Mapping Serial-Monadic Dynamic Programming onto CUDA-Enabled GPUs

Chao-Chin Wu, Kai-Cheng Wei, Jian-You Lin
Dept. of Computer Science and Information Engineering
National Changhua University of Education
Changhua, Taiwan
email: {ccwu, kcwei}@cc.ncue.edu.tw

Wei-Shen Lai
Department of Information Management
Chienkuo Technology University
Changhua, Taiwan
email: weishenlai@gmail.com

***Abstract*—With the advent of high performance computational power, processing particularly complex scientific applications and voluminous data is more affordable. One of the hot parallel processors is general-purpose graphics processing unit (GPU), which has been widely adopted to accelerate various time-consuming algorithms. This work demonstrates how to apply a more condensed data structure and the interblock synchronization to efficiently map the serial-monadic dynamic programming onto GPUs.**

Keywords-dynamic programming; parallel computing; graphics processing unit; CUDA; data dependence.

I. INTRODUCTION

Dynamic programming (DP) is a popular method used to solve complex problems. DP can be classified into four categories: (1) serial-monadic, (2) non-serial-monadic, (3) serial-polyadic, and (4) non-serial-polyadic. Recently, many efforts have studied how to map the DP problems onto emerging general-purpose graphics processing units (GPUs), where nVIDIA has introduced CUDA (Compute Unified Device Architecture) to ease the programming on their GPUs for various kinds of applications [1]. CUDA is a hardware and software coprocessing architecture for parallel computing enabling nVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, and other languages.

Previously, we have investigated how to optimize the mapping of non-serial-polyadic DP problems onto CUDA-enabled GPUs, where the Optimal Matrix Parenthesization (OMP) problem was chosen as our study example. This work focused on how to optimize the mapping of serial-monadic DP problems onto nVIDIA GPUs and the 0/1 knapsack problem is adopted for this study.

Recently, Boyer and his colleagues proposed a DP approach with a compression mechanism to implement the 0/1 knapsack problem on a CUDA-enabled GPU [2]. The primary data structure used in their method, called the item selection table, is a 2-dimensional (2D) array and used to record if an item is selected or not for each capacity, C_i, where $0 < C_i < C$ and $C_i = i$, assuming the capacity of the knapsack is C. If there are N items in the problem, the dimension of the 2D array is $N \times C$. When N and C are both large integers, the 2D array requires a large amount of global memory space. To address this problem, they used one bit to represent if a certain item is selected or not. Next, each thread compressed the outcomes of every 32 stages into one integer, which is called the row-compression method. Furthermore, after analyzing the result values stored in the vectors, the row-compressed data, they found a large portion to the right hand side on the vector is filled with 1. On the other hand, the left hand side is filled with 0. Each thread recorded the indices of the boundaries of continuous 1's and 0's in the vector and then used the indices to replace the huge number of 1's and 0s' on both ends of the vector. In this way, the amount of the data to be transferred between the CPU and the GPU were reduced significantly. When the problem size becomes larger, the compression becomes more effective.

We observed two disadvantages of Boyer's method. First, although the compression method can reduce the amount of data transferred between CPU and GPU, if the item selection table is one-dimensional (1D) rather than 2D, the data can be minimized significantly. Second, the data in the shared memory cannot be reused because the Boyer's method invokes the same kernel one time for each stage of the DP problem, which can be addressed with the interblock synchronization. Based on the above observation, we propose a new approach to improve the performance of the knapsack problem on CUDA-enabled GPUs.

The remainder of the paper is organized as follows. Section II introduces the main idea of our proposed approach and details the key design issues. Section III gives the conclusion of our work.

II. PARALLEL APPROACH

The main idea of the new approach consists of two factors. First, the item selection table is 1D, the dimension is $1 \times C$. Adopting 1D structure not only can minimize the amount of data transferred between CPU and GPU but also can be stored in shared memory. Second, the interblock synchronization [3] is adopted to reuse the 1D item selection table in shared memory.

Although the 1D item selection table requires less memory space and can be fit into high-speed shared memory, the problem about this solution is its potentially exploitable parallelism is much less than that in the 2D one. The reason is explained as follows. To use dynamic programming with a 1D item selection table to solve the knapsack problem, items will be determined one by one whether one specific item is included in knapsacks of different capacities or not. One item will be considered during one stage of dynamic programming and one thread is assigned with one knapsack with one specific capacity. If one thread determines that the current item is included in its assigned knapsack, it writes the current item ID to the corresponding field of the 1D table. Note that one item is processed during one stage of dynamic programming. To process the current item, one thread needs the information, produced by another thread, about how the previous item is included for another knapsack of one specific capacity that is related to the weight of the current item. Consequently, one thread cannot process the current item until the required information is written to the corresponding table field. There exists a read-after-write dependency for the above operations. Furthermore, the thread cannot update its assigned table field with the current item ID until every thread requiring the result, produced by the thread in the previous stage, has all finished his reading. There exists a write-after-read dependency. To enforce the correct data flow, two synchronization points should be inserted after each of the above two kinds of dependency. However, the second dependency occurs only for the 1D table and it can be eliminated totally if a 2D table is used instead because the results of two items for the same knapsack are placed on two different locations. To address the problem of the write-after-read dependency, we allocate multiple buffers to store several versions of the 1D table. With multiple buffers, one thread can write the result for the next item to one buffer even though the result for the current item on another buffer has not read by other threads. The maximum number of buffers is dependent on the shared memory space. In this way, only the read-after-write dependency needs a synchronization to enforce data consistency.

Because the required data for one thread might be in shared memory on other streaming multiprocessors, all the threads have to write the results of the current item selection to the global memory and all the thread blocks should participate a barrier synchronization followed also. Instead of invoking the same kernel as many times as the number of DP stages, as suggested by the CUDA programming guide, we adopt the interblock synchronization mechanism that requires to invoke the kernel only one time. The advantage of invoking the same kernel again is that the data in shared memory is stale and cannot be reused. Consequently, the results in shared memory have to be written to the global memory before return from the kernel and then retrieve the results from the global memory to shared memory after the kernel is invoked again. On contrast, invoking the kernel only one time allows the results in shared memory can be reused repeatedly for all the DP stages. That is, the results stored in shared memory can be read by all the threads on the same streaming multiprocessors even though we proceed to next DP stage. Using the interblock synchronization is able to significantly reduce the amount of data transferred between shared memory and the global memory. Moreover, reusing the results in shared memory can shorten the latency of accessing shared memory. However, we still need to write the results to the global memory because the threads of other blocks cannot access the shared memory on different streaming multiprocessor. There also exists a write-after-read dependency on global accesses. Therefore, the 1D table also has multiple copies to eliminate the write-after-read dependency.

We further analyze the data dependence between thread blocks. The blocks except the last one have to write data to the global memory to allow the subsequent blocks to read, as shown in Figure 1. On the other hand, the blocks except the first one have to read results from the global memory to local registers. Note that all threads have to write the final version of the 1D item selection table from shared memory to global memory during the last DP stage. The result 1D item selection table in global memory will then be transferred back to CPU.

We use CUDA version 3.2 to implement different algorithms, including our approach and Boyer's approach, for the 0/1 knapsack problem. They are run on a system consisting of one AMD Phenom 9850 CPU and one nVIDIA GEFORCE 460. Our approach, adopting a 1D table, outperforms the previous work, as shown in Figure 2.

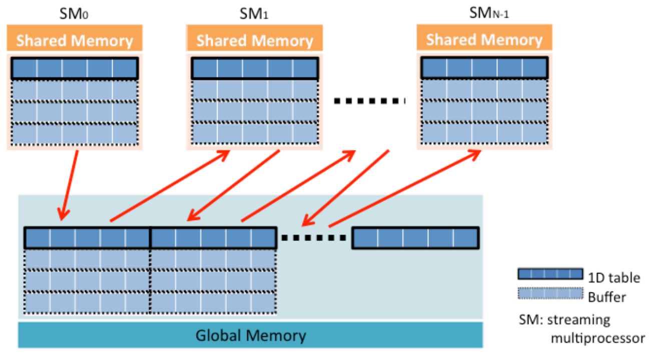

Figure 1. The addlocation of one-dimensional table and the buffers.

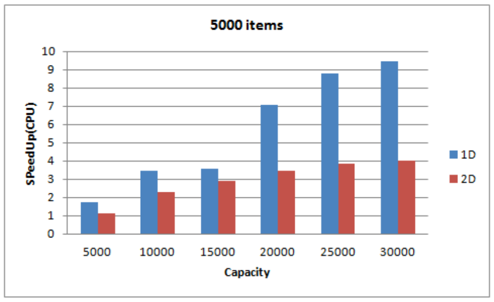

Figure 2. The speedup comparision between 1D and 2D tables.

In the legend of Figure 2, 1D and 2D represent our approach and the approach proposed by Boyer *et al.*, respectively. The number of items is 5000. The speedup is derived from dividing the execution time of the sequential CPU version by the execution time of one of the two GPU-based approaches. When the capacity is increased, the speedup is increased also. It is because either the 1D or the 2D Item Selection Table becomes larger, and the saved amount of global memory accesses becomes larger also. The size of 1D table is much smaller than that of 2D table. When using the 1D table, our approach can reduce the required memory traffic between CPU and GPU significantly. The larger the capacity, the more memory traffic can be saved by our proposed 1D table, resulting in better performance.

III. CONCLUSION

This work introduced how to use 1-dimensional data structure and the explicit inter-block synchronization to map the knapsack problem, an application of serial-monadic dynamic programming, on to a CUDA-enabled GPU. The results showed the proposed approach outperforms the previous work reported by Boyer *et al*.

ACKNOWLEDGMENT

The authors would like to thank the National Science Council, Taiwan, for financially supporting this research under Contract No. NSC103-2221-E-018-024.

REFERENCES

[1] NVIDIA GPU, http://www.nvidia.com/object/what-is-gpu-computing.html, retrieved: June 2015.

[2] V. Boyer, D. El Baz, and M. Elkihel, ”Solving knapsack problems on GPU”, Computers & Operations Research,vol. 39, no. 1, 2012, pp.42–47.

[3] C. C. Wu, K. C. Wei, and T. H. Lin, “Optimizing dynamic programming on graphics processing units via data reuse and data prefetch with inter-block barrier synchronization,” IEEE ICPADS, 2012 pp.45–52.