Spring 2018 ECE 8823: GPU Architectures **Moumita Dey**

Baseline Project: PDOM Algorithm for Branch Divergence in GPUs

Summary

In this project, PDOM algorithm for branch divergence in GPUs was implemented on HARP emulator. The algorithm was tested against three types of test cases: divergent warps, nested divergent warps and divergent loops. Each of these test cases was prepared in assembly code and the binary was generated using the compiler provided. The test cases support two warps, each containing four threads. The details of the implementation and result is described below.

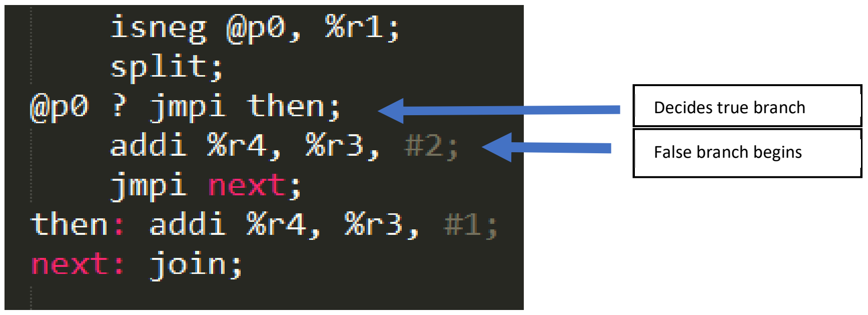
Overview of PDOM Algorithm

The PDOM algorithm is based on keeping a track of the next PC, reconvergence points and the masks of the threads active for that branch, every time the program encounters a divergent branch. The PDOM algorithm was implemented using the assembly instructions SPLIT and JOIN. The SPLIT instruction splits the warp depending upon the predicate register values for the threads and stores the relevant PDOM information to the stack. First, the *true* branch is executed, followed by the *false* branch. Each branch, after its branch's execution, waits at the JOIN instruction. The JOIN instruction handles the allocation of the divergent branch's PC and ensures the active masks are correct. It is also responsible for removing the top-of-stack (TOS) entry when that divergent branch is done with its execution.

Implementation of PDOM Algorithm

The stack of PDOM was implemented using vector data structures. The reconvergence PC is difficult to determine during real-time execution. Hence, only the next PC and the active mask were stored in the vector stack.

PC calculation:



```
      isneg @p0, %r1;
      split;
@p0 ? jmpi then;
      addi %r4, %r3, #2;
      jmpi next;
then: addi %r4, %r3, #1;
next: join;
```

It was observed that the SPLIT instruction is followed by an instruction that sets the predicate. The value of the predicate register effectively is the active mask for the *true* taken branch. As shown above, once the *true* branch is taken, it jumps to *then*. The *false* branch is however, always the next instruction after the set predicate instruction. Hence, the next PC determination is easy. For the *true* branch, the next PC is of the instruction that follows SPLIT, whereas for the *false* branch, the next PC is of the instruction following that.

Active mask calculation:

Again, the value of the predicate registers directly gives the active mask for the *true* branch. The active mask of the *false* branch is effectively the inverse of the *true* branch. However, this is not the case for nested divergent warps. To keep a track of the active mask of the parent warp, a new item was introduced within the *warp* structure of the emulator called *prevActiveMask_*, similar in behavior to the *activeThreads_* entity. *prevActiveMask_* stores the active mask of the ongoing warp, and only the threads that are active go inside the *execute* function. A local variable to the *execute* function called *nextActiveMask* keeps a track of the active mask for the next instruction.

When the SPLIT instruction is encountered, *prevActiveMask_* has the current active mask that is the active mask of the parent. Thus, now the active mask of the *false* branch can be obtained by:

False mask = *Parent* mask AND (! *True* mask)

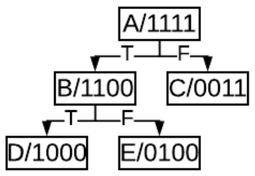
Both the masks and the next PCs are inserted at the beginning of the vector stack with the *true* branch's values above the *false* branch's values. The beginning of the vector stack behaves as the TOS. This works even for nested branches as only when one entire branch (with sub-branches) of the first SPLIT branch is complete, the second SPLIT branch begins. Hence, the TOS always contains the most recent parent branch's values and inserting the current branch's value at the beginning of the stack will be correct.

Reconvergence PC calculation:

To illustrate this, an example is shown below, with the stack contents after first and second branch respectively. On encountering the first branch, the *true* and the *false* branches are pushed into the stack with the *true* branch at TOS. The starting PCs of these branches are also pushed under the NextPC entry. When the second branch is encountered, since the parent is B, it is at TOS. The active masks of D and E are obtained as explained before and inserted at the TOS.

During execution, first the D branch will reach JOIN. At JOIN, the TOS is first popped out, and then the current TOS' NextPC and Masks are allocated in the emulator. Thus, the execution now continues with branch E and its associated threads. Once branch E completes execution and reaches JOIN, allocating B's NextPC would be incorrect as it stores the start of B branch. Hence, if all the children have reached JOIN, instead of taking the NextPC value from the stack, the PC is simply incremented to the next instruction in line, and the active mask of the parent is copied to within the emulator. Thus now, the remainder portion of B executes with the active mask of B. Once B reaches JOIN, it pops out its entry from the stack, goes to the NextPC value of branch C and executes with C's associated threads.

Thus, there is no real "reconvergence PC", but the JOIN instruction with the help of stack keeps a track of the reconvergence PC.



NextPC	Mask
B	1100
C	0011

NextPC	Mask
D	1000
E	0100
B	1100
C	0011

Test Cases and Sample Runs

All the test cases were written with two warps and each warp with four threads. They can be further scaled to any number of warps or threads. The number of warps and threads were kept low for the sake of simplicity. To support threadID based control divergence, a new instruction `tid %dest` has been added to the ISA that sets the threadID to the destination register.

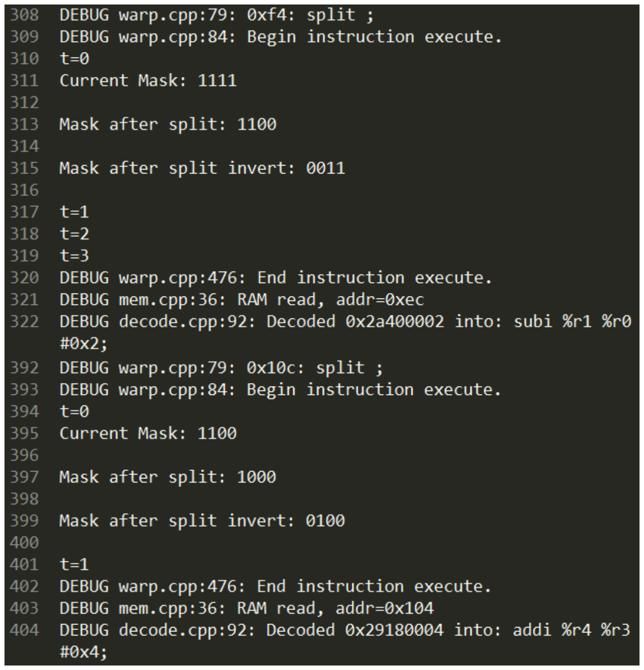
To run: `./miniharp.out test1.bin -t 4 -w 2 -r 8 -o output.log`

Divergent warp (test1.s):

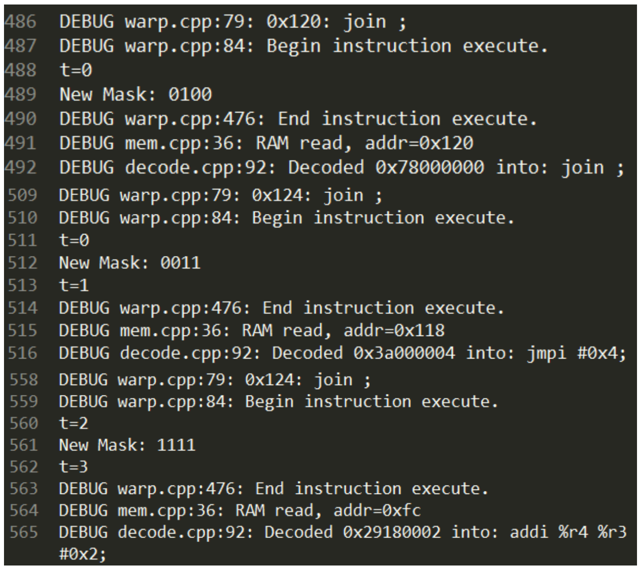
A simple program with two branches, split equally based on the threadID. The *true* branch adds #1 to the input value whereas the *false* branch adds #2 to the output value. First warp uses 0x2 as its initial value whereas second warp uses 0x5.

Nested divergent warp (test2.s):

Another simple program with nested branches as illustrated above. It builds up on test1.s. Debug support has been added to the emulator to keep a track of the diverging threads. The following figure shows snippets of the debug for a single warp at SPLIT and JOIN instructions. `t` denotes the threads entering that instruction. At SPLIT, the current mask and the masks being pushed are shown. The new mask assigned at JOIN is also printed as “New Mask”.



```
DEBUG warp.cpp:79: 0xf4: split ;
DEBUG warp.cpp:84: Begin instruction execute.
t=0
Current Mask: 1111

Mask after split: 1100

Mask after split invert: 0011

t=1
t=2
t=3
DEBUG warp.cpp:476: End instruction execute.
DEBUG mem.cpp:36: RAM read, addr=0xec
DEBUG decode.cpp:92: Decoded 0x2a400002 into: subi %r1 %r0
#0x2;
DEBUG warp.cpp:79: 0x10c: split ;
DEBUG warp.cpp:84: Begin instruction execute.
t=0
Current Mask: 1100

Mask after split: 1000

Mask after split invert: 0100

t=1
DEBUG warp.cpp:476: End instruction execute.
DEBUG mem.cpp:36: RAM read, addr=0x104
DEBUG decode.cpp:92: Decoded 0x29180004 into: addi %r4 %r3
#0x4;
```



```
DEBUG warp.cpp:79: 0x120: join ;
DEBUG warp.cpp:84: Begin instruction execute.
t=0
New Mask: 0100
DEBUG warp.cpp:476: End instruction execute.
DEBUG mem.cpp:36: RAM read, addr=0x120
DEBUG decode.cpp:92: Decoded 0x78000000 into: join ;
DEBUG warp.cpp:79: 0x124: join ;
DEBUG warp.cpp:84: Begin instruction execute.
t=0
New Mask: 0011
t=1
DEBUG warp.cpp:476: End instruction execute.
DEBUG mem.cpp:36: RAM read, addr=0x118
DEBUG decode.cpp:92: Decoded 0x3a000004 into: jmpi #0x4;
DEBUG warp.cpp:79: 0x124: join ;
DEBUG warp.cpp:84: Begin instruction execute.
t=2
New Mask: 1111
t=3
DEBUG warp.cpp:476: End instruction execute.
DEBUG mem.cpp:36: RAM read, addr=0xfc
DEBUG decode.cpp:92: Decoded 0x29180002 into: addi %r4 %r3
#0x2;
```

Divergent Loop (test3.s):

In this test case, each thread iterates over the loop depending upon its threadID+1. This test case essentially mimics a multiplication table. The first warp outputs the multiplication table for 0x2 whereas the second warp outputs the multiplication table for 0x5. It was observed that for a divergent loop, SPLIT is executed multiple times. Thus, when the last thread enters SPLIT, the extra values of the stack are cleared, and the original parent mask is retained. To point to this, “EXIT loop split” has been added to debug and the corresponding mask changes can be tracked.