Supporting Virtual Memory in GPGPU without Supporting Precise Exceptions

Hyesoon Kim
Georgia Institute of Technology
hyesoon@cc.gatech.edu

ABSTRACT

Supporting precise exceptions has been one of the essential components of designing modern out-of-order processors. It allows handling exception routines, including virtual memory support and also supports debugging features. However, GPGPU, one of the recent popular scientific computing platforms, does not support precise exceptions. Here, in this paper, we argue that supporting precise exceptions is not essential for GPGPUs and we propose an alternate solution to provide virtual memory support without supporting precise exceptions.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: SIMD

General Terms

Design, Performance

Keywords

GPGPU, virtual memory, precise exception

1. INTRODUCTION

Supporting precise exceptions is one of the fundamental requirements in building an out-of-order processor. It decouples micro-architectural states from architectural states, thereby guaranteeing sequential execution to programmers. This is essential to provide easy debugging features, exception handlers, virtual memory, and context switching. Lately, GPGPUs have attracted many scientific computing programmers, because of the wide-SIMD execution units and high memory bandwidth. In some sense, GPUs are not much different from vector processors. However, GPUs are connected with a host computing platform so the host processors handle general-purpose computing components such as file I/O, user interface, and complex control flow graphs. More importantly, GPGPUs have not supported precise exceptions yet.

We can find several reasons why supporting for precise exceptions has not been implemented: First and foremost, GPUs are developed for graphics. Supporting precise exceptions is not needed at all and it is extremely expensive due to the high number of registers. Second, typical GPGPU applications consist of simple kernels. They are relatively easy to debug, and occasionally have already been debugged on CPUs. Third, debugging support features have been very limited anyway. Until recently CUDA did not supports GDB. Hence, we wonder whether it is necessary to support precise exceptions in future GPGPUs or other massively parallel architectures.

In this position paper, we argue that supporting precise exceptions is not required in GPGPUs if there are other ways of supporting virtual memory. The reasons are as follows:

MSPC'12 , Jun 16 - June 16 2012, Beijing, China

1. GPGPU processors are always connected with a host processor. The host processor can handle asynchronous exceptions such as timers, and other hardware interrupts.
2. There are several known techniques to handle exceptions such as software restart markers [2, 3] or idempotent processors [1, 5].
   (a) Software restart markers can be used to handle many exception handler programs by deferring execution of an exception handler program until a software restart marker (i.e., barriers). Software restart markers can be also used to support virtual memory in vector processors [2].
   (b) Idempotent processors can provide an easy mechanism to support context switch.
3. To support debugging features such as `break` in GDB, a barrier instruction can be inserted, or an in-order execution can be enforced.

There could be some restrictions to supporting various other exceptions such as unaligned memory accesses, arithmetic exceptions, etc. However, many of those exception handlers are not commonly practiced by programmers unlike virtual memory. Hence, if we have other low-cost solutions to support virtual memory in GPGPUs, we argue that it is not essential to support precise exceptions.

In order to support virtual memory in GPGPUs, we assume that another processor will execute the kernel code to bring a page from another storage. Hence, the essential feature to support virtual memory is the ability to resume the thread that generated an exception. Software restart markers, idempotent processors can also support some degree of exceptions but their exception support is of too coarse granularity. Here, we propose another solution that provides fine-grained entry/exit points than previous mechanisms.

The key idea is that the instructions between potential virtual memory exception generating instructions and the exception handler instructions are predicated. Current GPUs already utilize predicate execution to support control divergence. Hence, supporting predicated execution is not costly. We provide the mechanism in more detail in the following section.

2. MECHANISM

The basic idea is that the compiler inserts exception check instructions in appropriate places. Similar to Intel's IA-64 speculative load and NaT bit [4], exception check instructions flag a set of state registers. The state register is used to predicate instructions, which are used to guard some instructions or trigger an exception handler routine. It is also built on the previous mechanisms of software restart markers and idempotent processors.

2.1 Basic Mechanism

The basic mechanism is composed of three instructions: `set start_marker`, `LD.pfchk`, `sw_call`. `set start_marker` indicates a place where a program can be restarted after a page fault exception handler is serviced. `sw_call` is composed of `barrier` and `call` instructions. When a processor fetches an `sw_call` instruction, it enforces an execution barrier. Instructions after `sw_call` can be fetched/renamed, but none of the instructions will be executed. `call` instructions invoke page fault handler. Implementing this execution barrier is very easy, but it reduces the benefit of a fully out-of-order scheduling processor.

An `LD.pfchk` instruction sets `pfbit`, when it generates a page fault. The `pfbit` registers behave like predicate registers in IA-64. Instructions that can potentially change program's states are predicated with `pfbit`. Similar to idempotent processors, instructions that can be safely reexecuted without changing the program's results do not need to be predicated. If all instructions are predicated, those instructions cannot be executed until the load instruction is completed, thereby degrading performance significantly. Hence, it is the compiler's job to reduce the number of predicated instructions.

We also do not want to convert all load and store instructions as `LD.pfchk` or `ST.pfchk` instructions. We like to have as few `set start_marker` and `sw_call` instructions. For statically allocated objects, the compiler can easily insert LD.pfchk/ST.pfchk whenever it is across page boundaries. We discuss the issues in some cases. Please note that the cases that can take advantage of idempotent regions are not discussed here.

2.2 Malloc Functions

Malloc functions should try to allocate memory at a page granularity to reduce the number of page fault check instructions. This will increase fragmentation, but many dynamic memory allocations use large size of memory.

2.3 Large Arrays

Whenever a program accesses a large array that is located across multiple pages, the program should include a check routine. If the memory access pattern is known at static time, the compiler can minimize the checking routine only to the page boundary. Figure 1 shows example code.

```
/* original C-code */
for (int ii=0; ii<N; ii++)
 a[ii] = b[ii]*2;


/* new code */
for (int ii=0; ii<N; ii++) {
 if (!(ii%kk)) {
      // kk = page size%(size of(a[0]))
      pfchk(&(a[0])+ii*kk));
      pfchk(&(b[0])+ii*kk));
  }
  a[ii] = b[ii]*2;
}

void pfchk(int addr) {
/* use intrinsics to insert assembly code */
 set start_marker;
 LD.pfchk(addr);
 (pfbit) sw_call(start_marker);
}
```

Figure 1: Array code example with LD.PFCHK in an array.

2.4 Stack Operations

Stack operations can be similar to the large array case. The address computation routine can be moved before the actual computation, and the page boundary check functions can be executed. The current mechanism cannot handle stack overflow or deep nested function calls. These problems will be studied in future work.

2.5 Pointers

The most challenging data structure is pointers or random data accesses. In that case, memory address computation cannot be easily precomputed. One solution is to replace every memory operation as a `pfchk` function, as shown in Figure 2 unoptimized pfchk code. In this case, every single loop iteration will be serialized because the `sw_call` function has an execution barrier. However, if we utilize predicated execution, `sw_call` can be placed outside of the loop code. When `LD.pfchk` sets the pfbit value, the `CMPNZ` instruction will be nop; therefore, it naturally exits the loop and executes the `sw_call` instruction.

2.6 Supporting Multiple PFCHK LDs

Just like other architecture registers, pfbit can also be renamed. When multiple loads are inside a loop, we need to have multiple architectural pfbit registers. Similar to IA-64 predicate registers (where there are 64 1-bit predicate registers), we can have multiple pfbits. Predicate AND/OR operations can be used to combine multiple predicate registers.

```
/* original c-code */
 /* init p */
while (p!=0){
 sum+=p->value;
 p=p->next;
}

/* original assembly code */
LOOP_START:
  ADD R2 R2 MEM[R1+offset1];
   // offset1 indicates value field
  LD R1 MEM[R1+offset2];
   // offset2 indicate next field
  CMPNZ R1, LOOP_START;

/* unoptimized pfchk code*/
LOOP_START:
  ADD R2 R2 MEM[R1+offset1];
  set sart_marker;
  LD.pfchk R1 MEM[R1+offset2];
  (pfbit) sw_call (start_marker);
  CMPNZ R1, LOOP_START;

/* optimized pfcheck code */
  set sart_marker;
  clear pfbit; // reset pfbit;
LOOP_START:
  (!pfbit) ADD R2 R2 MEM[R1+offset1];
  (!pfbit) LD.pfchk R1 MEM[R1+offset2];
  (!pfbit) CMPNZ R1, LOOP_START;
  (pfbit) sw_call (start_marker);
```

Figure 2: Pointer chasing code example with LD.PFCHK.

2.7 Page faults in Instruction Fetch

If an instruction fetch generates an exception, none of the instructions from the correct path have fetched or executed. So in this case, the hardware triggers a page fault handler after all the instructions that are in the pipeline are executed. This can be easily supported by hardware without any support from a compiler/ISA.

3. CONCLUSION

In this position paper, we discuss that supporting precise exceptions is not strongly needed for GPGPU computing platforms. We also propose using predicated execution to support virtual memory in GPUs. Although this solution requires support from both hardware and software (new ISA and compiler), we believe this option is more practical, compared to hardware support for precise exceptions. In our future work, we will detail the hardware mechanism and evaluate the performance implications of this approach. We will also investigate supporting non-trivial stack operations and global variables.

4. REFERENCES

[1] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th International Symposium on Microarchitecture (MICRO)*, December 2011.

[2] M. Hampton and K. Asanović. Implementing virtual memory in a vector processor with software restart markers. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 135–144, New York, NY, USA, 2006. ACM.

[3] M. J. Hampton. *Reducing Exception Management Overhead with Software Restart Markers*. PhD thesis, 2008.

[4] Intel Corporation. *IA-64 Intel Itanium Architecture Software Developer's Manual Volume 1: Application Architecture*, 2002.

[5] J. Menon, M. de Kruijf, and K. Sankaralingam. igpu: Exception support and speculative execution on gpus. In *Proceedings of 39th International Symposium on Computer Architecture(ISCA)*, 2012.