Analyzing Graphics Processor Unit (GPU) Instruction Set Architectures

Kothiya Mayank[1], Hongwen Dai[1], Jizeng Wei[2] and Huiyang Zhou[1]
[1] Department of Electrical and Computer Engineering, North Carolina State University, {mvkothiy, hdai3, hzhou}@ncsu.edu
2 School of Computer Science and Technology, Tianjin University, weijizeng@tju.edu.cn

I. INTRODUCTION

Because of their high throughput and power efficiency, massively parallel architectures like graphics processing units (GPUs) become a popular platform for generous purpose computing. However, there is few study and analysis on GPU instruction set architectures (ISAs) although it is well-known that the ISA is a fundamental design issue of all modern processors including GPUs.

Among GPU instructions, control instructions are of particular interests due to control divergence. A typical way to handle divergent branches is to serialize each execution path through SIMD lane masking. The immediate post-dominator is used as the reconvergence point such that when it is reached, the program counter will be rewound back to execute another divergent execution path. The reconvergence points are managed in a stack and this approach is referred to as PDOM [3]. However, many details of such PDOM-based branch execution in commercial GPUs are not clearly described. For example, how high-level control structures such as 'while' and 'for' loops are implemented besides 'if then else' statements. Furthermore, unlike CPUs, the binary compatibility is less a concern. As GPUs evolve across different generations, their ISAs also evolve and it is intriguing to see how the ISA improves and how such evolution impacts the micro-architecture as well as overall performance. . In this paper, we use microbenchmarks to demystify how different types of control flows are processed in NVIDIA Fermi architecture and look into the ISA changes from NVIDIA Tesla to Fermi to examine their impact.

II. GPU ISA FOR CONTROL FLOW PROCESSING

In this section, we use the Fermi architecture to demystify control flow processing in modern GPUs. We look into different types of control flow to show how the semantics are encoded in the ISA and how they are processed in architecture. To simplify our discussion, we assume a warp size of 8 in our microbenchmark studies.

"IF...THEN ELSE" is a forward branch control flow structure and some "SWITCH...CASE" structures are also converted to series of "IF...THEN ELSE" statements. This forward branch is the most common conditional branch In Fermi, it is handled with a stack structure, called token stack using the Nvidia terminology [2]. A three-field stack entry is shown in Fig.1. The "active mask" field shows which threads in a warp are active. The "re-convergence PC" is the reconvergent point of the branch. The "entry type" field is used to differentiate various types of control flow instructions. Besides the token stack, each SM also has special purpose registers to store per warp "active mask" and "active PC" to assist control flow processing. The former disables the threads that are inactive and the latter is current program counter for a warp.

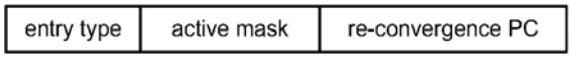
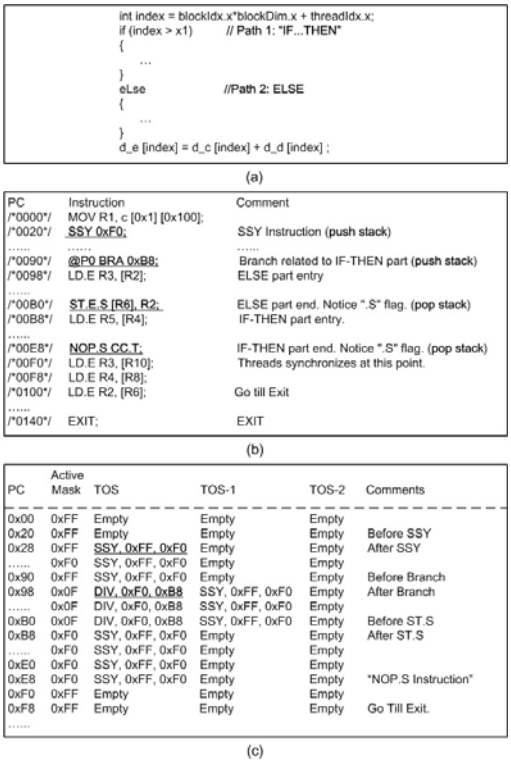


Fig. 1: The structure of a token entry in the token stack in the Fermi architecture.



```
int index = blockIdx.x*blockDim.x + threadIdx.x;
if (index > x1)        // Path 1: "IF...THEN"
{
   ...
}
eLse                   //Path 2: ELSE
{
   ...
}
d_e [index] = d_c [index] + d_d [index] ;
```

(a)

```
PC        Instruction                 Comment
/*0000*/  MOV R1, c [0x1] [0x100];
/*0020*/  SSY 0xF0;                   SSY Instruction (push stack)
.....     ......                      ......
/*0090*/  @P0 BRA 0xB8;               Branch related to IF-THEN part (push stack)
/*0098*/  LD.E R3, [R2];              ELSE part entry
.....
/*00B0*/  ST.E.S [R6], R2;            ELSE part end. Notice ".S" flag. (pop stack)
/*00B8*/  LD.E R5, [R4];              IF-THEN part entry.
.....
/*00E8*/  NOP.S CC.T;                 IF-THEN part end. Notice ".S" flag. (pop stack)
/*00F0*/  LD.E R3, [R10];             Threads synchronizes at this point.
/*00F8*/  LD.E R4, [R8];
/*0100*/  LD.E R2, [R6];              Go till Exit
.....
/*0140*/  EXIT;                       EXIT
```

(b)

PC	Active Mask	TOS	TOS-1	TOS-2	Comments
0x00	0xFF	Empty	Empty	Empty	
0x20	0xFF	Empty	Empty	Empty	Before SSY
0x28	0xFF	SSY, 0xFF, 0xF0	Empty	Empty	After SSY
.....	0xF0	SSY, 0xFF, 0xF0	Empty	Empty	
0x90	0xFF	SSY, 0xFF, 0xF0	Empty	Empty	Before Branch
0x98	0x0F	DIV, 0xF0, 0xB8	SSY, 0xFF, 0xF0	Empty	After Branch
.....	0x0F	DIV, 0xF0, 0xB8	SSY, 0xFF, 0xF0	Empty	
0xB0	0x0F	DIV, 0xF0, 0xB8	SSY, 0xFF, 0xF0	Empty	Before ST.S
0xB8	0xF0	SSY, 0xFF, 0xF0	Empty	Empty	After ST.S
.....	0xF0	SSY, 0xFF, 0xF0	Empty	Empty	
0xE0	0xF0	SSY, 0xFF, 0xF0	Empty	Empty	
0xE8	0xF0	SSY, 0xFF, 0xF0	Empty	Empty	"NOP.S Instruction"
0xF0	0xFF	Empty	Empty	Empty	
0xF8	0xFF	Empty	Empty	Empty	Go Till Exit.
.....					

(c)

Fig. 2: (a) The CUDA code segment of "IF...THEN ELSE"; (b) The assembly code of "IF...THEN ELSE" based on the token stack; (c) The contents of the token stack.

Next, we construct a microbenchmark containing the IF...THEN ELSE" structure. Fig. 2a and Fig. 2b show the CUDA code segment and native assembly code containing the "IF...THEN ELSE" structure for Fermi. Fig. 2c shows how the stack content is updated. In order to support divergence and re-convergence, a control instruction, 'SSY' at PC = 0x0020, is used in Nvidia GPUs. The operand of the 'SSY', 0xF0, specifies a reconvergence point. When this instruction is executed, a token is pushed onto the stack. This token includes the "active mask" copied from the active mask register, the "re-convergence PC" extracted from the SSY operand, and the "entry type" set as 'SSY'. Then, the conditional branch instruction "@P0 BRA 0xb8" is executed, where P0 is a

predicate register storing the branch condition. If branch divergence exists, a token, "DIV, 0xF0, 0xB8", is pushed onto the stack. It fills the "active mask" with the predicate register P0 (0xF0), the "reconvergence PC" field with the branch target address (0xB8), the "entry type" with 'DIV'. Then, the not-taken path is executed first by setting the active mask register as the inverse of P0 and the active PC register as branch PC + 8. Such execution continues until a stack pop operation is encountered. In the Fermi ISA, a stack pop operation can be appended to any instruction with a '.S' flag rather than a specific pop instruction. In this example, the pop flag is added to the store instruction at PC = 0x00B0, marking the end of the "ELSE" path. The stack pop operation takes the top of the stack to set the active mask register and the active PC register. As a result, the taken path is selected and executed until another pop operation at PC = 0x00E8. At this point, the top of stack contains "SSY, 0xFF, 0xF0". The pop operation again sets the active mask register and the active PC register and all the divergent threads are reconverged at the reconvergence point. The above process is similar to the PDOM approach except that the Fermi ISA explicitly encodes stack push and pop operations in the instruction stream. Such encoding simplifies hardware managing the stack and eliminates the need for hardware logic to check on whether the divergence point is reached.

TABLE I: The control instructions supported in the Fermi ISA.

Mnemonic	Description	Semantic
SSY	Set Synchronization	SSY 0x00F0;
BRA	Direct Branch	@P0 BRA 0x00B8;
BRA.U	Uniform Branch	@P0 BRA.U
CALL	Function Call	CALL 0x0168;
RET	Returning from Function Call	RET;
BRX	Indirect Branch	BRX R2;
PRET	Used with BRX	PRET 0x0060;
BRK	Break	BRK; or @P0 BRK;
PBK	Used with BRK	PBRK 0x0180;
".S" flag*	Stack Pop Operation	NOP.S CC.T;

*".S" flag can be attached to any normal instructions in GPUs.

Besides the "IF...THEN ELSE" structure, there are also many other control flow structures such as while/for loops, function calls, breaks, etc. Different instructions are introduced for these control flows. In Table I, we list the control instructions supported by the Fermi ISA.

III. ISA EVOLUTION ACROSS GPU GENERATIONS

We use the simulator GPGPU-Sim V3.2.1 [1] to analyze the ISA impact on application performance. While GPGPU-Sim supports both PTX and SASS execution for the Tesla architecture, it does not support the native Fermi ISA. So, we extend the GPGPU-Sim frontend to support the Fermi ISA.

Next we present a case study on the matrix multiplication (MM) benchmark to illustrate how the GPU ISA evolves from the Tesla to Fermi architect. Fig.3 presents a CUDA code section of the MM benchmark and the corresponding Tesla/Fermi ISA codes. The underlined code highlights the differences due to the ISA difference. As shown in the figure, to compute the index of the array access, "*A[a +WA∗ ty + tx]*", five instructions are needed in the Tesla ISA. With the 'ISCADD' (Integer Scaled Add) instruction, the instruction count is reduced to three in the Fermi ISA. By virtue of the 'ISCADD' instruction, the overall dynamic instruction count of MM in the Fermi ISA is decreased by 11.8% compared to that using the Tesla ISA.

From our case studies, we observe that evolving from Tesla ISA, the Fermi ISA introduces more complex ALU instructions, more flexible memory access modes, and more efficient control flow management. Both complex ALU instructions and flexible memory access modes lead to more instruction fusion. Efficient control flow such as predication can reduce the instructions for explicit stack management. Fig.4a presents the normalized dynamic instruction count (IC) for the benchmarks using two GPU ISAs. With the Fermi ISA, the average dynamic IC is 22.6% less than that with the Tesla ISA. Fig. 4b reports the normalized execution time of the benchmarks running on the same hardware but using different ISA. It shows that the ISA evolution results in a 15.4% performance improvement on average.

```
aBegin = WA * BLOCK_SIZE * by;
aEnd   = aBegin + WA - 1;
aStep  = BLOCK_SIZE;
bBegin = BLOCK_SIZE * bx;
bStep  = BLOCK_SIZE * WB;
cSub = 0;
for (int a = aBegin, b = bBegin; a <= aEnd; A += aStep, b += bStep){
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    AS(ty, tx) = A[a + WA * ty + tx];
    BS(ty, tx) = B[b + WB * ty + tx];
    __synchthreads();
    ……}
```

(a)

```
Tesla ISA:
……
SHL R1, R2, 0x4;
IADD32 R5, R2, R0;
IADD32 R4, R1, R4;
SHL R5, R5, 0x4;
IADD32 R4, R3, R4;
IADD32 R5, R3, R5;
SHL R4, R4, 0x2;
IADD R1, R3, R1;
SHL R5, R5, 0x2;
IADD R4, g [0x4], R4;   //Address for A[]
……
```

```
Fermi ISA:
……
ISCADD R4, R3, R4, 0x4;
IADD R5, R2, R3;
IMAD.U32.U32 RZ, R1, RZ, RZ;
IADD R4, R12, R4;
ISCADD R6, R5, R12, 0x4;
ISCADD R5, R4, c [0x0] [0x20], 0x2;
// Address for A[];
ISCADD R4, R6, c [0x0] [0x24], 0x2;
```

(b)

Fig. 3: (a) The CUDA segment of the MM benchmark; (b) The assembly code of the MM using the Tesla and Fermi ISAs.

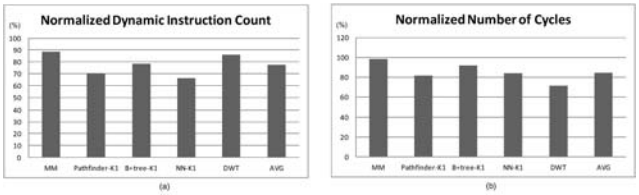


Fig. 4: (a) The dynamic instruction count of benchmarks using the Fermi ISA normalized to that using the Tesla ISA; (b) The number of cycles of benchmarks using the Fermi ISA normalized to that using the Tesla ISA.

References

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," In International Symposium on Performance Analysis of Systems and Software (ISPASS), 2009.

[2] B. W. Coon et al, "Processing an indirect branch instruction in a SIMD architecture", United States Patent US7,761,697 Bl, Assignee NVIDIA Corporation, issued Jun. 20, 2010.

[3] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation: Efficient mimd control flow on simd graphics hardware," ACM Trans. Archit. Code Optim., vol. 6, no. 2, pp. 7:1-7:37, 2009.