Adjusting Thread Parallelism Dynamically to Accelerate Dynamic Programming with Irregular Workload Distribution on GPGPUs

Chao-Chin Wu, National Changhua University of Education, Changhua, Taiwan

Jenn-Yang Ke, Tatung University, Taipei, Taiwan

Heshan Lin, Virginia Tech, Blacksburg, VA, USA

Syun-Sheng Jhan, Ling Tung University, Taichung, Taiwan

ABSTRACT

Dynamic Programming (DP) is an important and popular method for solving a wide variety of discrete optimization problems such as scheduling, string-editing, packaging, and inventory management. DP breaks problems into simpler subproblems and combines their solutions into solutions to original ones. This paper focuses on one type of dynamic programming called Nonserial Polyadic Dynamic Programming (NPDP). To run NPDP applications efficiently on an emerging General-Purpose Graphic Processing Unit (GPGPU), the authors have to exploit more parallelism to fully utilize the computing power of the hundreds of processing units in it. However, the parallelism degree varies significantly in different phases of the NPDP applications. To address the problem, the authors propose a method that can adjust the thread-level parallelism to provide a sufficient and steadier parallelism degree for different phases. If a phase has insufficient parallelism, the authors split threads into subthreads. On the other hand, the authors can limit the total number of threads in a phase by merging threads. The authors also examine the difference between the conventional problem of finding the minimum on a GPU and the NPDP-featured problem of finding the minimums of many independent sets on a GPU. Finally, the authors examine how to design an appropriate data structure to apply the memory coalescing optimization technique. The experimental results demonstrate our method can obtain the best speedup of 13.40 over the algorithm published previously.

Keywords: *Dynamic Programming, GPU, Optimization, Parallel Computing, Parallelism*

DOI: 10.4018/ijghpc.2014010101

INTRODUCTION

Dynamic programming (DP) is a popular method used to solve complex problems, including scheduling, string-editing, packaging, and inventory management (Grama, Gupta, Karypis & Kumar, 2003). Since it is believed DP will remain important into the future for science and engineering, it is one of Berkeley's 13 dwarfs, where a dwarf is defined as an algorithmic method capturing a pattern of computation and communication for a class of applications (Asanovic et al., 2006). The solution to a DP problem is usually expressed as a minimum (or maximum) of all possible alternate solutions. Dynamic programming can be classified into four categories based on the following two criteria (Grama, Gupta, Karypis, & Kumar, 2003). (1) If solutions to problems in a phase depend only on solutions to problems at the previous level, the dynamic programming is called serial, otherwise it is termed non-serial. (2) If the right hand side of the optimization equation contains only one recursive term, the dynamic programming is called monadic. Otherwise, it is termed polyadic. The four categories are (1) serial-monadic, used in the single source shortest path and 0/1 knapsack problems, (2) non-serial-monadic, used in the longest common subsequence problem and the Smith-Waterman algorithm (Smith & Waterman, 1981), (3) serial-polyadic, used in Floyd all pairs shortest paths problem, and (4) non-serial-polyadic, used in the Optimal Matrix Parenthesization problem (Hafeez & Younus, 2007; Lee, Kim, Hong & Lee, 2003) and the Zuker algorithm (Lyngso & Zuker, 1999; Tan, Sun & Gao, 2009).

Recently, many efforts have examined how to map the DP problems onto emerging graphics processing units (GPUs). The modern GPU is not only a powerful graphics engine, but also a highly parallel programmable processor (Nickolls & Dally, 2010). Today's GPUs use hundreds of parallel processor cores executing tens of thousands of parallel threads to rapidly solve large problems and they are now available in many PCs, laptops, workstations, and supercomputers. However, because the architecture and programming of the GPU are quite different from most other commodity single-chip processors, implementing parallel algorithms on GPUs requires different optimization techniques. For instance, CUDA (an acronym for Compute Unified Device Architecture) is a hardware and software coprocessing architecture for parallel computing enabling NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages (CUDA, 2012).

Due to the wide variety of problems solved using DP, it is difficult to develop generic parallel algorithms for them on GPUs. In this work, we focus on the non-serial-polyadic DP appearing in the Optimal Matrix Parenthesization problem. There are two distinct features which make a difference between non-serial-polyadic DP and the other three DP problems (Tan, Sun & Gao, 2009). First, the DP matrix for non-serial-polyadic DP is triangular, as opposed to being rectangular as in other DP problems, where the DP matrix is used to show all data dependence occurring during the computation. The property makes the optimization of memory accesses and load balancing difficult. Second, data dependence in non-serial-polyadic DP is dynamic, where the data dependence appears among nonconsecutive levels and the number of dependent elements varies for each subproblem.

There are two methods proposed that map the non-serial polyadic DP problems onto GPUs (Solomon & Thulasiraman, 2010; Rizk & Lavenier, 2009). The two methods mentioned above suffer from the following two problems. First, the number of threads created in part of phases cannot provide sufficient parallelism to fully utilize the massive parallel computing power of a GPU because each subproblem is executed by a thread. Second, finding the optimal cost for each subproblem is computed by only one thread, resulting in long execution time. To address these two problems, we propose an algorithm that can adjust the number of threads dynamically to fully utilize all the computing power in a GPU.

In our proposed algorithm, when we require more parallelism, a thread for each subproblem is split into multiple subthreads which together execute a parallel reduction algorithm to find