Many-Core Final Project Report

Name: 吳浩寧

ID: 105062635

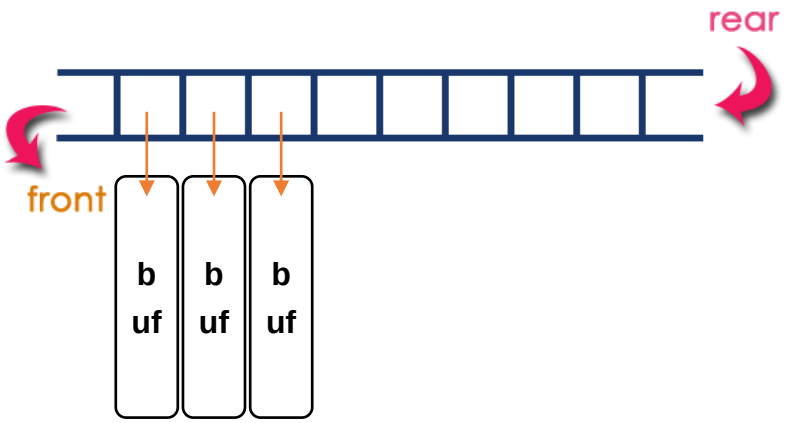
Introduction

這次作業主要分為兩個部分，都是以多核心的 rocket chip 做為平台。第一部分，我以助教們已經做出來的 pthread library 為基礎，在 shared memory 的架構上實做簡化版本的 message passing API；第二部分，則是將期中做的題目：Mandelbrot Set 改成平行化的版本，並將部分程式碼放到 rocc accelerator 上面使用硬體加速。

Implementation details

Message Passing API

這一部分我實作方式為為每個 core 在 shared memory 上分配一塊記憶體空間，作為接受訊息的 buffer，如下圖，每接收到一筆資料都會放到一塊 buffer，採用 FIFO 的方式存取，每當 process 執行 recv()時就會檢查自己的 queue 中有沒有資料。



主要修改的部分為 common/util.h，實作的 functions 有三個

1. int get_rank()

   使用 gcc inline assembly 的語法，使用 RISC_V 中的 csrr 指令讀取 mhartid 這個硬體的 thread id，供 send()和 recv()使用。

```
static int get_rank(){
    int r;
    __asm__ __volatile__ (
        "csrr %0, mhartid"
        : "=r" (r)
        ::"memory"
    );
    return r;
}
```

2. void ilib_send(int rank, const void* buffer, size_t size)
   rank 為目標 core 的 id；buffer 為存傳送資料的位址，size 則為資料長度。在送資料的時候要先將 receiver 的 queue 用 mutex 鎖住，避免同時寫入資料造成資訊錯亂。

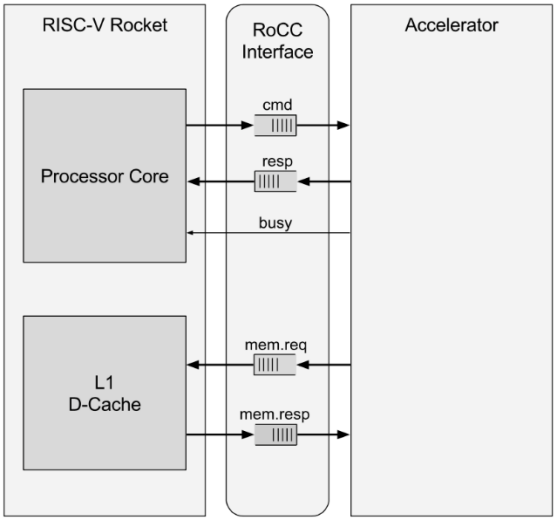
```
static void ilib_send(int rank, const void* buffer, size_t size){
    int myrank = get_rank();
    mutex_lock(&qm);
    memcpy(mem[rank].recv_buf[mem[rank].head], buffer, size);
    mem[rank].recv_rank[mem[rank].head] = myrank;
    mem[rank].head = (mem[rank].head+1)%QUEUE_SIZE;
    mutex_unlock(&qm);
}
```

3. int ilib_recv(const void* buffer, size_t size)
   回傳的整數為接收資料的來源 core id，其他參數同 send()，recv 因為同時只可能有一個 process 呼叫因此不需使用 mutex。

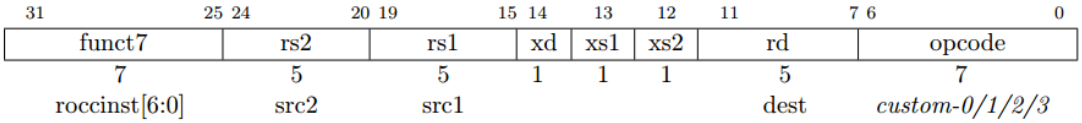
```
static int ilib_recv(const void* buffer, size_t size){
    int sender;
    int rank = get_rank();
    while(1){
        if(mem[rank].tail != mem[rank].head){
            memcpy(buffer, mem[rank].recv_buf[mem[rank].tail], size);
            sender = mem[rank].recv_rank[mem[rank].tail];
            mem[rank].tail = (mem[rank].tail+1)%QUEUE_SIZE;
            return sender;
        }
    }
}
```

RoCC

RoCC 全名為 Rocket Custom Coprocessor interface，是 rocket core 與 accelerator 之間溝通的介面，彼此之間溝通的信號主要包括以下幾個：cmd 負責傳送 rocket core 的指令與 source registers 的內容，resp 則回傳 destination register 的值，busy 是供 process 判斷 accelerator 是否正在進行計算用的信號，此外，RoCC 對 L1 cache 也同樣有一組 cmd、resp 的訊號，用來對記憶體進行存取。其他功能包括對 processor 發出 interrupt 或透過 Tilelink 直接對外部 memory 進行存取等等。



RoCC 的指令格式如下：其中 xd、xs1、xs2 是用來判斷哪些欄位有被使用，其他則和一般的 instruction 大同小異。



首先，改 rocketchip/Configs.scala，將平台改成 4 個 cores，並使用自己修改過的 RoccExample。

```
class DualCoreConfig extends Config(
  new WithNCores(4) ++ new WithRoccExample ++ new WithL2Cache ++ new BaseConfig)
```

修改 coreplex/Configs.scala，並使用他原本多餘的 custom3 (b1111011)作為我們新加指令的 opcode。

```
class WithRoccExample extends Config(
    (pname, site, here) => pname match {
        case BuildRoCC => Seq(
            ...
            RoccParameters(
                opcodes = OpcodeSet.custom3,
                generator = (p: Parameters) => Module(new MandelbrotExample()(p))))
        case RoccMaxTaggedMemXacts => 1
        case _ => throw new CDEMatchError
    })
```

接著便是修改 rocket/rocc.scala，原本預計像期中一樣將 Mandelbrot 計算的整個 while 迴圈放進去加速，不過若將 reg 傳進來的值當作 SInt，compile 時發現會出現：chisel3.internal.ChiselException: cannot connect chisel3.core.SInt@2d35f and chisel3.core.UInt@2ca2c 的錯誤訊息，因此最後只選了一行平方和的運算進行加速。其他解決方式可能要在接收輸入後先進行正負號的判斷，且每次計算完都得再進行坐一次判斷，因此麻煩許多。

程式碼中 s_idle 用來表示 accelerator 現在可以接收 processor 的指令，s_cal 代表正再進行計算，s_resp 代表將回傳值放回 destination register 傳回 processor，只會維持一個 cycle。當接收到 cmd 時便會將一些變數初始化。

```
val s_idle :: s_cal :: s_resp :: Nil = Enum(Bits(), 3)
val state = Reg(init = s_idle)

io.cmd.ready := (state === s_idle)
when (io.cmd.fire()) {
    req_rd := io.cmd.bits.inst.rd
    cx := io.cmd.bits.rs1
    cy := io.cmd.bits.rs2
    result := UInt(0)
    state := s_cal
}
```

主要計算部分，因為傳入的數有可能是負值，所以必須先使用 bit operation 的方式轉成絕對值，計算完後才可進行平方和運算。

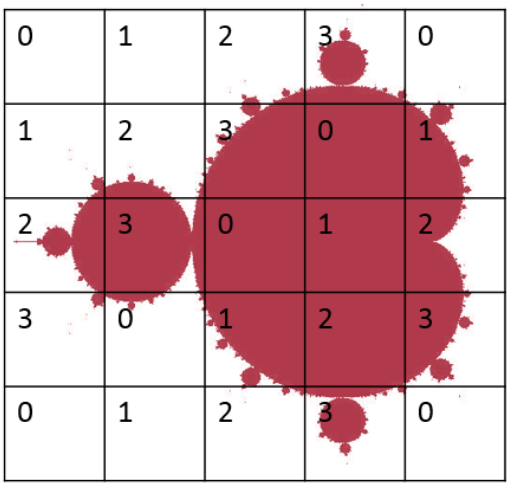
```
val px = cx^(cx >> 31)-(cx >> 31)
val py = cy^(cy >> 31)-(cy >> 31)

when (state === s_cal) {
    state := s_resp
    result := px*px + py*py
}
```

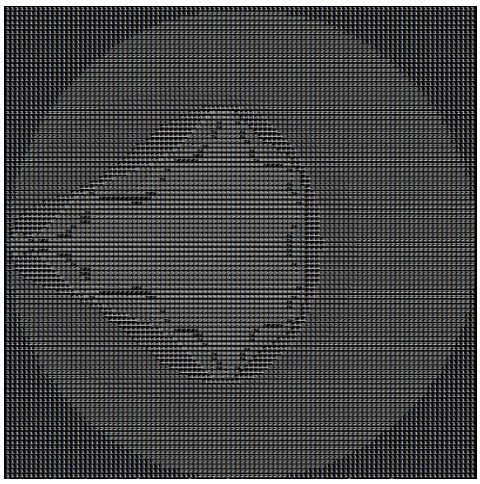
計算完 state 變為 s_resp，將 valid 拉起，資料放進 data 傳回 processor。

```
io.resp.valid := (state === s_resp)
io.resp.bits.rd := req_rd
io.resp.bits.data := result
io.busy := (state =/= s_idle)
```

Software

由於這次要在 many cores 的平台上運行自己的軟體，因此必須先將期中的程式碼轉成平行版本，我採用的工作分配方式如下，以 4 個 cores 為例，將圖形切割成數塊正方形，從左上角開始輪流分配工作給每個 core，如此可以避免如中央顏色特別深，導致計算量分配不均的問題。





Shared Memory Version：直接將運算結果寫回 shared 的二維陣列中，透過第一個 core 將圖形印出。

Message Passing Version：使用前面自己實作的 function，每個 core 運算完必須將結果送到第一個 core 的 private memory，再一次印出。

```
if(rank != 0){
    int send_buf[3];
    send_buf[0] = i;
    send_buf[1] = j;
    send_buf[2] = repeats;
    ilib_send(0, &send_buf, 3*sizeof(int));
}

if(rank == 0){
    int recv_buf[3];
    ilib_recv(&recv_buf, 3*sizeof(int));
    color_buf[recv_buf[0]][recv_buf[1]]=recv_buf[2];
}
```

RoCC Version：將註解掉的部分改為自己的 instruction，其中 mandelbrot()定義在 mandelbrot.h 中，另外使用網路上找到的 rocc.h，已經幫我們處理好 inline assembly 轉換的部分。

```
//lengthsq = z.real*z.real + z.imag*z.imag;
mandelbrot(result, (int)(z.real*100), (int)(z.imag*100));
lengthsq = (double)result/10000;
```

Discussions & Conclusions

在實作中發現在 rocket chip 的軟體部分，不同地方插入 printf()會造成程式卡住 output，任意調動計算的 pixels 數量與 printf()位置皆會有非預期的狀況產生，因此最後固定採用 width=100、height=100，使用 4 個 cores 來測試，並將最後的 printf()關掉。分別用 shared memory、message passing；有無使用 RoCC 加速的組合來比較結果。

	Cycles	Cache hit	Cache miss
SM	705351	3426	745
SM+RoCC	856902	3472	744
MP	10825971	406960	172187
MP+RoCC	10865952	404294	172220

經實驗後發現，message passing 的版本慢了非常多，雖然是預料之中的結果，因為 Mandelbrot Set 的計算是 embarrassingly parallel ，因此在使用 shared memory 的方式時完全不必考慮 race condition 的問題，而且算完結果可以直接存到記憶體中，不用像 message passing 的版本要花上取得 mutex 的 communication 時間。

另外，使用了 RoCC accelerator 的版本反而都慢了一些，不像期中有顯著的加速，可能是因為這次加速的部分不多，從 processor 傳送指令到 RoCC accelerator，再接收計算結果的時間反而成了 overhead 的關係。