Speculative Parallelization on GPGPUs

Min Feng Rajiv Gupta Laxmi N. Bhuyan
University of California, Riverside
{mfeng, gupta, bhuyan}@cs.ucr.edu

Abstract

This paper overviews the first speculative parallelization technique for GPUs that can exploit parallelism in loops even in the presence of dynamic irregularities that may give rise to cross-iteration dependences. The execution of a speculatively parallelized loop consists of five phases: scheduling, computation, misspeculation check, result committing, and misspeculation recovery. We perform misspeculation check on the GPU to minimize its cost. We optimize the procedures of result committing and misspeculation recovery to reduce the result copying and recovery overhead. Finally, the scheduling policies are designed according to the types of cross-iteration dependences to reduce the misspeculation rate. Our preliminary evaluation was conducted on an nVidia Tesla C1060 hosted in an Intel(R) Xeon(R) E5540 machine. We use three benchmarks of which two contain irregular memory accesses and one contain irregular control flows that can give rise to cross-iteration dependences. Our implementation achieves 3.6x-13.8x speedups for loops in these benchmarks.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – compilers

General Terms Performance

1. Introduction

Dynamic irregularities have been widely studied for high performance computing on General-Purpose Graphics Processing Units (GPGPUs or GPUs for short) [3–5]. Existing works have focused on optimizing the performance in the presence of such irregularities. However, in this work, we consider a new class of dynamic irregularities in loops that may cause cross-iteration dependences at runtime. Thus, presence of such dynamic irregularities prevents existing techniques from parallelizing the loops for GPUs. In particular, we have identified two types of dynamic irregularities that may dynamically cause cross-iteration dependences to arise. Next we illustrate them using examples.

Dynamic irregular memory accesses refer to memory accesses whose memory access patterns are unknown at compile time. They may result in infrequent cross-iteration dependences at runtime. Figure 1(a) shows an example, where each iteration of the loop reads $A[P[i]]$ and writes to $A[Q[i]]$. The memory access patterns of $A[P[i]]$ and $A[Q[i]]$ are determined by the runtime values of the elements in arrays P and Q. It is possible that an element in array A is read in one iteration and written in another at runtime, which results in a dynamic cross-iteration dependence. Since the memory access patterns are unknown at compile time, it is not possible to identify the dynamic dependences at that time. Therefore, the loop cannot be parallelized by any existing GPU compiler.

```
for (i=0; i<n; i++) {
    . . . = A[P[i]];
    A[Q[i]] = . . .;
}
```
(a) Irregular memory access

```
for (i=0; i<n; i++) {
    . . . = A[i];
    if ( A[i] ) A[i+1] = . . . ;
}
```
(b) Irregular control flow

Figure 1. Examples of dynamic irregularities that cause cross-iteration dependences.

Irregular control flows are introduced by conditional statements, which may cause execution of paths that may give rise to cross-iteration dependences at runtime, as illustrated in Figure 1(b), where each iteration of the loop usually only reads $A[i]$. In the loop, there is a conditional branch that guards a write to $A[i+1]$, which is to be read in the next iteration. The outcome of branch condition is determined by the runtime value of $A[i]$. If the condition is true in the current iteration, a cross-iteration dependence occurs between the current iteration and the next iteration. Since the value of $A[i]$ is unknown at compile time, there is no way to know at compile time in which iteration the branch condition will be true. Therefore, like the previous example, this loop cannot be parallelized by any existing GPU compiler.

In this paper, we propose a speculative execution framework for GPU computing. It is used to parallelize loops that may contain cross-iteration dependences caused by dynamic irregularities. The execution of a speculative parallel loop consists of five phases: scheduling, computation, misspeculation check, result committing, and misspeculation recovery. For efficiency, we develop a scheduling policy that is optimized for different types of cross-iteration dependences to reduce the misspeculation rate. We reduce the runtime overhead by performing misspeculation check on the GPU and utilizing its massive number of stream processors. We optimize the result committing procedure to reduce the size of data transferred between the CPU and GPU. Recovery is performed on the CPU for as few iterations as possible to minimize its runtime overhead.

Our preliminary evaluation was conducted on an nVidia Tesla C1060 hosted in a Intel(R) Xeon(R) E5540 machine. We used three benchmarks, where two benchmarks have loops with irregular memory accesses and one have loops with irregular control flows. Our implementation achieves 3.6x-13.8x speedup for the parallelized loops in these benchmarks.

2. Overview

Figure 2 gives the overview of executing a speculative parallel loop using GPUs. The procedure consists of five phases: *scheduling*, *computation*, *misspeculation check*, *result committing*, and *mis-*

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

Benchmark	Description	Function	LOC	Irregularities	% of time	Speedup
ocean	Boussinesq fluid layer solver	ftrvmt	150	irregular memory accesses	45%	3.62
trfd	two-electron integral transformation	intgrl	37	irregular memory accesses	6%	5.43
mdg	water molecule simulator	interf	208	irregular control flows	94%	13.76

Table 1. Benchmark summary. From left to right: benchmark name, name of the function where the loop is located, lines of code in the function, type of irregularities that cause cross-iteration dependences, percentage of total execution time taken by the loop, and speedup of the loop.

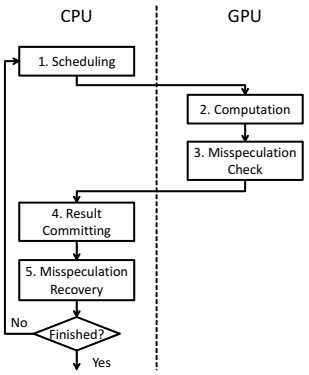

Figure 2. Execution framework of a speculative parallel loop with GPUs.

speculation recovery, among which *computation* and *misspeculation check* are performed on the GPU. The five phases are repeated until the entire loop is finished. We briefly describe the five phases as follows.

Scheduling. Upon entering a speculatively parallelized loop, the CPU needs to determine the proper number of iterations that will be executed on the GPU in the next phase. Assigning large number of iterations to the GPU may cause excessive misspeculations while assigning small number of iterations may limit performance while leaving the GPU under-utilized. Our scheduling policy adaptively adjusts the size of each assignment to minimize the misspeculation rate while keeping each assignment large enough to make full use of the massive parallel architecture on GPUs.

Computation. After scheduling, the GPU executes the iterations in parallel by speculating on the absence of cross-iteration dependence. To enable speculative execution, we need to track the irregular memory accesses and control flows during the computation.

Misspeculation check. Misspeculation check consists of two steps: detection and localization. Misspeculation detection is used to determine whether the iterations have been executed correctly. If misspeculation is detected, the misspeculation localization step is used to identify the iterations that were executed incorrectly. In addition, for speculative execution on GPUs, we need to identify the correct part of the results, which must be copied back to the CPU memory. To make misspeculation checks efficient, they are performed in parallel on the GPU. Since there is data parallelism in misspeculation checks, executing them on the GPU can lead to better performance.

Result committing. After misspeculation checks, we need to copy the results from the GPU memory to the CPU memory. For better performance, our runtime only copies the correct results using the information obtained through misspeculation check.

Misspeculation recovery. We need to re-execute the iterations where misspeculation occurs. We should re-execute on the CPU as few iterations as possible to minimize the recovery overhead. Executing more iterations on the GPU will get us better performance. Therefore, our runtime only re-execute on the CPU the misspeculated iterations on which other misspeculated iterations depend. Other misspeculated iterations will be executed on the GPU in the next assignment.

3. Preliminary Evaluation

This section presents our preliminary evaluation of the proposed speculative parallelization framework. We have developed a prototype implementation of our framework, whose core components consist of: a source-to-source translator and a runtime library. The source-to-source translator is based on OpenMPC [3], which is an OpenMP-to-CUDA compiler. The runtime library implements the core steps of our technique, i.e. scheduling, misspeculation check, result committing, and misspeculation recovery. We used an nVidia Tesla C1060 as the experimental platform. The device is connected to a host system consisting of Intel(R) Xeon(R) E5540 processors. The machine has CUDA 3.0 installed.

We evaluated our speculative parallelization framework on three benchmarks shown in Table 1. These benchmarks were obtained from the test benchmark suites for PIPS [1] and LLVM [2]. We selected them because they contain dynamic irregularities that may cause cross-iteration dependences at runtime. These benchmarks cannot be parallelized without speculation because all of them may have dynamic cross-iteration dependences due to irregularities.

The rightmost column of Table 1 shows the speedups of the speculative parallel loops in the three benchmarks. The baseline is the execution time of the sequential loops on the host system. Numbers higher than 1 indicate speedup. Overall, the speedups are between 3.62x and 13.76x, with 7.6x on average. *The speedups demonstrate the effectiveness of our framework for employing GPUs on irregular loops that may contain cross-iteration dependences*.

Acknowledgments

This research is supported by NSF grants CCF-0963996 and CCF-0905509 to UC Riverside.

References

[1] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ICS*, 1991.

[2] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[3] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP*, pages 101–110, 2009.

[4] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97, 2010.

[5] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, pages 369–380, 2011.