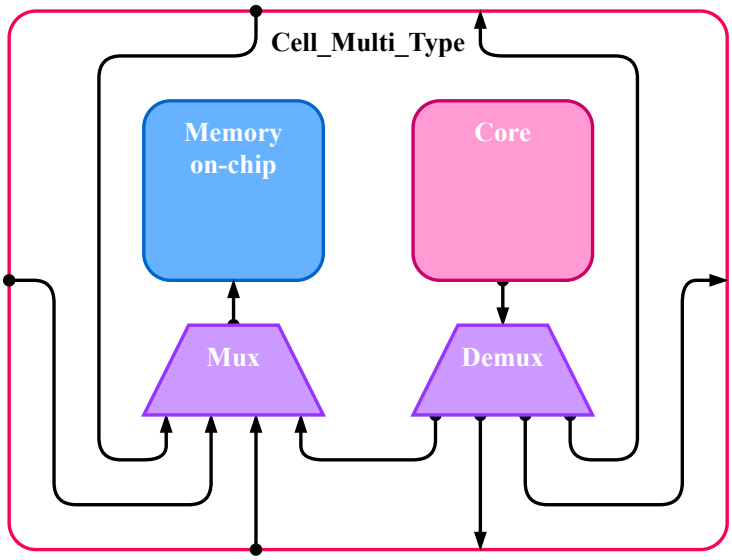

Figure 4: High-level Schematic of a Generic Multi-Type Cell Inside Checkerboard with Core & Core Demux on Right and Memory & Mem Mux on Left, figure modified from [20]

actual physical property (for example distance and latency between cells). Regardless if a Cell have its Core and Core Demux on the left or right side, as mentioned before, a Core always have access to four memories, and that is done via the Core Demux. A core with only 1 socket is connected to a Core Demux, the Core Demux then takes the request from that core and forward it to different connected memories based on the address of the Core and the address of the payload. The detailed memory addressing will be explained in detail in later section.

While the Core will always have access to 4 memories, each on-chip Memory does not necessarily have 4 connected Cores. Each on-chip memory has one socket connected to the memory Mux and in generic cases there would be 4 connections from both inside and outside the memory. For example, if a cell is in the middle of Checkerboard, then inside that Cell, the on-chip Memory will have 1 connection from its local Core and 3 other connection from its neighboring Cores. If that Cell is on the outer border of the Checkerboard, then depends on the location of that Cell and the location of on-chip Memory (whether it's on the left or right side of the Cell), some Memories will have less than 4 connected Cores. For example the Cell03 inside this 4x4 example checkerboard will be missing incoming connections from the top and the right because that on-chip Memory does not have adjacent Core from top and right, as shown in Figure 5.

In addition to the missing connections going inside Cells, each border Cells would have outgoing connections to different off-chip Memory Muxes. This is tedious and error prone if done by hand. Therefore the Checkerboard architecture code takes care of the connectivity completely for both inside and outside of a Cell. Checkerboard program will automatically instantiate needed sockets for the Core Demuxes and Memory Muxes and bind them depends on the x and y ID and the size of the Checkerboard. The user will only need to specify the Height and Width of the Checkerboard and everything will automatically connect on the architecture side:

```
#ifndef GRID_HEIGHT
#define GRID_HEIGHT 4
```

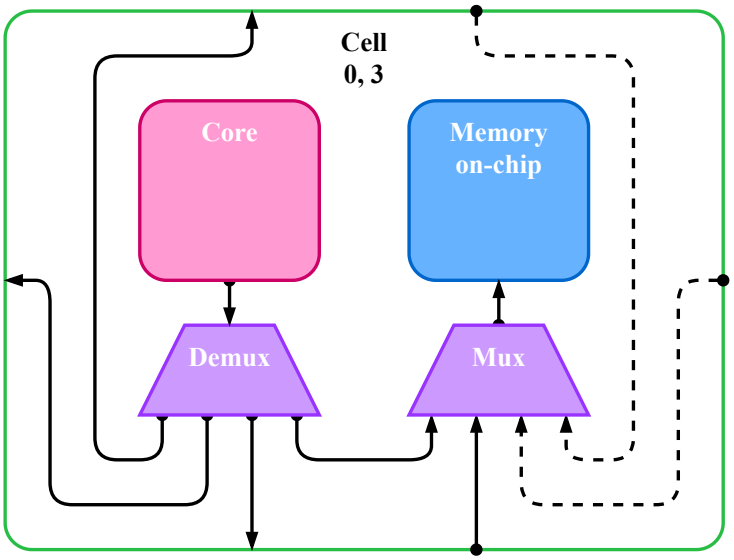

Figure 5: High-level Schematic of Corner Cell03 Inside Checkerboard without Adjacent Core on Top and Right, figure modified from [20]

```
#endif
#ifndef GRID_WIDTH
#define GRID_WIDTH 4
#endif
```

If no parameters are provided during compile time, then the program will default to a 4x4 Checkerboard. The maximum width and height that is supported by the Checkerboard Model is 16x16 due to current constraints in address space. This will be explained in detail in a later section. Single row cases are special because Cores then have access to both the top off-chip memory and bottom off-chip memory. In the case of a single Core (1x1 Checkerboard), that one Core has access to top, left and bottom off-chip memories. Both mentioned special cases are taken care of in the Checkerboard program and are fully tested.

From the user's perspective, each Core inside a Cell should have its own functionalities, whether it is a complex algorithm or just simply passes value from its left neighbor to its right neighbor. Inside the user file, each Core with it unique ID is extended from a universal Core class that allows every user Core have a different main (and also other functions). Before Checkerboard Version 1.3, this process of creating user Cores and passing id and signals have to be done manually by the user. This is again, very repetitive work, especially when the dimensions of the Checkerboard model grow. In version 1.3 of Checkerboard model, an auto code generator for the user file is introduced to take care of the creation of each User Core and passing the parameters during instantiation. Details on the code generator follow in Section 2.3. An example User Core is shown here:

```
class Core00: public Core{
public:
        void main(void);
```

```
        Core00(sc_module_name n,
                                int _y,
                                int _x,
                                sc_event &_S_UP,
                                sc_event &_S_LEFT,
                                sc_event &_S_RIGHT,
                                sc_event &_S_DOWN)
                            : Core(n,_y,_x,_S_UP,_S_LEFT,_S_RIGHT,_S_DOWN){
                SC_THREAD(main);
        }
};

....

void Core00::main(){
        //Your code here
}
```

y and x representing the location of that Core (in this case, location (0, 0) in Checkerboard); the `sc_events` used to listen to its connected memories so when there is a need to pass message to other Cores, there is a way to notify other Cores. On top of that, each User Core comes with a `main` function that is completely up to the user to fill. The user can of course add other functions to each Core to make it more sophisticated. The `main` function is first declared inside a User Core class and defined later so if the user choose to, they can relocate the `main` functions of each User Core to a different file, which will give the user file better readability.

There is one more element on Figure 1 left untouched and that is the yellow lines that goes across Cells. As the legends for the schematic indicate, these yellow lines are the event signals used to notify cores that a nearby memory has been accessed. All events are SystemC standard class named `sc_event` and detailed information can be found on official SystemC reference page. Events for on-chip memories inside Cells are instantiated under Checkerboard module level and therefore able to pass through multiple cells (think of each event as a interrupt signal). Events for the off-chip memories are instantiated under module TOP and therefore can be passed to module Checkerboard and then Cells and Cores. As mentioned above, each User Core, when instantiating, need multiple parameters (as shown below in the constructor for Core):

```
Core(sc_module_name n,
                                int _y,
                                int _x,
                                sc_event &_S_UP,
                                sc_event &_S_LEFT,
                                sc_event &_S_RIGHT,
                                sc_event &_S_DOWN);
```

the four events "S UP, S LEFT, S RIGHT, S DOWN" are for memories that each Core has access to. Recall that every Core, does not matter where that Core is in Checkerboard, has access to 4 memories, therefore 4 signals representing all 4 memories. An example for this 4x4 example Checkerboard would be when Core11 writes a message to its local memory (memory 11), Core11 will also notify that event for memory11. Because that event is passed to the neighbors of Core11, Core01 (top neighbor), Core10 (left neighbor), Core11 (self), Core21 (bottom neighbor), all will be notified by that event.

There are alternative ways to notify other Cores when reading and writing data to a memory. One common approach is to have every Core listen to every event. However, that method will require each Core having

16 parameters in the constructor and listening to 16 events for only a 4x4 Checkerboard. That number will soon increase to a unmanageable amount when the width and height of Checkerboard increases. Besides that, having every Core also listening to every event is also inaccurate in terms of actual hardware, a Core on the top left corner in a large Checkerboard grid should not have access to signals from across the chip.

Cores on the outer border of Checkerboard will also listen to events from the off-chip memories. The only difference is that the events for on-chip memories can be signaled by only the neighbors of that Memory while the events for off-chip memories can be signaled from every core that have access to that off-chip memory (they do not need to be neighbors). Before version 1.3 of Checkerboard model, this event mapping is done manually by the user, after version 1.3, this is done automatically with the code generator (details in Section 2.3. This memory-event system allows each Core to read/write to every memory it has access to and also notify all other Core that have access to that specific memory, thus making communication between different Cores across Checkerboard possible.

2.2 Core-Memory Communication and Checkerboard Address Space

As described in section 2.1 that when a Core sends read or write request to a specific memory address, the payload (request) will first get sent to the Core Demultiplexer (Core Demux). It is up to the Core Demux to forward the payload to the correct memory based on the requested address. Once the memory receives request from a Core, the memory will respond to that request, whether it is a read or write. When a Core is initialization a request, a helper function called MemAccess is used:

```
void Core::MemAccess(
        tlm::tlm_command cmd,
        tlm::tlm_initiator_socket<> &s,
        sc_dt::uint64 addr,
        void *data,
        unsigned int len){
        ..... (check for OK response, add delay...)
}
```

This helper function is a nice indication of the basic component of a payload: tlm command, tlm initiator socket, an address, pointer to data, and the length of the data. After calling this MemAccess function and everything is filled, this payload is first send to the Core's connected Core Demux for forwarding. Once the payload is received by Core Demux, it will first be masked to only 32 bit (the current Checkerboard model is using 32 bit address space, but TLM2 payload is 64bit) with this:

```
        sc_dt::uint64    adr = trans.get_address();
        // (reference: SystemC_Part3.pdf, slide 31)
        adr = adr & 0xFFFFFFFF; //make sure address is 32 bits
```

This masking process is necessary because when a 32 bit address is assigned to this 64 bit payload and the most significant bit is a 1, the address will get sign extended with 1s, which then breaks the forwarding logic without masking it to 32 bit.

The following image shows the addressing for a 4x4 Checkerboard used in previous section: As shown in Figure 6, the address for both on-chip and off-chip memories are 32 bit wide. Starting with off-chip memory. The most significant bit (the 31st) bit will always be 1, this is to differentiate the on-chip and off-chip memories. When the most significant bit is a 1 from the payload, the Core Demux know to send that payload to one of the off-chip memories. Because there are always 4 off-chip memories on the outside of Checkerboard module (can be configured to have different numbers of off-chip memories by user), 2 extra bits are needed to identify each off-chip memory. The 2 bits used will be the 30th and 29th bit in the address

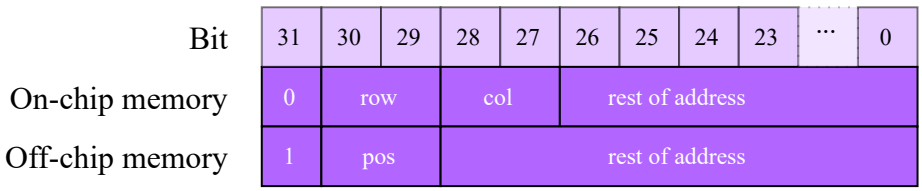

Figure 6: Example Addressing for 4x4 Checkerboard

since 2 bits are enough to represent 4 unique addresses. Top off-chip memory is 00, left off-chip memory is 01, right off-chip memory is 10, and bottom off-chip memory is 11.

```
if(adr>>31){//global
        switch (adr>>29) {
                case 0b100:
                        trans.set_address(adr & 0x1FFFFFFF);
                        out_up->b_transport(trans,delay);
                        break;
                case 0b101:
                        trans.set_address(adr & 0x1FFFFFFF);
                        out_left->b_transport(trans,delay);
                        break;
                case 0b110:
                        trans.set_address(adr & 0x1FFFFFFF);
                        out_right->b_transport(trans,delay);
                        break;
                case 0b111:
                        trans.set_address(adr & 0x1FFFFFFF);
                        out_down->b_transport(trans,delay);
                        break;
        }
}else{//local
        ......
```

The partial code from the Core Demux shows that it will first check the most significant bit, and then route the payload to the off-chip memory depends on the address bits (30th and 29th) bit. The payload address will also need to be masked to get rid of the top 3 bits used only by the routing, the memories do not see these bits. Because the left most 3 bits are used for every off-chip memory, the maximum memory size for off-chip memories is limited to 0x20000000 with 29 bits available for data entires. The user can set their own memory sizes but if that size is larger than the maximum size, the program will report an error.

On-chip memories on the other hand is more complicated because the numbers of on-chip memories changes depending on the Height and Width of the Checkerboard. The 4x4 example on-chip memory addresses displayed in figure 6 has one bit to distinguish on-chip and off-chip memories, 4 more bits for the x and y ID of the memory – 2 bits for row number and 2 bits for column number. Because there are 4 Cells per row and 4 rows in this 4x4 example, 2 bits are needed for the x and y memory ID. Similar to the off-chip memories, because the left most 5 bits are used to identify each on-chip memory, the maximum memory size

of then limited to 0x08000000 with 27bits available for data entires. The user can also set their own memory size for on-chip memory but if that size is larger than the maximum size allowed, the program will report an error. For generic Checkerboard that have Width and Height defined by the user (maximum width and height

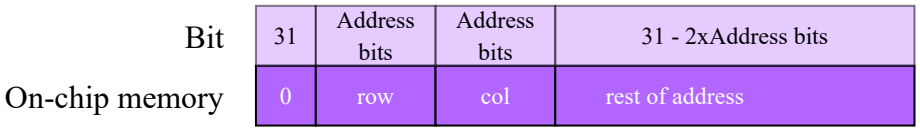

Figure 7: Example Addressing for Generic Checkerboard

is 16x16), this 5-bit scheme will not work. If the Checkerboard is 4x5 then there will be 1 column of on-chip memories without address and can never be accessed. In version1.2 and later Checkerboard model, instead of asking the user to edit the forwarding logic and addresses for on-chip memories, a variable called "Address Bits" is used to automate on-chip memory addressing and payload forwarding in Core Demux. The Address Bits are set in a series of C++ preprocessor macros:

```
#if GRID_WIDTH > 8 || GRID_HEIGHT > 8
#define ADDRESS_BITS 4
#define ON_CHIP_MEMORY_SIZE_MAX 0x00800000        // 23bits available
#elif GRID_WIDTH > 4 || GRID_HEIGHT > 4
#define ADDRESS_BITS 3
#define ON_CHIP_MEMORY_SIZE_MAX 0x02000000        // 25bits available
#elif GRID_WIDTH > 2 || GRID_HEIGHT > 2
#define ADDRESS_BITS 2
#define ON_CHIP_MEMORY_SIZE_MAX 0x08000000        // 27bits available
#else
#define ADDRESS_BITS 1
#define ON_CHIP_MEMORY_SIZE_MAX 0x20000000        // 29bits available
#endif
```

Basically the preprocessor macros ensures that there are enough bits to represent all on-chip memories. If there is more bits needed for the width than height or vise versa, the other address bits simply matches the one with more bits needed. For example, in the case of a 2x10 Checkerboard (2 row x 10 columns), to represent all 10 columns, 4 bits for the x ID are needed while only 1 bit is needed to represent 2 rows, the Checkerboard program simply set address bits to 4 for both x and y ID for the memory. Keeping the address bits same across the x and y IDs does decrease the maximum size of the on-chip memory as some of the bits might not be used, but it also avoid confusion and keeps the forwarding logic clean.

Once payload has arrives at Core Demux, the module will first check the most significant bit of the target address, if 1 then off-chip memory, if 0 then on-chip memory. In the case of on-chip memory payload, an address bit mask is created to get the y and x ID of the target address:

```
int addr_bits_mask = 0b11111 >> (5 - ADDRESS_BITS);
int addr_mask = 0x7FFFFFFF >> (2 * ADDRESS_BITS);
int addr_y = (adr>>(31 - ADDRESS_BITS)) & addr_bits_mask;
int addr_x = (adr>>(31 - 2*ADDRESS_BITS)) & addr_bits_mask;
```

Then Core Demux will forward the payload to either the top, left, right, or bottom neighboring memory of the Core depending on the x and y location of that Core and the x and y location of the target address of that payload. Right before the payload gets forwarded, the address is masked with the "addr_mask" shown in the code above to make sure that the bits used only for routing is filtered out. If the address is unreachable by that Core or the address is out of range, then Core Demux will stop the simulation and report an error message. Below is an example reading/writing to all nearby memories from a Core:

```
--LOCAL MEMORY ACCESS TESTS FOR CORE 00--
TIME           0 s: 0:
                                    D0=------------
                M00=------------ M01=------------ M02=------------ M03=------------
D1=------------ M10=------------ M11=------------ M12=------------ M13=------------ D2=------------
                M20=------------ M21=------------ M22=------------ M23=------------
                M30=------------ M31=------------ M32=------------ M33=------------
                                    D3=------------

00 Writing to 0x00000000 (here):  HELLO 00!
00 Read from  0x00000000 (here):  HELLO 00!
TIME           0 s: 2:
                                    D0=------------
                M00=HELLO 00!    M01=------------ M02=------------ M03=------------
D1=------------ M10=------------ M11=------------ M12=------------ M13=------------ D2=------------
                M20=------------ M21=------------ M22=------------ M23=------------
                M30=------------ M31=------------ M32=------------ M33=------------
                                    D3=------------

00 Writing to 0x80000000 (DRAM0): Above 00!
00 Read from  0x80000000 (DRAM0): Above 00!
TIME           0 s: 4:
                                    D0=Above 00!
                M00=HELLO 00!    M01=------------ M02=------------ M03=------------
D1=------------ M10=------------ M11=------------ M12=------------ M13=------------ D2=------------
                M20=------------ M21=------------ M22=------------ M23=------------
                M30=------------ M31=------------ M32=------------ M33=------------
                                    D3=------------

00 Writing to 0xA0000000 (DRAM1): Left 00!
00 Read from  0xA0000000 (DRAM1): Left 00!
TIME           0 s: 6:
                                    D0=Above 00!
                M00=HELLO 00!    M01=------------ M02=------------ M03=------------
D1=Left 00!     M10=------------ M11=------------ M12=------------ M13=------------ D2=------------
                M20=------------ M21=------------ M22=------------ M23=------------
                M30=------------ M31=------------ M32=------------ M33=------------
                                    D3=------------

00 Writing to 0x20000000 (below): Below 00!
00 Read from  0x20000000 (below): Below 00!
TIME           0 s: 8:
                                    D0=Above 00!
                M00=HELLO 00!    M01=------------ M02=------------ M03=------------
D1=Left 00!     M10=Below 00!    M11=------------ M12=------------ M13=------------ D2=------------
                M20=------------ M21=------------ M22=------------ M23=------------
                M30=------------ M31=------------ M32=------------ M33=------------
                                    D3=------------
```

The test for Core00 in a 4x4 Checkerboard setting above shows that the read and write requests from a Core can indeed be routed to the correct memory. In version 1.3 Checkerboard model, this test is performed in every Core to make sure that there are no payload forwarding bugs in other Checkerboard configurations. To get the correct address from the ID of a memory, a function called "`id_to_memAddress`" is provided to translate the y and x ID value to 32bit address. To get the address of off-chip memories, simply type x and y values 1 outside the border. In the case shown above, id -10 (off-chip memory above Core00) returns 0x80000000 and id 10 (on-chip memory below Core00) returns 0x20000000. This helper function will work for any width and height value within the 16x16 limit because it also utilizes the "Address Bits" variable mentioned above.

2.3 Features and Functionalities of Checkerboard Model

This section includes the basic features and functionalities of the Checkerboard model and the options that the user have at compile time. This section also include the usage and options that the user have if the user decides to use the automatic code generator for Checkerboard user file. Notice that without the user filling in codes in each Core, the Checkerboard itself does not have any functionality and will not have any output. There will be a demo function included (also a flag for the code generator) that allows quick testing for Checkerboard's setup and connectivity. Details are included later this section.

To Compile the Checkerboard program manually without the help of codegen, the user will first have to set the system environment variable (this can be done either inside or outside makefile): If edit `makefile` variable:

```
SYSTEMC_HOME = path_to_systemc-2.3.3_pt
```

If set system variable in Linux (must comment out makefile `SYSTEMC_HOME`):

```
setenv SYSTEMC_HOME path_to_systemc-2.3.3_pt
```

After setting the path to SystemC library. The `makefile` commands can function normally. Use "`make help`" to display help messages.

```
> make help
"make width=[] height=[] checkerboard" to compile
"make checkerboard" to compile default 4x4
use codegen -h for codegen help
```

Without any additional arguments,"`make checkerboard`" would compile a standard 4x4 Checkerboard with just the provided file.

```
> make checkerboard
g++ -g -Wall -c checkerboard_user.cpp -o checkerboard_user.o \
        -I/opt/pkg/systemc-2.3.3_pt/include -DGRID_HEIGHT=4 -DGRID_WIDTH=4
g++ -g -Wall -c checkerboard_arch.cpp -o checkerboard_arch.o \
        -I/opt/pkg/systemc-2.3.3_pt/include -DGRID_HEIGHT=4 -DGRID_WIDTH=4
g++ -g -Wall checkerboard_user.o checkerboard_arch.o -o checkerboard \
        -I/opt/pkg/systemc-2.3.3_pt/include -L/opt/pkg/systemc-2.3.3_pt/lib-linux64 \
        -Xlinker -R -Xlinker /opt/pkg/systemc-2.3.3_pt/lib-linux64 -lsystemc
```

Use "`make height=[1-16] width=[1-16] checkerboard`" will compile Checkerboard model with user defined width and height. This approach requires the user to provide their own checkerboard_user.cpp file with all the connectivities mentioned in previous section. If the provided height and width does not match with the user file, the model will still likely to compile but will not execute properly. An example is shown here:

```
> make height=3 width=3 checkerboard
g++ -g -Wall -c checkerboard_user.cpp -o checkerboard_user.o \
        -I/opt/pkg/systemc-2.3.3_pt/include -DGRID_HEIGHT=3 -DGRID_WIDTH=3
g++ -g -Wall -c checkerboard_arch.cpp -o checkerboard_arch.o \
        -I/opt/pkg/systemc-2.3.3_pt/include -DGRID_HEIGHT=3 -DGRID_WIDTH=3
g++ -g -Wall checkerboard_user.o checkerboard_arch.o -o checkerboard \
        -I/opt/pkg/systemc-2.3.3_pt/include -L/opt/pkg/systemc-2.3.3_pt/lib-linux64 \
        -Xlinker -R -Xlinker /opt/pkg/systemc-2.3.3_pt/lib-linux64 -lsystemc
```

Version 1.3 Checkerboard introduced user code generator "codegen.py". The code generator does automatic code generation for the aforementioned "checkerboard_user.cpp" file. User can type "codegen.py -h" to get help with the code generator, as shown below:

```
> codegen.py -h
usage: codegen.py [-h] [-d] [-o filename] [-c | -lc] [-r] height width

Code generator for checkerboard_user file

positional arguments:
  height                int value for the height of checkerboard in range of
                        1...16
  width                 int value for the width of checkerboard in range of
                        1...16

optional arguments:
  -h, --help            show this help message and exit
  -d, --demo            add demo codes to generated file
  -o filename, --output filename
                        output filename, by default
                        codegen_chekcerboard_user[h]x[w].cpp
  -c, --compile         compile the checkerboard
  -lc, --onlycompile    only compile codegen_checkerboard without printing
                        code
  -r, --run             run the executable after compile
```

3 The Mandelbrot Set and Mandelbrot Visualization Application

In this work, we choose the computation of the Mandelbrot Set as benchmark to evaluate the Checkerboard platform model.

3.1 Definition of the Mandelbrot Set

The Mandelbrot set was first defined in 1978 by Robert W. Brooks and Peter Matelski [4], then later on visualized with higher quality by Benoit Mandelbrot at IBM [21]. Mathematically, according to the original document, the Mandelbrot set defined as a set of complex numbers c for which the function $f_c(z_{n+1})) = z_n^2 + c$ does not diverge to infinity where iterated from $z = 0$ [4]. To visualize the Mandelbrot Set, a program needs to treat the real and imaginary parts of c as image coordinates on the complex plane, the number of iterations each point takes to break the boundary can then be used to color the pixel that represent a dot on the complex plane. For detailed definition and other mathematical experiments of the Mandelbrot Set, please refer to the original documentation [4].

3.2 Base Model of Mandelbrot Set Visualization

The base model of Mandelbrot Set Visualization program is one of the many Mandelbrot Visualization examples introduced in The Recoding Infrastructure for SystemC (RISC) [22] [23]. The Mandelbrot Set Visualization SystemC model was developed by Professor Rainer Doemer, Guantao Liu, Center for Embedded and Cyber-physical Systems (CECS) as a demonstration for the various functionalities of RISC. As mentioned in

section 1.2, this Mandelbrot Set Visualization program is chosen to be mapped onto Checkerboard architecture because of its outstanding scalability. This program allows the user to choose between 1 to 256 (must be the power of 2) active parallel units to calculate the Mandelbrot Set, as shown in Figure 9. In addition, this program also offers many customizable parameters using preprocessor conditional definitions, for example, the maximum number of iterations, the width and height of the generated images, the zoom factor, number of images, etc. (Figure 8).

This model, although with many customizable parameters and many more features, cannot be used directly as the model to map onto the Checkerboard model. In addition, because the goal of this project is to show the scalability of the Checkerboard architecture, many features are left out unused from this base model. For example, the option to choose between linear motion, faster motion and slower motion is completely irrelevant to this project and therefore was simply kept in the 1.0 version of this model (explained in Section 3.3), there also exist a cleaned-up version of the 1.0 model without these features for easier manipulation. Because omitted features are relatively irrelevant to this project, details to these features in the base model will not be included in this section.[1]

```
#ifndef WIDTH           // size of the image in pixels
#define WIDTH           640
#endif
#ifndef HEIGHT
#define HEIGHT          512
#endif
#ifndef MAX_ITER        // depth of calculation
//#define MAX_ITER      1024
//#define MAX_ITER      2048
//#define MAX_ITER      3072
#define MAX_ITER        4096
#endif
#ifndef TARGET_ZOOM     // zoom factor per step towards the target
#define TARGET_ZOOM     0.7
#endif
#ifndef NUM_IMAGES      // number of images produced
#define NUM_IMAGES      20
#endif
#ifndef MAX_IMAGE_FILES // number of image files kept
#define MAX_IMAGE_FILES 0
#endif

#define LINEAR_MOTION   // move in equidistant steps (constant speed)
//#define FASTER_MOTION // move in logarithmic steps (slow, then faster and faster)
//#define SLOWER_MOTION // move in logarithmic steps (fast, then slower and slower)
...
```

Figure 8: Examples of customizable parameters

[1]More information can be found in the RISC git repository at http://www.cecs.uci.edu/~doemer/risc [24].

```
#ifndef PAR             // number of parallel units instantiated
//#define PAR           1
//#define PAR           2
//#define PAR           4
#define PAR             8
//#define PAR           16
//#define PAR           32
//#define PAR           64
//#define PAR           128
//#define PAR           256
#endif
...
#define M(n) MANDELBROT(n, (HEIGHT/PAR*(n-1)), (HEIGHT/PAR*n))
    M(1)
#if PAR > 1
    M(2)
#endif
#if PAR > 2
    M(3) M(4)
#endif
...
```

Figure 9: Parallelism parameters

3.2.1 The Base Model of Mandelbrot Set Visualization Explained

This subsection includes a high-level schematic and detailed top-down explanation of the base model. Some module's explanation involve SystemC knowledge, the specifics of which can be found on SystemC official reference page at http://www.systemc.org [2].

A image of the high level schematic of this model is included here to show the structure (Figure 10). Starting from the top level, the module that contains all submodules is named TOP. In the schematic, TOP is the largest box that encloses every other module in this base model.

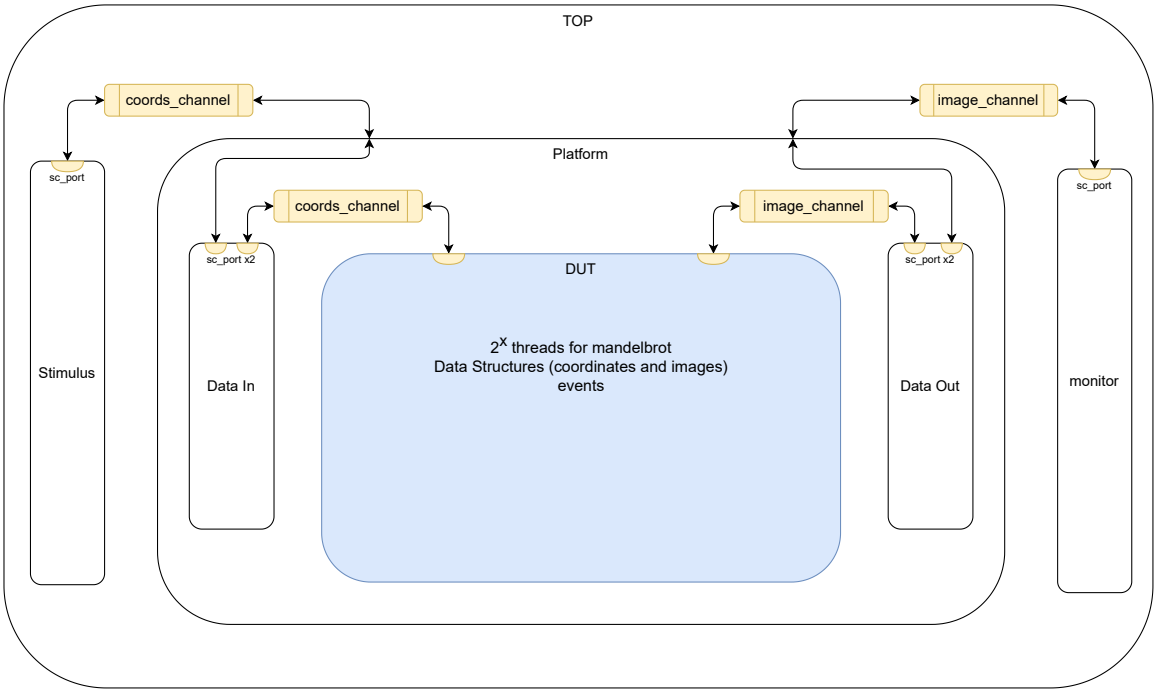

Figure 10: High Level Schematic of Base Model of Mandelbrot Set Visualization

The next group of modules includes stimulus, platform and monitor. Stimulus is in charge of creating the coordinates used for the Mandelbrot algorithm. The coordinates have a custom data structure with 4 floating point numbers which represent the top, left, right and bottom bound, as displayed in Figure 11. The coordinates are given as macro definitions and can be edited either by changing the code or adding definitions during compile time. The number of coordinates generated depends on the parameter "NUM_IMAGES", which stands for number of images generated. Since each image would require a coordinate, the number of coordinates generated from stimulus depends on the number of image parameter. If the user is asking for more than 1 image, then each coordinate generated after the first one will be zoomed in compared to the previous generated coordinate based on a zoom factor. The zoom factor is also given as a macro definition and can be edited just like the starting coordinates mentioned above. For example, the default zoom factor is set to 0.7 which means each image will be zoomed into a 70% area of its previous image (accomplished by zooming in the coordinates).

Once a coordinate is created in module Stimulus, it needs to be passed to module named Platform, which is the container for module Data In, Data Out, and DUT. The module in charge of communication between Stimulus and Platform, named "coords_channel", is a SystemC specific class called "Channels". Since the class is rather complex, only the basics will be explained in this paper. The minor details will not be included here in this paper,all the detailed information about SystemC Channels can be found on SystemC official reference page at http://www.systemc.org [2]. SystemC Channels, in short, are communication classes between 2 modules. Channels are used for Transaction-level Modeling (TLM) and generally used in TLM-1.0 SystemC modeling, the differences between TLM-1.0 and TLM-2.0 (which is used in later models including Checkerboard) are explained in detail in Section 3.2.3. The channels used to communicate between Stimulus and Platform are user-defined first-in-first-out (FIFO) channel that sends coordinates from Stimulus to Platform. The other channels (marked with yellow color in Figure 10) are also user-defined FIFO channels but

```
typedef struct Coordinates
{
    float_t l;  // left bound, e.g. -2.5
    float_t r;  // right bound, e.g. 1.0
    float_t b;  // bottom bound, e.g. -1.0
    float_t t;  // top bound, e.g. 1.0

    friend ostream& operator<<(ostream& os, const Coordinates& co);
} COORDS;
```

Figure 11: Custom Data Structure for Coordinates

used for communicating different data/data structures. In addition to Channels being used for inter-module communication, each module also have ports thats connected to channels in order to successfully transfer data. Port(s) inside each module are marked as yellow semi-oval in Figure 10.

When coordinate reaches Platform, they first get fed into the module "Data In" as shown in Figure 10. The module "Data In" is a infinite loop of sending coordinate right after receiving the coordinate from Stimulus. Similarly, the module "Data Out" is also a infinite loop of sending images through the image channel (marked yellow in Figure 10) right after receiving images from DUT. The purpose of "Data In" and "Data Out" is to keep a clear boundary between the clean SystemC hardware model and the "dirty" C++ software code in Stimulus and Monitor. Same as every other component in this model, both "Data In" and "Data Out" have sc_port to communicate with SystemC Channel in order to send and receive data from other modules.

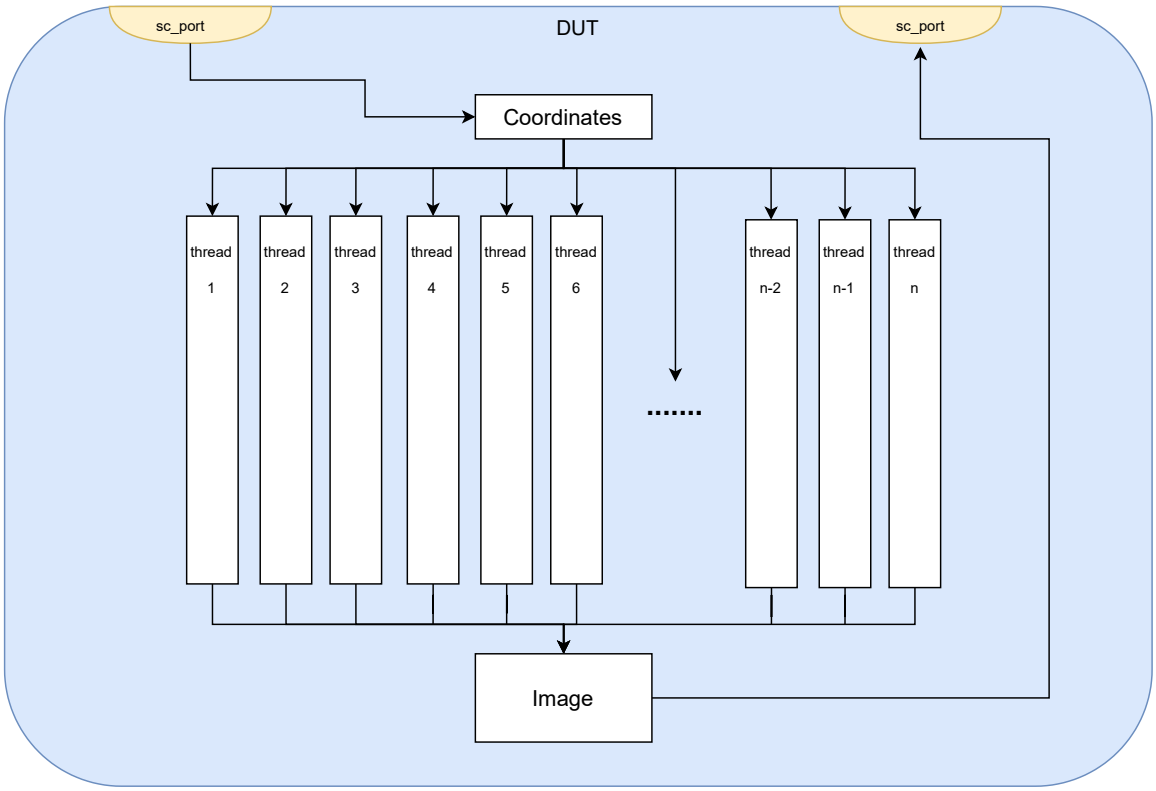

Figure 12: Schematic of DUT module of Base Model of Mandelbrot Set Visualization

Once "Data In" sends coordinates to Design Under Test (DUT) module, the Mandelbrot parallel units, as shown in Figure 12, would start. As mentioned above, the number of parallel units "n" used is a preprocessor macro defined at either the beginning of the C++ file or during compile time. The number of parallel units can only be power of 2, such as 1, 2, 4, 8, etc., the maximum number of parallel units is 256. Using the power of 2 numbers of parallel units (SystemC threads) is to avoid indivisible numbers since each thread is taking a equal number of rows in the image. For example, if the image width is 640 and height is 512, using 16 units means each unit will get (512/16) = 32 rows of pixels. If using 15 units, each thread would have (512/15) rows, which is not a whole number. Given using n number of units, the x-th unit will compute and draw row $\frac{\text{HEIGHT}}{n} * (x-1)$ to row $\frac{\text{HEIGHT}}{n} * (x)$. For example, if using 16 units, or n = 16, with image size $640 * 512$, the 1st unit will computer and draw row 0 to row 32, as illustrated in Figure 13.

All parallel unit will be working on 1 instance of a custom data strucutre named "Image" as shown in Figure 14. The data structure "Image" consists of three conflict-free 2-D array of characters to store a pixel's Red, Green and Blue value. Each color value ranges from 0 to 255, therefore arrays of characters are used to represent RGB values of pixels (in C++, characters are 1byte in size, which ranges from 0 to 255). The size of each 2-D array is defined by the user, which is HEIGHT * WIDTH, the height value by default is 512 and width by default is 640. Inside each thread, the function $f_c(z_{n+1})) = z_n^2 + c$ will be executed in a while loop to determine the number of iterations each pixel takes to breach the threshold. The while loop will also break if the number of iterations reaches the maximum number of iterations defined by the user (set to 4096 by default). After getting the iteration number for a pixel, that iteration number is modulated and mapped to a color to be displayed on a image. There are 16 available colors as shown in Figure 15, this is also shown

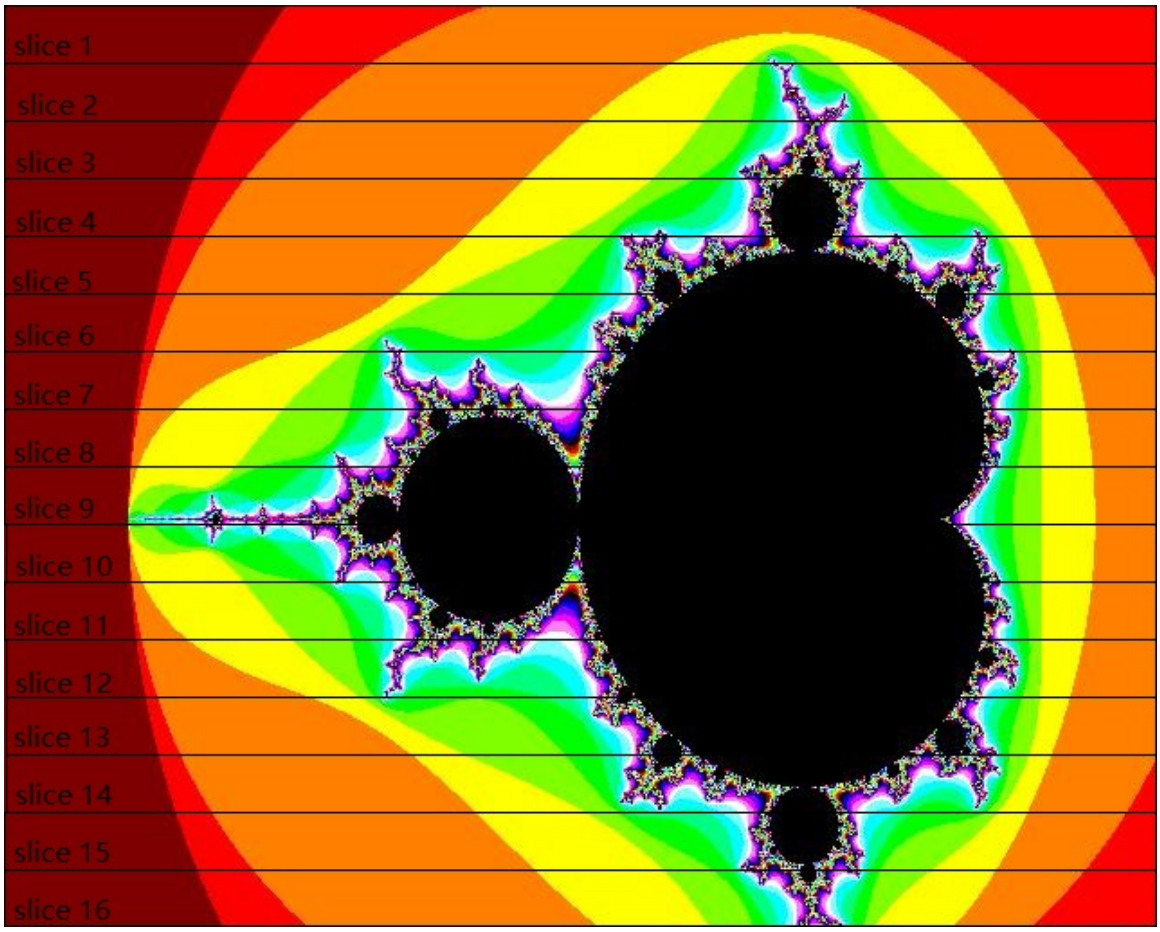

Figure 13: Example of 640 * 512 Mandelbrot with 16 Parallel Slices

```
typedef struct Image
{
    unsigned char R[HEIGHT][WIDTH];
    unsigned char G[HEIGHT][WIDTH];
    unsigned char B[HEIGHT][WIDTH];
            ......
} IMAGE;
```

Figure 14: Custom Data Structure for Images

in Figure 13. Once the color (R,G,B values) are assigned to that pixel, the program will update the pixel value on Simple DirectMedia Layer (SDL), which reflects on user's display. After calculating and displaying all assigned pixels of that image, thread will pause and wait for the next coordinate to start the next image. If the entire image is filled, DUT will sent the image to module "Data Out" via custom channels like the

channel between “Data In” and DUT. When an image is sent, DUT reads the next available coordinate from “coords_channel” and begins the next image. This cycle is inside an infinite while loop, in other words, DUT will always read coordinate, calculate and draw pixels, it does not end the program. The module “Monitor” is in charge of ending the program when it receives the expected number of images.

```
const unsigned char palette[NCOLORS][3] = {
    //  r    g    b
    {   0,   0,   0 },  // 0, black
    { 127,   0,   0 },  // 1, brown
    { 255,   0,   0 },  // 2, red
    { 255, 127,   0 },  // 3, orange
    { 255, 255,   0 },  // 4, yellow
    { 127, 255,   0 },  // 5, light green
    {   0, 255,   0 },  // 6, green
    {   0, 255, 127 },  // 7, blue green
    {   0, 255, 255 },  // 8, turquoise
    { 127, 255, 255 },  // 9, light blue
    { 255, 255, 255 },  // 10, white
    { 255, 127, 255 },  // 11, pink
    { 255,   0, 255 },  // 12, light pink
    { 127,   0, 255 },  // 13, purple
    {   0,   0, 255 },  // 14, blue
    {   0,   0, 127 }}; // 15, dark blue
```

Figure 15: 16 Colors Available to be Mapped

The module “Monitor”, like “Stimulus”, is not part of the Platform or Design Under Test (DUT), therefore it contains dirty code that is not SystemC and does not follow SystemC standard. As the name suggest, the module “Monitor” receives and observes the output of the DUT. In addition to receiving the image from DUT, module “Monitor” carries other functionalities. If the user decides to save the displayed Mandelbrot image, the user can edit the preprocessor directive “MAX_IMAGE_FILES” to more than 0 to keep the generated image. Images saved will be in PPM format, if the user add “-JPEG” flag during compilation of the program, “Monitor” will keep the image in both PPM format and JPG format (there is no difference between JPG and JPEG format). Another important functionality of “Monitor” is to terminate the program when the number of images are generated and sent to “Monitor”. Because the modules inside “Platform” are running forever (just like real hardware would), module “Monitor” is needed to terminate the SystemC simulation when there is no more work to be done by the system.

3.2.2 Functionalities of the Base Model of Mandelbrot Set Visualization

This subsection will include the basic functionalities of this base model and detailed explanation of how they work. Some of the explanations will involve SystemC knowledge, the specifics of which can be found on their official reference page at http://www.systemc.org [2]. As mentioned above, some features of this base model will not be introduced in detail in this session. These omitted features are listed here:

- Use -DBLACKWHITE to display black and white images instead of colored ones.
- Use -DVECTORIZE to use OpenMP SIMP vectorization with Intel compiler.

- Use -DFASTER_MOTION and -DSLOWER_MOTION to use logarithmic steps to speed up or slow down the motion.
- Use -DUSE_SC_THREAD option to use SystemC Method instead of SystemC Thread.

The listed features are simply kept untouched in later versions of the Mandelbrot set visualization program but is not mapped onto Checkerboard. There exist a variation of the version 1.0 Mandelbrot set visualization program without the code of the listed features, which allows for easier understanding and code manipulation.

When downloading and compiling files, both the `mandelbrot.cpp` file and the SDL files (`sdlsysc.c`, `sdlsysc.h`) are needed. The SDL files are a modified version of the original SDL library to fit for the SystemC libraries. Using the provided `Makefile`, user can compile the program with default settings. Before running the `make` command, user must set the correct system environment, the recommended system environment setting is as follows (SystemC 2.3.1 is also recommended, using other versions might cause problems):

```
setenv SYSTEMC_HOME path_to_/systemc-2.3.1_pt
```

An example complication process is as follows:

```
> make mandelbrotX_seq
g++ sdlsysc.c -c -O2 `sdl-config --cflags` -o sdlsysc.o
g++ mandelbrot.cpp \
        -I${SYSTEMC_HOME}/include \
        -L${SYSTEMC_HOME}/lib-linux64 \
        -Xlinker -R -Xlinker ${SYSTEMC_HOME}/lib-linux64 \
        -O2 -DPAR=32 -DDISPLAY -DWIDTH=640 -DHEIGHT=512 -DMAX_ITER=4096 -DNUM_IMAGES=5
        -DMAX_IMAGE_FILES=0 sdlsysc.o `sdl-config --libs` -lX11 \
        -lsystemc -o mandelbrotX_seq
```

Simply type `make mandelbrotX_seq` will compile the program with basic default arguments for the Mandelbrot Visualization program. If the user decides to manually compile with different arguments, the available arguments are:

1. Use -DPAR = (integer) to changing the number of working units: this argument determines the number of working units in threads (or SystemC methods) of the Mandelbrot Visualization program. More units working means the image will be divided into more blocks and each unit will get smaller blocks, given that the image size stay the same. In the example given above, setting 32 working units with -DPAR=32 means each unit will work on $512/32 = 16$ rows, each row have 640 pixels.
2. Use -DDISPLAY = (integer) to toggle display option: The -DDISPLAY will turn on the X11 display window for this program and allows the user to see the image getting formed, the user need to have X11 enabled to see the display window. This is among one of the omitted features in the mapped model so no detailed information will be provided here. This feature is kept in version 1.0 Mandelbrot Set Visualization model.
3. Use -DWIDTH = (integer) and -DHEIGHT = (integer) to change the size of image being calculated and produced: by default the image size is 640 pixels in width and 512 pixels in height. Changing these parameters will directly result in different simulated time and simulator runtime. If the width and height of a image is too large, it may result in extra long simulator runtime.
4. Use -DMAX_ITER = (integer) to change the number of iterations required to stop calculating a pixel: as explained in section 3.2.1, this model uses the Mandelbrot function $f_c(z_{n+1})) = z_n^2 + c$ in a while

loop and the MAX_ITER is the terminating condition. Getting a smaller max iteration number will result in a different image. If the MAX_ITER is set to a higher number than default, which is 4096, then the simulator run time and simulated time will be longer and vise versa.

5. Use -DNUM_IMAGES = (integer) to change the number of generated images: this argument determines the number of coordinates sent to module DUT and the number of images calculated and displayed on X11 window. By default there will be 5 images, changing this to a larger number will result in longer simulator run time and simulated time and vise versa.

6. Use -DMAX_IMAGE_FILES = (integer) to change the number of generated image kept (in folder): this argument determines the number of generated images kept on the machine. By default, this is set to 0, which means all images generated on the machine will be deleted. This argument is changed to -DSAVEIMAGE later in the mapped version of Mandelbrot Set Visualization, but the functionality remains the same. User can also use -DJPEG to save images to JPG format. Without -DJPEG argument, all the images are in .ppm format.

3.2.3 Limitations of the Base Model

The major reason why this version of the Mandelbrot Visualization program is not directly used is that this SystemC model is programmed with TLM-1. According to the official IEEE Std 1666-2011 Chapter 10.1 [25], TLM-1 standard "has three shortcomings with respect to the modeling of memory-mapped buses and other on-chip communication networks." These shortcomings include: TLM-1 has no standard transaction class and TLM-2.0 uses generic payload to address this issue. TLM-1 has no explicit support for timing annotation, no standardized way of communicating timing information between models. TLM-2.0 added timing annotation function arguments to the transport interface. TLM-1 interfaces require all transaction data to be passed by value or constant reference (which may slow down simulation) and TLM-2 passes transaction object by non-constant references, which is faster for modeling memory-mapped buses.

In addition to the lack of TLM-2.0 features, the base model is also not memory-accurate. As shown in Figure 12, the module DUT contains all the working units, the coordinate and the image. This is not an accurate representation of the actual hardware, data should be separated from the working units in a different module. In other words, coordinates and images cannot be stored in a "CPU", they should be stored in a memory. Because of this lack of memory and communication between memory and processor, the simulated time of the base model would not be accurate. On top of the inaccuracy of simulated time, the base model also cannot properly simulate memory space: this model have essentially infinite memory for coordinates and images, which is not accurate at all. Since memory management is essential for any hardware designs, having memories represented in this model and taking memory-processor communication are necessary. Also, because the "checkerboard" model will be memory accurate and implemented with SystemC TLM-2.0, a TLM-2.0 base model is needed for comparing their timing data.

3.3 Version 1.0 of Mandelbrot Set Visualization

The version 1.0 Mandelbrot Set Visualization model overcomes the limitations within the base model and kept all the features (detailed in section 3.2) that are not relevant to this project. The 1.0 model is implemented with SystemC TLM-2.0 features and therefore contains standard transaction class have support for detailed timing annotation without sacrificing the speed of the simulator. Because the 1.0 model does contain memories and have the correct behavior of the actual hardware, this model will be used to mapped onto Checkerboard model and also work as a reference for ideal model with perfect scalability. Detailed information on the differences between the base model and 1.0 model is included in this section.

3.3.1 1.0 Model of Mandelbrot Set Visualization Explained

This subsection will include a high-level schematic of the 1.0 model of Mandelbrot Set Visualization and detailed top-down explanation the different module compared to the base model. Some sections will involve SystemC knowledge, the specifics of which can be found on SystemC official reference page at http://www.systemc.org [2].

A image of the high level schematic of this model is included here to show the structure and conntectivity of the model (Figure 10). Starting from the top level, similar to the base model, the module that contains all submodules is named TOP, the largest box that encloses every other module in this model.

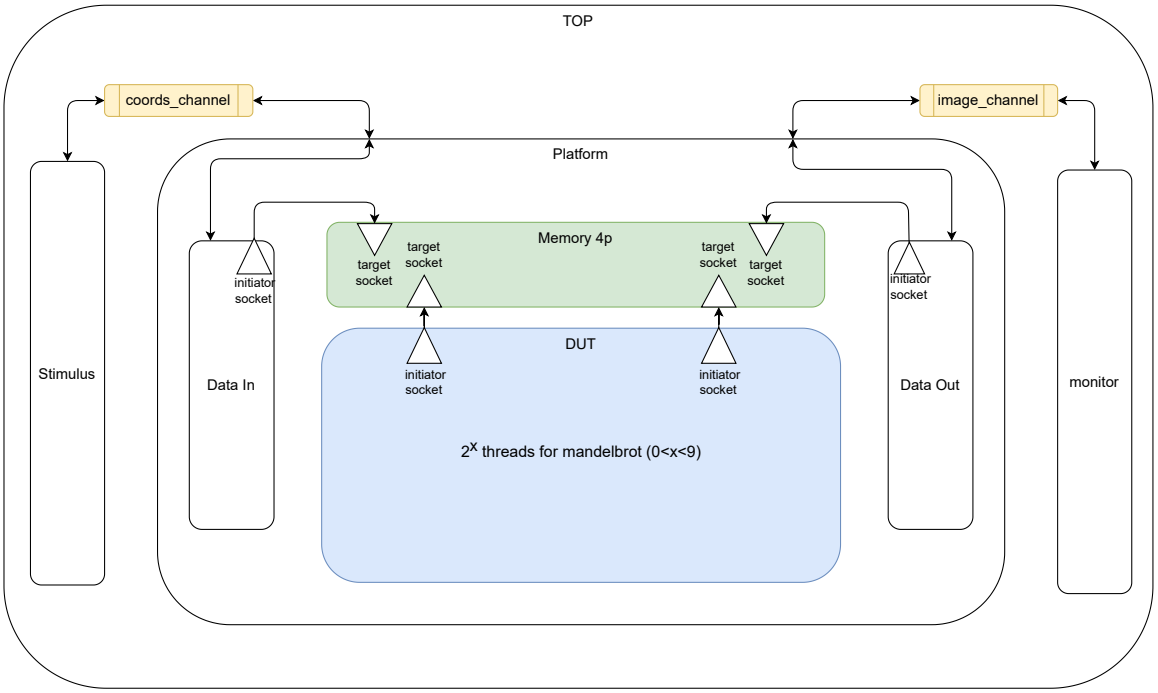

Figure 16: High Level Schematic of 1.0 Mandelbrot Set Visualization

All updates from the base model to 1.0 model are within the Platform module. Module Stimulus, Monitor, coords_channel and image_channel all remain the same, because they are not part of design under test and can have non-TLM-2.0 modules or "dirty" code.

Starting from module Data In, the difference is shown in Figure 16 as the triangle pointing outwards of the Data In module. Replacing the `sc_port` in that position, a `simple_initiator_socket` is used for communication between TLM-2.0 modules (simple initiator socket is under `tlm_utils` name space, detailed can be found on SystemC reference page). To put it in a simpler way, initiator sockets start the communication process and sends a payload toward target sockets, when target sockets receive a request, it loads the data pointer based on the requested address in the payload sent by the initiator socket. In the 1.0 model, all TLM-2.0 modules are connected via initiator sockets and target sockets, which will have generic payload and have explicit support for timing annotation. Each initiator socket is and must connect to target socket. In the case of the initiator socket in Data In, the initiator socket is connected to a target socket in module `Memory 4p` (4p stands for 4 ports). The same type of initiator socket and target socket pair is also used for communication between module Data Out and `Memory 4p`.

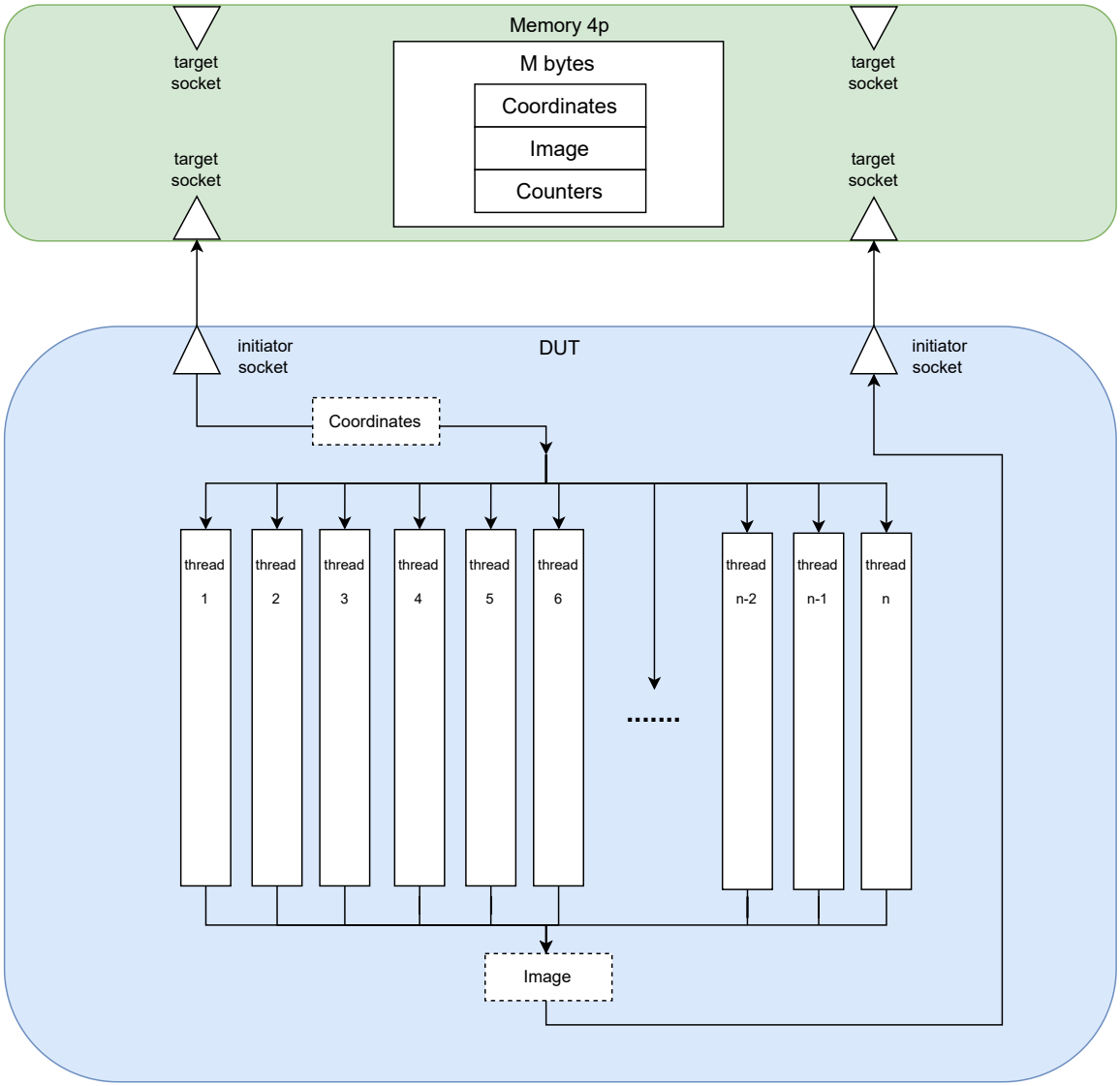

Figure 17: DUT schematic of 1.0 Mandelbrot Set Visualization

Another difference between version 1.0 model and the base model of Mandelbrot Set Visualization is the addition of memory unit. Because the inclusion of memory, 1.0 model is more memory accurate compared to the base model: all the data structures and variables are stored inside DUT in the base model whereas in 1.0 model, key data structures are stored in memory. Since the 1.0 model is not a total accurate representation of the hardware and is a high-level model, not all data will be stored in memory, as shown in Figure 17, there are local copies of coordinates and images. Although the model is not 100% memory accurate, module DUT still need to read coordinates from memory and write `Image` into memory with proper timing. The added counters in the memory and DUT was needed to prevent the lost of events and to synchronize between producer modules and consumer modules (losing events is a SystemC issue and will not be introduced in

detail).

Besides the addition of memory module and change of means of communication between different modules, there is no significant change in the calculation for each pixel from the Mandelbrot function in module DUT. There is also no change to module Stimulus and Monitor, therefore there is no change to how the coordinates are produced or how each pixel is updated and displayed in SDL window.

3.3.2 Functionalities of the 1.0 Model of Mandelbrot Set Visualization

The functionalities of the 1.0 Model of Mandelbrot Set Visualization are the same as base model with only few additions. The overlapping features will be listed below and will not be explained in detail. The output images (given the same parameters) of the 1.0 model is exactly the same as the base model as well. There are new functions added to calculate simulated time and will be explained in this session in detail. To use the model, system environment settings and compilation process is similar to the base model. This is an example to setup system environment and compile with the provided `Makefile`:

```
setenv SYSTEMC_HOME path_to_/systemc-2.3.1_pt

>make mandelbrotX_TLM2
g++ sdlsysc.c -c -O2 `sdl-config --cflags` -o sdlsysc.o
g++ mandelbrot_TLM2.cpp \
        -I${SYSTEMC_HOME}/include \
        -L${SYSTEMC_HOME}/lib-linux64 \
        -Xlinker -R -Xlinker ${SYSTEMC_HOME}/lib-linux64 \
        -O2 -DPAR=32 -DDISPLAY -DWIDTH=640 -DHEIGHT=512 -DMAX_ITER=4096 -DNUM_IMAGES=5
        -DMAX_IMAGE_FILES=0 sdlsysc.o `sdl-config --libs` -lX11 \
        -lsystemc -o mandelbrotX_TLM2
```

If the user decides to compile manually, these are the available arguments, the overlapped arguments are listed here details can be found in section 3.2.2.

- Use -DBLACKWHITE to display black and white images instead of colored ones.
- Use -DVECTORIZE to use OpenMP SIMP vectorization with Intel compiler.
- Use -DFASTER_MOTION and -DSLOWER_MOTION to use logarithmic steps to speed up or slow down the motion.
- Use -DUSE_SC_THREAD option to use SystemC Method instead of SystemC Thread.
- Use -DPAR = (integer) to changing the number of working units.
- Use -DDISPLAY to toggle display option.
- Use -DWIDTH = (integer) and -DHEIGHT = (integer) to change the size of image being calculated and produced.
- Use -DMAX_ITER = (integer) to change the number of iterations required to stop calculating a pixel.
- Use -DNUM_IMAGES = (integer) to change the number of generated images.
- Use -DMAX_IMAGE_FILES = (integer) to change the number of generated image kept in folder.

The added functionalities are listed here:

- Use -DCALC_TIMING to display simulated time for calculation: add this argument during compilation makes the program display the total unit time all calculation units spend on calculating the pixels for the Mandelbrot Set. Each iteration spent on the Mandelbrot equation will add 1 to the total number of calculation time. This argument should not be used with -DMEM_TIMING because both timings are calculated in unit time and cannot be distinguished between one another.
- Use -DMEM_TIMING to display simulated time for communication: add this argument during compilation makes the program display the total unit time the memory unit spend on responding to all reading and writing requests from other components in the model. Each word read and write will add 1 to the total number of communication time. This argument should not be used with -DCALC_TIMING because both timings are calculated in unit time and cannot be distinguished between one another.

The timing arguments will be used to get the simulated calculation and communication time of the 1.0 Mandelbrot Set Visualization model. The obtained timing data will be used to compare with timing data from the mapped GPC version of the Mandelbrot Set Visualization in later sections in this paper.

3.3.3 Scalability and Timing of the 1.0 Model of Mandelbrot Set Visualization

As mentioned in section 3.3.1 and section 3.3.2, the 1.0 model is more memory accurate relative to the base model. In terms of the memory accuracy of the model, it is still far off from the actual hardware. The 1.0 model's DUT in Figure 17 is designed to have perfect scalability, because each calculating unit has uncontested access to coordinate and image. There is no contention between the single coordinate instance (or image instance) and the presumably hundreds of working units. This is equivalent to a hardware having an infinite speed data bus that allows instant transfer from the memory to each working unit. Therefore the simulated communication time only includes the delay caused by the memory itself and nothing between the memory and working units. Even though the simulated communication time is only approximate, this is still valuable data. Because the mapped checkerboard model will have a more realistic communication timing, timing data from the 1.0 model (which is perfectly scalable) can be used to compare against the more realist model to prove that the checkerboard model can scale up or down without being that much slower than the ideal model.

4 Mapping of Mandelbrot Set Visualization onto Checkerboard Model

This chapter includes a detailed explanation of an example mapping of a Mandelbrot Set Visualization application on a 4x4 Checkerboard model (the same example model used in the previous chapter). This mapping had many variations and has gone through multiple versions and stages. In the features and schematic sections, only the latest version of the Mandelbrot on Checkerboard model will be used.

4.1 Example Mandelbrot on Checkerboard 4x4 Explained

This section includes a high-level schematic of the Mandelbrot on Checkerboard model with an example 4x4 layout Checkerboard base model (the same layout used in previous chapter) as well as a schematic detailing how this mapping works on a logic level. Given that the simulated hardware of the Checkerboard model does not change, there is no need to explain everything about the Checkerboard again. Additionally, the Mandelbrot Set Visualization program also remains very similar to the Model 1.0 Mandelbrot Set Visualization program in Chapter 3, only the few differences between the mapped model and the 1.0 Mandelbrot model will be included here.

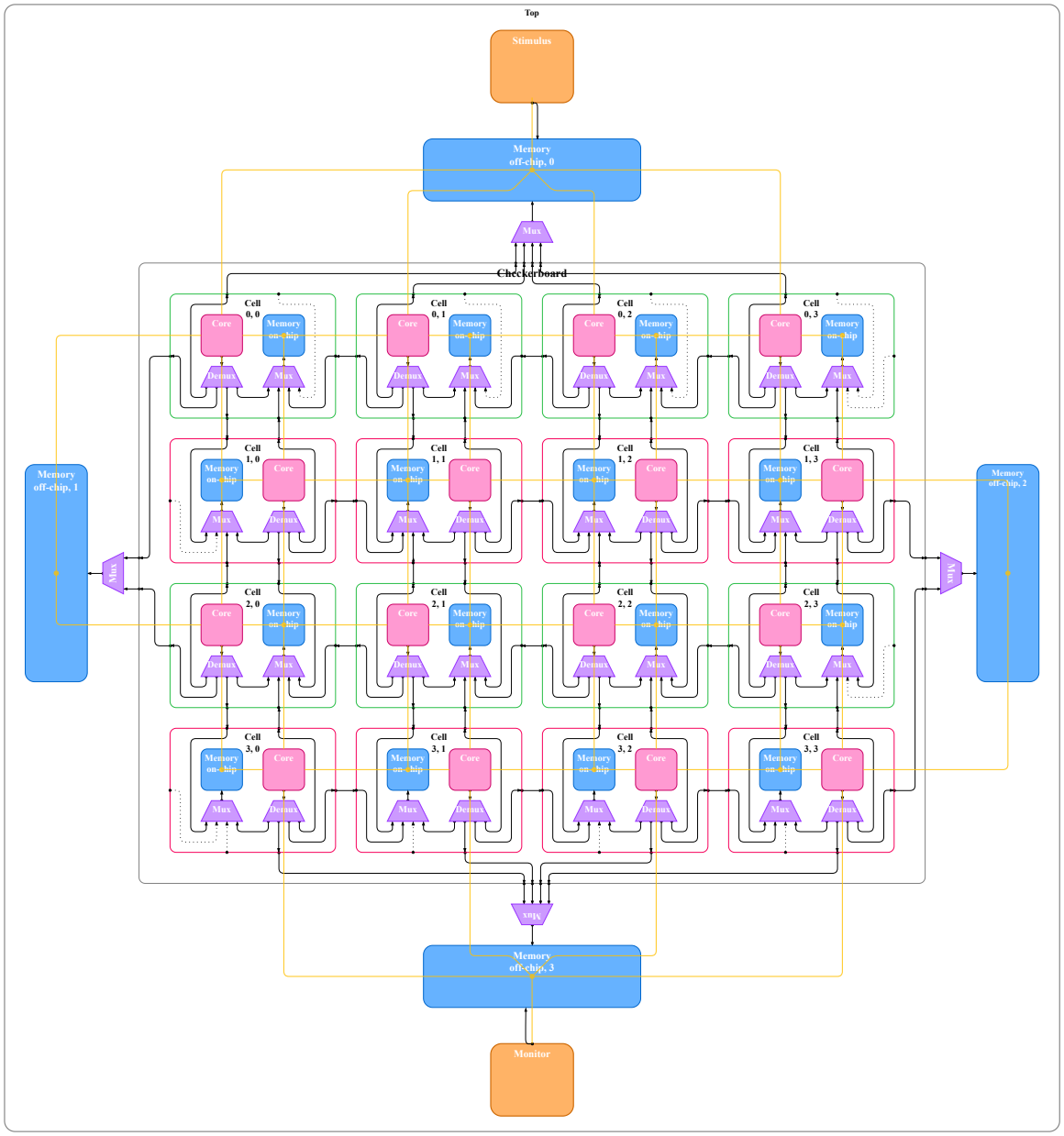

Figure 18: High-level Schematic of Mandelbrot on Checkerboard 4x4 Model, figure modified from [20]

As shown in Figure 18, the overall structure of the mapped model is very similar to the Checkerboard structure in Figure 1 in the previous chapter. The only difference between the two models is how the Stimulus and Monitor are connected to the off-chip memories on top and bottom outside of the center Checkerboard. The top and bottom off-chip memories, different from the default off-chip memories which have one socket and are only connected to the off-chip memory Mux, have two target sockets. One of the two sockets is used for the off-chip memory Mux and the Checkerboard, the other socket is connected directly to Stimulus and

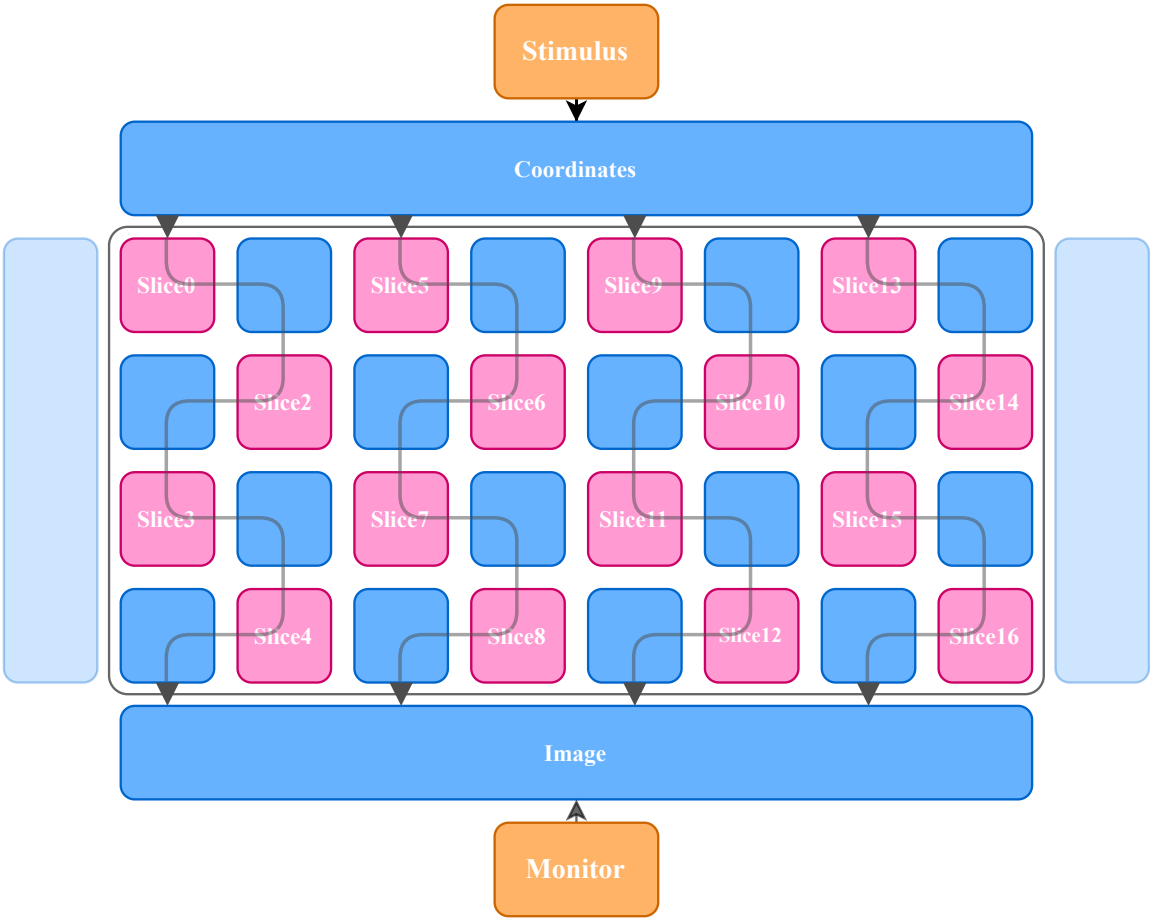

Figure 19: Logic Schematic of Mandelbrot on Checkerboard 4x4 Model

Monitor. This direct connection from Stimulus/Monitor to off-chip memory sockets avoids the additional contention introduced to the off-chip memory Mux compared to the other method.

Since the hardware schematic does not help much with the understanding of the actual mapping, another schematic is needed. The logic schematic, displayed in Figure 19, shows the separation and distribution of a Mandelbrot image. Similar to the base Mandelbrot Set Visualization model and the later versions, every image is divided into different slices (shown in Figure 13). Each Core in the mapped Checkerboard model is in charge of one slice of the image as illustrated in Figure 20. However, dividing the image among many cores and combining all slices into one image at the bottom off-chip memory is no trivial task, many more steps are needed to map the Mandelbrot Set Visualization Application. To put it in a simple way, each black line in Figure 19 represents a flow of 4 slices of the complete image. Because the Checkerboard model in this example has a 4x4 layout, every column would produce 4 slices. In a 3x3 Checkerboard layout, the Mandelbrot on Checkerboard Model will then have 9 slices, which also makes each column of Cells produce 3 slices.

In Figure 21, the simplified dataflow of a single column of Mandelbrot on Checkerboard 4x4 is shown. The red block represents the Core and the blue block represents the memory. There are 4 columns of Cells

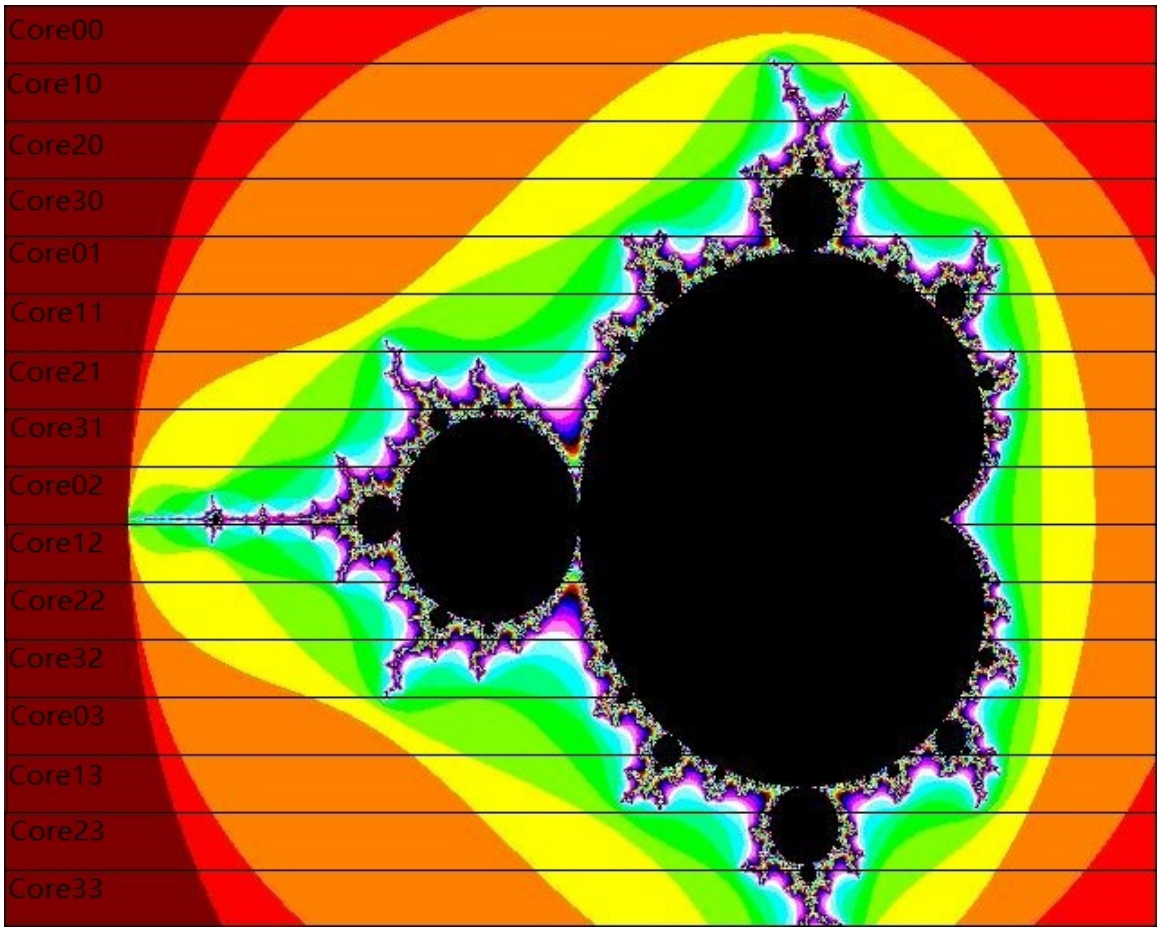

Figure 20: Mandelbrot Slices assigned to GPC Cores

in this example 4x4 Checkerboard and therefore 4 black lines shown in Figure 19 (the logic schematic of Mandelbrot on Checkerboard 4x4). Starting from the very first row (Cells with a id y=0), each Core needs to fetch the current Coordinates (the same Coordinates in Mandelbrot Base model and later variants) from the memory above that Core. The function PopCoord() distinguishes if the Core is the first row, in the case of first row Cores, PopCoords() takes the generated Coordinate from the top off-chip memory and writes a counter for the Stimulus to keep track. If the Core is not in the first row, then PopCoords() will simply take the Coordinates from the on-chip memory of its neighbor above and write a counter to keep track of the number of Coordinates.

The next function, PushCoords(), as the name suggest, pushes the Coordinates to the local memory of that Core. This function is rather simple and straightforward, with the only exception being Cores in the last row. Because there is no more Core below the last row, there is no need for the Cores in last row to forward more Coordinates for other Cores, and thus there is no PushCoords() in the last pink block in Figure 21

After a Core acquires a Coordinate for the Mandelbrot calculation function with PopCoord() and passes that Coordinate to the next Core with PushCoord() (with the exception of last row), that Core can start its

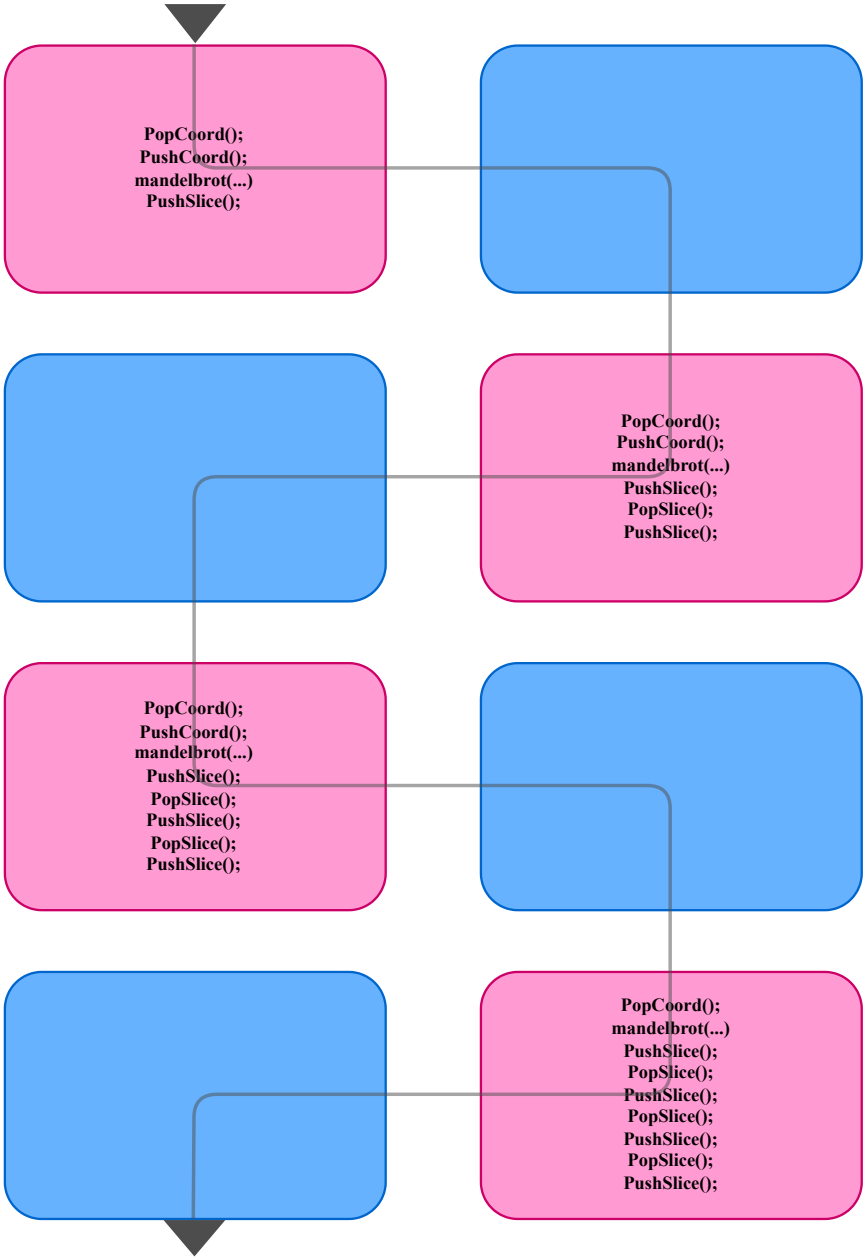

Figure 21: Example Dataflow per Column of Cells of Mandelbrot on Checkerboard 4x4

Mandelbrot calculation function. The Mandelbrot function in each Core can be simplified to the following:

```
void Core::mandelbrot(int rowStart, int rowEnd, SLICE& slice){
        for(int row = rowStart; row < rowEnd; row++){
                for(int col = 0; col < WIDTH; col++)
                {
                        [mandelbrot calculation]
                }
#ifdef DISPLAY
                [update slice in X11 window]
#endif
}
```

The Mandelbrot function takes the starting row location, the end row location and the address to the slice data structure. The parameter "rowStart" and "rowEnd" is determined based on the x id and y id of the Core.

Once each pixel in a slice is calculated, the Core needs to write that slice to a memory. Depending on the position of the Core, a non-last row Core will push the slice into its local on-chip memory while the last row Core will push the slice directly onto the bottom off-chip memory. The PushSlice() function checks if the Core is in the last row and make adjustments to the place where the slices are pushed into. Besides the last-row special case, the PushSlice() function also checks if the local memory is on the left or the right of the Core and notifying the right memory signal. Similar to the PushCoord() function, the PushSlice() function also notifies the event of the memory that it wrote into, letting other Cores know that a Slice has been pushed.

Slices in the on-chip memories cannot magically get to the bottom off-chip memory and therefore the Cores will also have to forward slices from their neighbors once they are done calculating their own Mandelbrot slice. The PopSlice() function, when used with the PushSlice() function, allows each Core to transfer a Slice of the Mandelbrot image from the above neighboring memory to its local on-chip memory. As shown in Figure 21, the first row Cores do not need PopSlice(), this is simply because they do not have a neighbor Core on top of them to produce a Slice. For other Cores that are not in the first row, the number of Pushing and Popping Slice depends on the id y of that Core. For example, the Core in 3rd row will need to Push its own Slice, Pop Slice from second row, Push it, then Pop Slice from first row, then Push it again, so total of 3 Pushes and 3 Pops. Similar to the PushSlice() function, PopSlice() also notifies the event of the memory that it reads from, letting the writer Core knew that the Slice was popped from the queue.

When a Slice gets pushed to the bottom off-chip memory, module Monitor, which listens to the event of the bottom off-chip memory, gets notified that a Slice has been written to the off-chip memory. Monitor waits till every slice of that image is filled and then displays a message that an image has been formed. Once the images received matches the number of images specified by the user, Monitor will stop the simulation. The Monitor module also has other functionalities such as display the received image on a X-11 window with the -DMONITOR flag during compilation and writing the image to a ppm/jpeg file.

4.2 Functionalities of Scalable GPC Mandelbrot Model

This section includes the basic features and functionalities of the Mandelbrot on Checkerboard model. This section also includes the usage and options that the user has if the user decides to use the automatic code generator for Mandelbrot on Checkerboard user files. Notice that the Mandelbrot Set Visualization program and its image calculation methods remains the same across the base model of Mandelbrot Set Visualization, its later variants and this mapped Mandelbrot on Checkerboard model.

To compile the Mandelbrot on Checkerboard model manually without the help of code generator, the user will first have to set the system environment variable either inside the Makefile or in linux terminal. If edit the Makefile variable:

```
SYSTEMC_HOME = path_to_systemc-2.3.3_pt
```

If set system variable in Linux (must comment out makefile SYSTEMC_HOME):

```
setenv SYSTEMC_HOME path_to_systemc-2.3.3_pt
```

After setting the path to systemC library. The makefile can compile properly. Use **"make help"** to display help messages.

```
> make help
use "make mandelbrotX_GPC" to compile base version 4x4 layout
use "make width=[] height=[] mandelbrotX_GPC" to compile with custom user file
use "codegen -h" to get help info for codegen
```

These makefile commands have exactly the same functionalities as introduced in Section 2.3. Please refer to previous chapter for detailed information on basic compilation options. Besides the basic compilation rules, Mandelbrot on Checkerboard also offers other compilation options. Some options are different from the Mandelbrot 1.0 model, here are some of the similar options:

- Use -DWIDTH = (integer) and -DHEIGHT = (integer) to change the size of image being calculated and produced.
- Use -DMAX_ITER = (integer) to change the number of iterations required to stop calculating a pixel.
- Use -DNUM_IMAGES = (integer) to change the number of generated images.

The additional/different functionalities are listed here:

- Use -DSAVE_IMAGE to write and save image in the working directory. A reference image can be found in Appendix.
- Use -DJEPG to also save the image as .jpg format. A reference image can be found in Appendix.
- Use -DDISPLAY to enable display window for views inside the working Checkerboard Cores as the pixels are calculated.
- Use -DMONITOR to enable display window for views inside the bottom off-chip memory as the Monitor detects incoming Slices of the image.
- Use -DCALC_TIMING to display simulated time for calculation: add this argument during compilation makes the program display the total unit time all calculation units spend on calculating the pixels for the Mandelbrot Set. Each iteration spent on the Mandelbrot equation will add 1 to the total number of calculation time. This argument should not be used with -DMEM_TIMING because both timings are calculated in unit time and cannot be distinguished between one another.
- Use -DMEM_TIMING to display simulated time for communication: add this argument during compilation makes the program display the total unit time the memory unit spend on responding to all reading and writing requests from other components in the model. Each word read and write will add 1 to the total number of communication time. This argument should not be used with -DCALC_TIMING because both timings are calculated in unit time and cannot be distinguished between one another.

Version 1.5 Mandelbrot on Checkerboard introduced a user code generator "codegen_mandelbrot.py". This code generator is built on top of the existing code generator for the Checkerboard model with the additional functions and scalable Core codes needed for the Mandelbrot on Checkerboard model. User can type **"codegen_mandelbrot.py -h"** to get help with the code generator, as shown below:

```
> codegen_mandelbrot.py -h
usage: codegen_mandelbrot.py [-h] [-o filename] [-c | -lc] [-r] height width

Code generator for checkerboard_user_mandelbrot file

positional arguments:
  height                int value for the height of checkerboard in range of
                        1...16
  width                 int value for the width of checkerboard in range of
                        1...16

optional arguments:
  -h, --help            show this help message and exit
  -o filename, --output filename
                        output filename, by default
                        codegen_chekcerboard_user[h]x[w].cpp
  -c, --compile         compile the checkerboard
  -lc, --onlycompile    only compile codegen_checkerboard_mandelbrot without
                        printing code
  -r, --run             run the executable after compile
```

5 Experimental Results

This chapter introduces our experimental setup and testing methodology for Mandelbrot Set Visualization 1.0 Model and Mandelbrot on Checkerboard. All recorded results are in unit time since the actual data for components, such as on-chip memory, off-chip memory, multiplexers and demultiplexers requires in-depth research and cannot be done with the time constraints.

5.1 Experimental Setups

All experiments for Mandelbrot Set Visualization share the same parameters with the only difference being the amount of parallel units. The parallel unit parameter for Mandelbrot 1.0 Model is set by changing the -DPAR= (integer) flag and the amount of parallel working Cores for Mandelbrot on Checkerboard Model is set by changing the GRID_WIDTH and GRID_HEIGHT parameter. Then user code can be generated with the code generator. All experiments run on the same machine 'delta' with fixed frequency at 3.4Ghz. All experiments also have display turned off and no images saved at the end. This is done to avoid delays introduced from the network. The calculation unit time is obtained by adding only the -CALC_TIMING flag for both models. The communication unit time is obtained by having both the -CALC_TIMING and -MEM_TIMING flag, then subtract the calculation unit time. The detailed information of the timing flags are included in Chapter 2 and Chapter 4.

Experiments for Mandelbrot Set Visualization 1.0 Model and Mandelbrot on Checkerboard both use the following parameters:

```
-DWIDTH=640 -DHEIGHT=576 -DMAX_ITER=4096
-DNUM_IMAGES=5 -DMAX_IMAGE_FILES=0
```

5.2 Data Collected

Table 1 show experimental results for Mandelbrot Set Visualization 1.0 Model. Due to spacial constants, some acronyms are used in the table: "PU" stands for number of Parallel Units used in Mandelbrot set calculation. "Calc UT" stands for Computation Unit Time, which is the simulated time for only computation. "Comm UT" stands for Communication Unit Time, which is the simulated time for only communication. "SRT" stands for Simulator Run Time (in seconds), which is measured in real time. "IF" stands for Improvement Factor, which is calculated from dividing the current Calculation Unit Time by the previous Calculation Unit Time.

PU	Calc UT	Comm UT	SRT (sec)	IF
1	4075078882	1658988	18.45	
2	2543059511	1658988	18.85	1.602
4	1471609768	1658988	19.06	1.728
8	764406229	1658988	19.49	1.925
16	389276159	1658988	19.83	1.963
32	196793623	1658988	19.89	1.978
64	98881005	1658988	20.24	1.990

Table 1: Experimental Results for Mandelbrot Set Visualization 1.0 Model

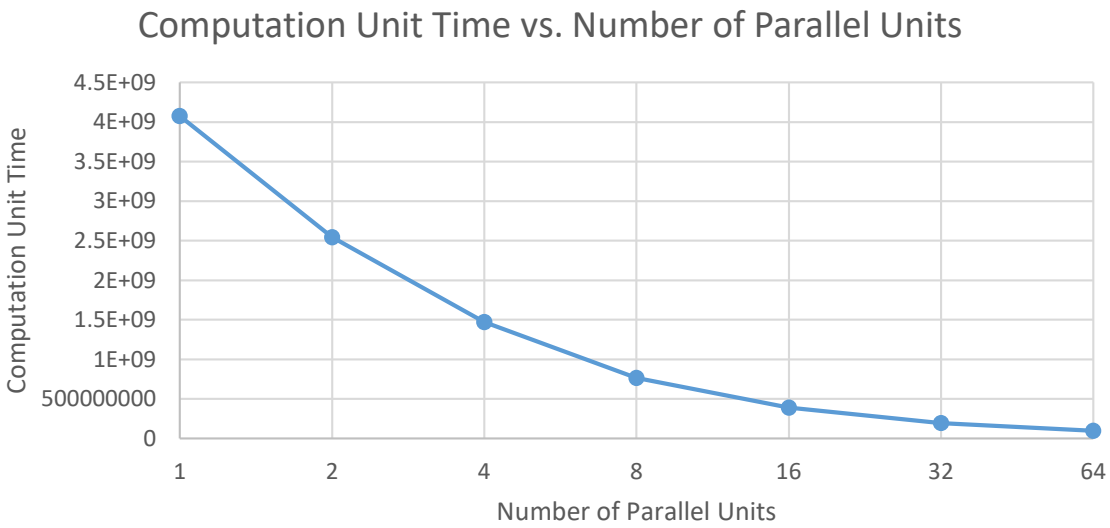

Figure 22: Mandelbrot Set Visualization 1.0 Model Computation Unit Time vs. Number of Parallel Units

The computation unit time vs. number parallel units graph shown in Figure 22 reflects the fact that for the Mandelbrot 1.0 Model, with almost perfect scalability, doubling the amount of calculating units cuts the computation time in half. It is normal when parallel units go from 1 to 2 to 4 the computation time did not go down by a half, the reason being that the Mandelbrot Image is more computationally expensive to calculate in the middle rows than to calculate on the edge rows. As the number of parallel units increase, each core has a smaller number of rows to calculate, thus the difference between heavy load rows in the middle and light

load rows on the edge is smaller. This is why as the parallel unit number goes up, the improvement factor stays close to 2x.

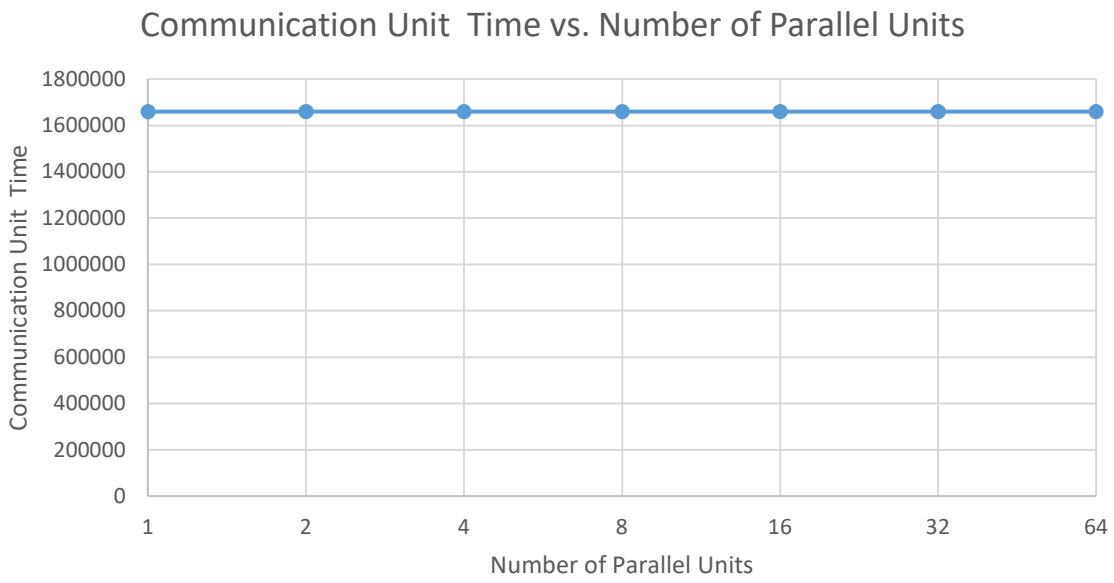

Figure 23: Mandelbrot Set Visualization 1.0 Model Communication Unit Time vs. Number of Parallel Units

Because the Mandelbrot Set Visualization 1.0 Model writes the whole image after all the calculation is done, the communication unit time vs. number of parallel units is straightforward. The communication unit time is just the number of words written to and read from the memory. Even though both timings are recorded in unit time, the communication unit time spent is not on the same magnitude as the computation unit time since Mandelbrot Set Visualization is very computation heavy.

Table below show experimental results for Mandelbrot on Checkerboard Model. Due to spacial constants, some acronyms are used in the table: “PU” stands for number of Parallel Units used in Mandelbrot set calculation. “Layout” is the Height and Width of the Checkerboard model. “Calc UT” stands for Computation Unit Time, which is the simulated time for only computation. “Comm UT” stands for Communication Unit Time, which is the simulated time for only communication. “SRT” stands for Simulator Run Time (in seconds), which is measured in real time. “IF” stands for Improvement Factor, which is calculated from dividing the current Calculation Unit Time by the previous Calculation Unit Time.

PU	Layout	Calc UT	Comm UT	SRT	IF
1	1x1	4075078982	1659029	18.61	
2	1x2	2543059611	829598	18.72	1.602
3	1x3	1799580598	553130	19.22	
4	2x2	1471609868	1106205	19.30	1.728
6	2x3	1001359981	553151	19.40	
8	2x4	748676317	553270	19.69	1.965
9	3x3	677227735	430285	19.87	
12	3x4	513788640	530309	19.96	
16	4x4	388649106	484295	20.26	1.926
20	5x4	304283538	430599	20.00	
24	6x4	260990624	461403	20.19	
28	7x4	218037155	423066	19.75	
32	8x4	196733139	415443	20.07	1.975
36	9x4	174894521	398790	20.19	
40	10x4	153292084	417555	19.88	
44	11x4	142358440	400279	20.23	
48	12x4	131401723	404199	20.52	
52	13x4	120450230	391782	20.17	
56	14x4	109542155	394545	20.34	
60	15x4	98877301	416078	19.75	
64	16x4	98877301	381393	20.36	1.989

Table 2: Experimental Results for Mandelbrot on Checkerboard Model

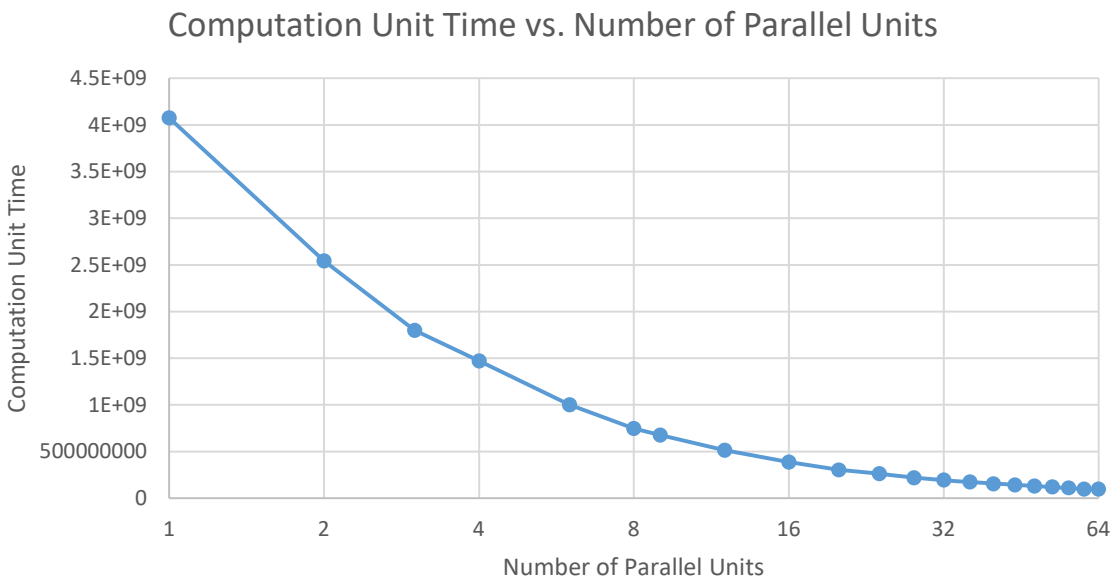

Figure 24: Mandelbrot on Checkerboard Model Computation Unit Time vs. Number of Parallel Units

The computation unit time vs. number of parallel units graph in Figure 24 shows a very similar curve to the Mandelbrot 1.0 model's result. The improvement factor in Table 2 also displays very similar calculation unit time improvements when the number of parallel units double. Similar to the Mandelbrot 1.0 Model, when the number of parallel units is low, the Cores with heavier load (middle rows in the image) will take longer to execute, which leaves the other Cores with lighter weight jobs waiting after they finish. Therefore the improvement factor is lower than 2x when parallel units double when PU is a small number. When PU increases, this difference in work load reduce, which then result in a improvement factor a lot closer to 2x when doubling the number of parallel units.

One outstanding point is that the Calculation Unit Time does not change when going from 60 parallel units (15x4 layout) to 64 parallel units (16x4) while other data suggest that increasing the number of calculation units decreases the calculation unit time. However, this is to be expected because each Core is in charge of calculating only a few rows of the pixels of the Mandelbrot Set image. Because the image height is set to 576, when there are 60 units, 576/60 = 9.6 resulting each Core to calculate 9 rows of pixels (only the last row calculate more). In the case of 64 calculating units, 576/64 = 9, meaning each Core also calculate 9 rows which results in the same calculation unit time.

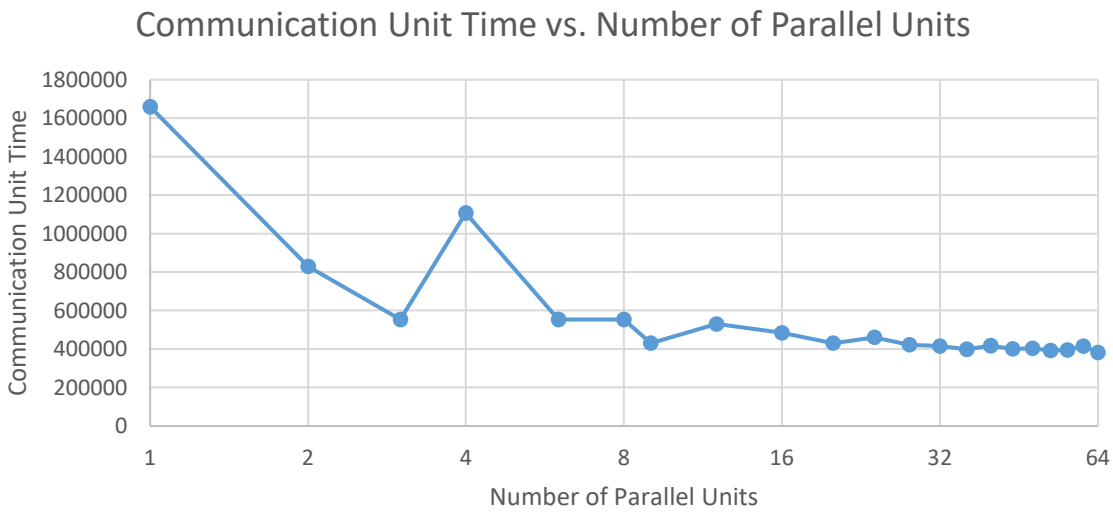

Figure 25: Mandelbrot on Checkerboard Model Communication Unit Time vs. Number of Parallel Units

Unlike the Mandelbrot 1.0 model which writes the entire image to memory after every parallel unit finishes their job, Mandelbrot on Checkerboard has multiple Cores pushing and popping images from different on-chip memories across the columns. Expanding the width and height of the Checkerboard also increases the amount of memories, allowing more reading and writing to occur at the same time. One unusual data point occurs at 4 parallel units with a 2x2 layout, because the mapping method used Mandelbrot on Checkerboard, the 1x3 layout model has one more column than the 2x2 layout. As shown in Figure 26, even though 2x2 layout has one more calculation unit, it has a longer pipeline for the image to go through from the top off-chip memory to bottom off-chip memory.

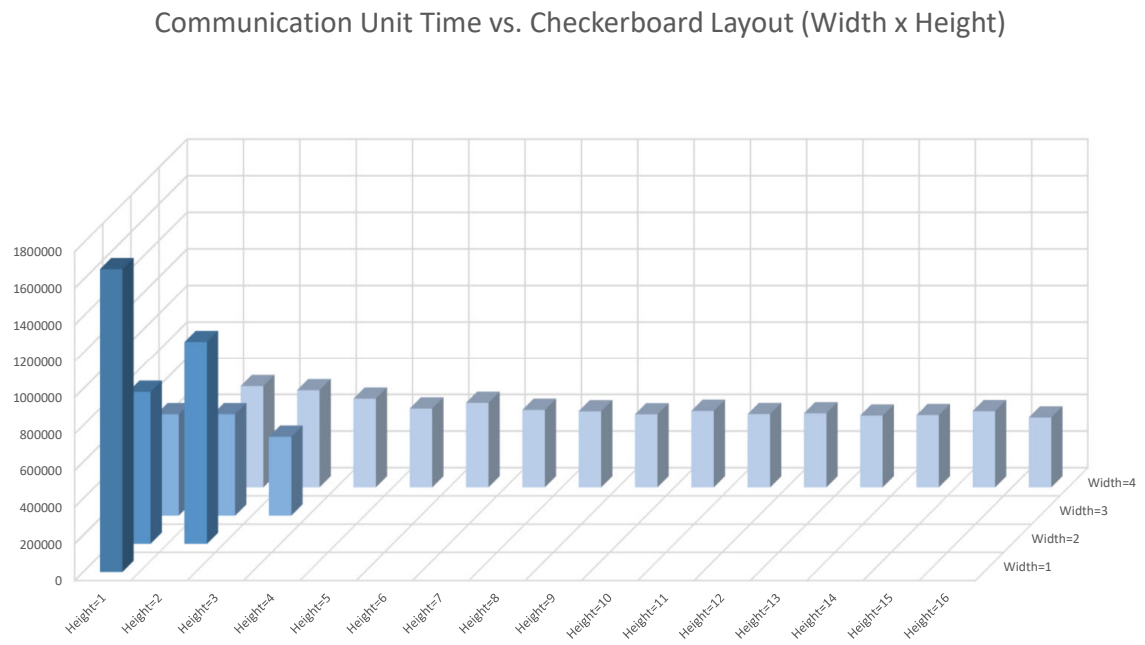

Figure 26: Mandelbrot on Checkerboard Model Communication Unit Time vs. Checkerboard Layout

Figure 27 and Figure 28 displays data from both the 1.0 Mandelbrot model and Mandelbrot on GPC model for comparison.

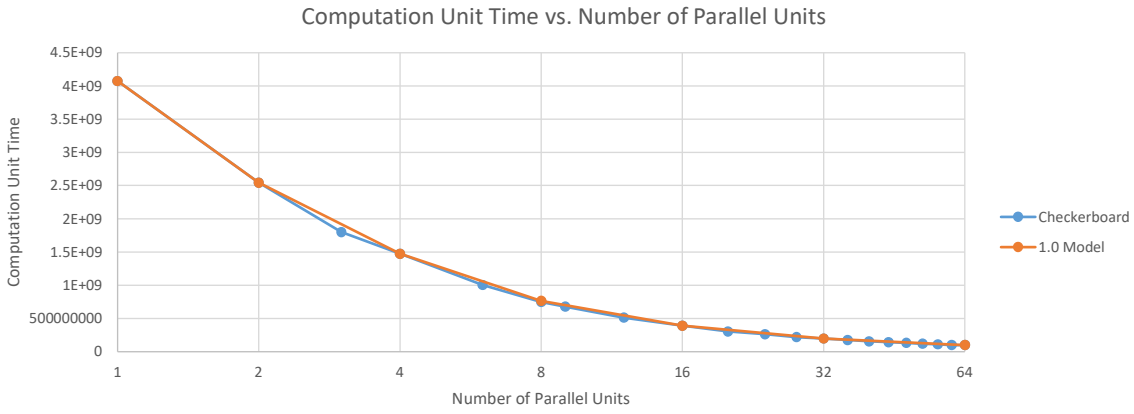

Figure 27: Computation Unit Time vs. Number of Parallel Units Comparison

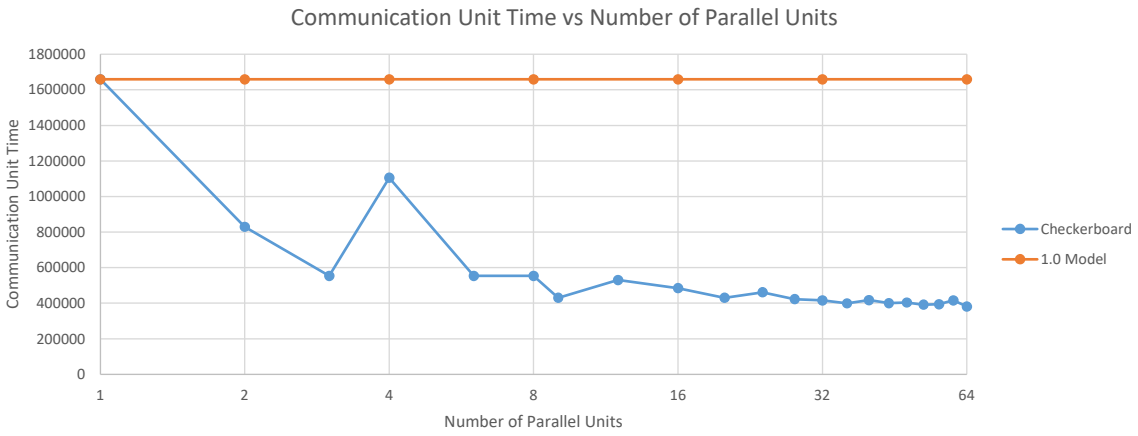

Figure 28: Communication Unit Time vs. Number of Parallel Units Comparison

Comparing both graphs reveals that the computation unit time curve of both models are almost identical proving that the Mandelbrot on Checkerboard model is able to match with the scaling of the Mandelbrot Set Visualization 1.0 Model with ideal scalability. The communication unit time of the Mandelbrot on Checkerboard model is able to out-perform the Mandelbrot Set Visualization 1.0 Model, showing that the Checkerboard architecture is indeed scalable.

6 Conclusion and Future Work

The Checkerboard model, while a high-level abstract software model, is a good starting point for more complex and accurate SystemC models for a scalable Checkerboard Grid of Processing Cells architecture. This Checkerboard project [1] at its current state, has the ability to automatically generate models with varying layout from 1 by 1 all the way up to 16 by 16. It is also shown to be stable in other mapping projects, i.e. a APNG encoder [26] and a JPEG encoder [20]. With such versatility and stability, the Checkerboard model offers a stable and flexible platform for more simulations and enables further explorations on the viability of the Checkerboard architecture.

Although the architecture at this stage is rather complex with many cores and many more memories, the experimental result from the mapped Mandelbrot on Checkerboard show that the Checkerboard architecture is indeed scalable.

In the future, we plan to design a custom compiler that can perform the mapping of any given software automatically.

References

[1] Yutong Wang. A Scalable SystemC Model of a Checkerboard Grid of Processing Cells. UCI Library, May 2022.

[2] Open SystemC Initiative, http://www.systemc.org. *Functional Specification for SystemC 2.0*, 2000.

[3] Rainer Dömer. A Grid of Processing Cells (GPC) with Local Memories. Technical Report Technical Report 22-01, UCI, Center for Embedded and Cyber-Physical Systems, April 2022.

[4] Robert Brooks and Peter Matelski. The dynamics of 2-generator subgroups of psl(2,c). *Irwin Kra (ed.)*, 1978.

[5] John von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, June 1945.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012.

[7] Wikipedia contributors. Modified harvard architecture — Wikipedia, the free encyclopedia, 2022. [Online; accessed 22-May-2022].

[8] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2011.

[9] Michael Bedford Taylor, Jason Sungtae Kim, Jason E. Miller, Fae Ghodrat, Ben Greenwald, Paul R. Johnson, Walter Lee, Albert Ma, Nathan R. Shnidman, David Wentzlaff, Matthew I. Frank, Saman P. Amarasinghe, and Anant Agarwal. The raw processor: A composeable 32-bit fabric for embedded and general purpose computing. 2001.

[10] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[11] Charlie Demerjian. A look at the 100-core Tilera Gx. https://www.semiaccurate.com/2009/10/29/look-100-core-tilera-gx/, 10 2009. [Online; accessed 30-May-2022].

[12] Tilera. Manycore without Boundaries: TILE64 Processor. http://www.tilera.com/products/processors/TILE64. [Online; accessed 30-May-2022].

[13] Tilera. Manycore without Boundaries: TILEPro64 Processor. http://www.tilera.com/products/processors/TILEPRO64. [Online; accessed 30-May-2022].

[14] The tile processor™ architecture: Embedded multicore for networking and digital multimedia. In *2007 IEEE Hot Chips 19 Symposium (HCS)*, pages 1–12, 2007.

[15] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskote, and Nitin Borkar. An 80-tile 1.28tflops network-on-chip in 65nm cmos. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 98–589, 2007.

[16] Li-Shiuan Peh, Stephen W. Keckler, and Sriram Vangal. *On-Chip Networks for Multicore Systems*, pages 35–71. Springer US, Boston, MA, 2009.

[17] Jim Held. “single-chip cloud computer”, an ia tera-scale research processor. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, pages 85–85, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[18] Intel Labs. Introducing the Single-chip Cloud Computer. https://simplecore.intel.com/newsroom-en-eu/wp-content/uploads/sites/13/2010/05/Intel_SCC_whitepaper_4302010.pdf. [Online; accessed 30-May-2022].

[19] Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M. Baas. Kilocore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits*, 52(4):891–902, 2017.

[20] Arya Daroui. A loosely timed TLM 2.0 Model of a JEPG encoder on a Checkerboard GPC. UCI Library, March 2022.

[21] Benoit B. Mandelbrot and Benoit B. Mandelbrot. *The fractal geometry of nature / Benoit B. Mandelbrot*. W.H. Freeman New York, updated and augmented [ed.] edition, 1983.

[22] Z. Cheng D. Mendoza R. Dömer G. Liu, T. Schmidt. Risc compiler and simulator, release v0.6.0: Out-of-Order Parallel Simulatable SystemC Subset. Technical report.

[23] Raka Jovanovic and Milan Tuba. A new visualization algorithm for the mandelbrot set. 01 2009.

[24] RISC open source. http://www.cecs.uci.edu/~doemer/risc.html.

[25] Ieee standard for standard systemc language reference manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2012.

[26] Vivek Bala Govindasamy. A TLM 2.0 Model of a PNG encoder on a Checkerboard GPC. UCI Library, March 2022.