A Scalable GPU-based Approach to Accelerate the Multiple-Choice Knapsack Problem

Bharath Suri[1] Unmesh D. Bordoloi[2] Petru Eles[2]
[1]Delopt, India [2]Linköpings Universitet, Sweden
e-mail: bharath_s@delopt.co.in, {unmesh.bordoloi, petru.eles}@liu.se

***Abstract*—Variants of the 0-1 knapsack problem manifest themselves at the core of several system-level optimization problems. The running times of such system-level optimization techniques are adversely affected because the knapsack problem is NP-hard. In this paper, we propose a new GPU-based approach to accelerate the multiple-choice knapsack problem, which is a general version of the 0-1 knapsack problem. Apart from exploiting the parallelism offered by the GPUs, we also employ a variety of GPU-specific optimizations to further accelerate the running times of the knapsack problem. Moreover, our technique is scalable in the sense that even when running large instances of the multiple-choice knapsack problems, we can efficiently utilize the GPU compute resources and memory bandwidth to achieve significant speedups.**

I. Introduction

Given a set of items, each associated with a profit and a weight, the 0-1 knapsack problem deals with how to choose a subset of items such that the profit is maximized and the total weight of the chosen items is less than the capacity of the knapsack. There exists several variants and extensions of this knapsack problem. In this paper, we focus on the multiple-choice knapsack (MCK) problem because it is a general version of the 0-1 knapsack problem. The MCK problem deals with the case where the items are grouped into disjoint classes and it will be formally introduced in Section II.

The knapsack problem manifests itself in many domains like cryptography, financial domain, bioinformatics as well as electronic design automation. Several algorithms and tools in the domain of system-level design and analysis techniques are variants of the knapsack problem. For instance, it has been shown that a voltage assignment problem and a code-size reduction problem on multi-processor platform are variants of the 0-1 knapsack problem [8] and the MCK problem [2], respectively. The knapsack problem is known to be NP-hard and hence the running times of large instances of the problem are significantly high. Moreover, the knapsack problem is found at the core of optimization loops in system-level problems [6] which also leads to high running times. While heuristics [7], approximation schemes [7] and meta-heuristics [1] have been proposed to solve the problem in polynomial-time, such techniques do not return the optimal solution. A dynamic programming algorithm to solve the problem optimally is known [7], but it has a pseudo-polynomial running time.

Our contributions and related work: To mitigate the high running times for solving the knapsack problem, in this paper, we propose a GPU-based technique. In particular, we map the dynamic programming algorithm on the GPU and show that significant speedups can be achieved in terms of the running times.

Recently, a thread of research work has focused on mapping various dynamic programming algorithms on to GPUs. A dynamic programming algorithm builds a table via a number of iterations, where each iteration fills one row of the table. Due to data dependencies across the iterations, results from the previous iterations must be ready before proceeding to the next iteration. This calls for a synchronization between two iterations. Typically, CPU-based synchronization approaches have been used for implementations of dynamic programming algorithms on the GPU and this hampers the potential speedups. Recent techniques have proposed a GPU-based synchronization technique in order to avoid CPU synchronization [13], [12]. However, such methods put a limit on the total of number of threads that can run on each GPU multi-processor concurrently and this can limit the scalability. In contrast, our proposed technique is scalable even with a GPU-based synchronization technique. Moreover, we propose a multi-phased synchronization scheme that significantly reduces the number of threads that are otherwise busy waiting.

We would like to mention that GPUs have been deployed to accelerate system-level design problems based on knapsack problems [3]. Recently, Boyer et al. [4] reported some results of solving the knapsack problem on GPUs. However, our scheme has several advantages over both approaches. First, they have not leveraged GPU-based synchronization. Second, unlike above techniques, we configure the thread block size in synergy with methods to exploit with on-chip memory. Moreover, the papers above did not address the scalability issue. Finally, we compare our GPU results with implementations on powerful multi-core platforms. We also note that our efforts are orthogonal to work on using FPGAs for dynamic programming and knapsack problems [9].

II. The Multiple-Choice Knapsack problem

In this section, we formally introduce the MCK problem. Given m classes of items, with each class consisting of n_i items, each item in class i associated with a value of v_{ik} and a weight of w_{ik}, and a knapsack of capacity C, the MCK

978-3-9810801-8-6/DATE12/2012 EDAA

problem is to choose exactly one item from each class such that we maximize the value (profit), while having the total weight of the chosen items at most equal to C.

We now succinctly describe a well-known dynamic programming algorithm [7] that solves the multiple-choice knapsack problem in pseudo-polynomial time [7]. For a problem of capacity C with m classes, the dynamic programming algorithm builds a table with $m+1$ rows and $C+1$ columns.The algorithm iterates m times corresponding to the m items, and computes the values of all $C+1$ cells of one row of the table in each iteration.At each cell j in the ith row, the equation $z_i(j) \leftarrow max(z_i(j), z_{i-1}(j-w_{ik}) + v_{ik})$ computes the profit $z_i(j)$ based on the weights w_{ik} and the values v_{ik}). The time complexity of the algorithm is $O(m \cdot n \cdot C)$, where $n = \sum_{i=1}^{m} n_i$.

III. Proposed Approach

In this section, we present our proposed approach. This is a short paper and for the sake of brevity, we will assume that the reader is familiar with the CUDA programming platform. For a complete description of CUDA, we refer the reader to NVIDIA's guide [10]. Our approach has four major components and they will be described in the following.

A. Identifying data parallelism

To accelerate the dynamic programming algorithm using GPUs, it is important to identify the dependencies and determine the data-parallel portions of the algorithm. From Section II, we see that computation of any cell in a row (z_i) depends on the weights and values of the items in that class and values in the previous row of the table (z_{i-1}). Thus, there exists no dependency along the row being computed and we can compute a single row concurrently via separate threads. With each thread computing a cell in a row, we launch $C+1$ threads to compute a row concurrently. Note that the threads must not start computation of the values of row $i+1$ before all the cells in the row i have been computed. To ensure this, a straightforward approach is to perform synchronization on the host CPU that enforces all threads of row i to run to completion and then, launch the GPU kernel once again for the next iteration $i+1$. However, this implies that with a problem instance having m classes of items, we must launch the kernel m times to compute the entire table which leads to performance deterioration [13]. We overcome this situation by using GPU-based global synchronization as discussed in Section III-C.

B. Shared memory and thread block size

During the computation of the ith row, each thread must fetch the required input values and weights, i.e., the set $\{(v_{i,1}, w_{i,1}), \ldots, (v_{i,n_i}, w_{i,n_i})\}$, corresponding to the ith class in the given problem instance. Each time a thread fetches these data from the *Global Memory*, there is an additional latency (see [10]). In order to hide such penalties during memory accesses, we first note that all threads require the same set ($\{(v_{i,1}, w_{i,1}), \ldots, (v_{i,n_i}, w_{i,n_i})\}$) of data. Hence, we pre-fetch the entire set into the on-chip *Shared Memory* before the computation by the threads start.

On-chip *Shared Memory* is shared only between the threads of the same thread block [10]. Hence, pre-fetching data into shared memory hides latencies for threads within a thread block but each thread block must fetch the data at least once. Hence, larger thread blocks (i.e., less thread blocks overall) imply that more threads share the on-chip *Shared Memory* and the number of times that data is fetched from the *Global Memory* is less, thereby improving the performance. Thus, it appears as if choosing the largest thread block size allowed by CUDA hardware would be the optimum choice from the perspective of performance.

Thread block size: We choose the number of thread blocks launched (B') and the number of threads in each thread block (T), such that the total number of active threads results in maximum utilization. We illustrate our choice in the context of nVIDIA Tesla M2050 but note that the same principle can be applied to other devices. nVIDIA Tesla M2050 has 14 multi-processors and allows a maximum of 1536 active threads per multi-processor. We choose 768 threads per thread block and a total of 28 thread blocks. Such a configuration assigns two thread blocks or equally, 1536 threads per multi-processor leading to a 100% utilization or occupation, as encouraged by CUDA [10]. Hence, in this set-up, $B' = 28$ and $T = 768$. With a constant number of thread blocks irrespective of the problem size, we ensure that *Global Memory* accesses remains limited even for large problem sizes while ensuring high utilization.

Typically, such a limit the number of thread blocks, and hence the total number of threads, imposes a limit on the problem size CUDA can tackle. As an illustration, let us assume that the thread block size is T. Considering a knapsack of capacity C, we know the size of the row in the dynamic programming table is $C+1$. If each thread computes one cell of the table, we need $ceil(\frac{C+1}{T})$ thread blocks to compute one row. For problem instances where C is large, $ceil(\frac{C+1}{T}) > B'$. Hence, to ensure scalability, we compute multiple cells with single thread and this is described in Section III-D.

C. GPU-based synchronization

Recall from Section III-A that we need to synchronize the threads between two kernel launches. We utilize a GPU-based synchronization approach instead where the kernel is launched only once to compute the complete table instead of m times as required by CPU-based synchronization.

We use inter-block communication via *Global Memory* to achieve synchronization. Each thread updates a variable in *Global Memory* upon completing its computation. A small number of threads of size S, where S is the size of the warp, in each thread block are kept idling in an infinite loop until this variable in the *Global Memory* has been updated by all the other thread blocks as well. A warp in CUDA is essentially the smallest set of threads that are scheduled in parallel by CUDA. While S threads are idling for global synchronization, the rest of threads in the thread block can proceed to pre-fetch data

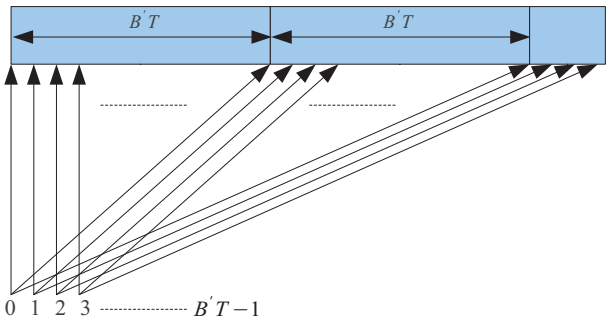

Fig. 1. Multiple cells computed by each thread in coalesced fashion. Computation of cells in a single row is divided into several iterations, and in each iteration the threads from 0 to $B'T-1$ compute adjacent cells.

for the next iteration thereby reducing the bottleneck involved in global synchronization. We note here that according to CUDA schedulers, the active thread blocks do not yield the execution. Thus, if S threads of a thread block is waiting on a global variable, not only this thread block but all other non-active thread blocks will be deadlocked as well. However, as described in the previous section, we configure the number of thread blocks on each multi-processor such that there are only active thread blocks in our kernels. This does not limit us from solving large instances of the problem, though, because we utilize each thread to compute multiple cells of the dynamic programming table. This will be described in the next section.

D. Single Thread Mutli-cell Computation

We have discussed in Section III-B and Section III-C, that we configure our thread block size on each multi-processor to a small number with limited number of threads. This is in order to ensure (i) high processor utilization and optimize memory accesses and (ii) facilitate GPU-based synchronization. Even with this restricted thread block size, we are able to solve large problem instances by employing each thread to compute multiple cells in the dynamic programming table. Our approach is described below.

Considering B' thread blocks and each block having T threads, we have a total of $B'T$ threads. For simplicity of elucidation, let us consider a problem instance where each thread computes exactly L cells, $L = \frac{C+1}{B'T}$. An intuitive approach is employ each thread to compute L adjacent cells of the table. For instance, thread-0 computes $0 \ldots L-1$, thread-1 computes $L \ldots 2L-1$ and so on. In this approach, computation of a row of the dynamic programming table is done in L iterations. During the 1^{st} iteration, thread-0 computes cell 0, thread-1 computes cell L, thread-2 computes cell $2L$ and so on, in parallel to exploit the parallelism Section III-A. During the 2^{nd} iteration, thread-0 computes cell 1, thread-1 computes cell $L+1$, thread-2 computes cell $2L+1$ and so on. Note, however, that in this approach, threads with consecutive *id*s must access non-adjacent memory locations and this results in a non-coalesced access pattern [10]. This severely reduces the global memory throughput and impedes performance. We propose an approach to maintain high global memory throughput by coalescing global memory access. This is illustrated in Figure 1. We see that any thread, X, is responsible for the cells X in the 1^{st} iteration, $X + B'T$ in the 2^{nd} iteration and so on until all the $C+1$ cells in the row are computed. This means that in each iteration threads with adjacent ids will compute adjacent cells and accessing adjacent memory locations leading to coalesced memory access.

Set Number	**Number of classes (m)**	**Capacity (C)**	**Problem size ($m \times C$)**
Set 1	5	12665	63325
Set 2	10	15700	157000
Set 3	20	94280	1885600
Set 4	50	390500	19525000
Set 5	100	303500	30350000

TABLE I
PROBLEM INSTANCES

IV. EXPERIMENTS AND RESULTS

Experimental setup: Our proposed GPU-based implementation, henceforth referred to as the CUDA-SYNC, was executed on a nVIDIA Tesla M2050 GPU, which consists of 14 multi-processors and a total of 448 processing elements running at 1147 MHz. The GPU was connected to host system via on-board PCI-express (16x) slot. The host machines consisted of 2 Xeon E5520 CPUs, where each CPU has 4 cores — a total of 8 cores. Apart from CUDA-SYNC, we also implemented three other versions on the CUDA in order to illustrate the improvements achieved from our proposed techniques. First, we have CUDA-GLOBAL where we have not implemented any pre-fetching to on-chip *Shared Memory*, GPU synchronization or multi-cell computation techniques. In this regard, CUDA-GLOBAL is similar to the existing techniques [3], [4]. Second, we have implemented CUDA-SHARED where we allow *Shared Memory* usage but neither GPU synchronization or multi-cell computation has been included. Finally, we have also implemented CUDA-MULTI cell that allows multiple cells to be computed by the same thread but do not include GPU-based synchronization. We compare the running times of each of these implementations with CUDA-SYNC and show that CUDA-SYNC outperforms all of them.

Apart from the GPU implementations, an OpenCL (spec v1.1) implementation of the dynamic programming algorithm that was executed on the host to compare how an implementation on multi-core platform performs against a GPU. In this implementation, we computed the rows of the dynamic programming algorithms in parallel in the 8 cores. All the cores of the host were clocked at 2.27 GHz. We also had a sequential version (CPU) of the algorithm that ran on a single core of the host CPU.

To evaluate the performance of the above implementations, random problem instances of 5 different sizes were generated. Table I shows the five different problem sets that were generated for the experiments. In each set, we generated 5 instances and the running times were averaged over these 5 instances. The values and weights in the problem instances were randomly generated, with no correlation between the value and weight of an item [11], [5]. In all the instances, the number of choices in each class (n_i) was between 10 and

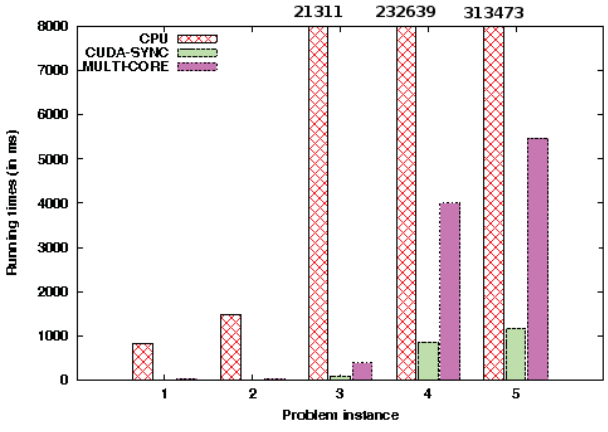

Fig. 2. Comparison of our proposed CUDA-SYNC algorithm against sequential CPU and 8-core implementations.

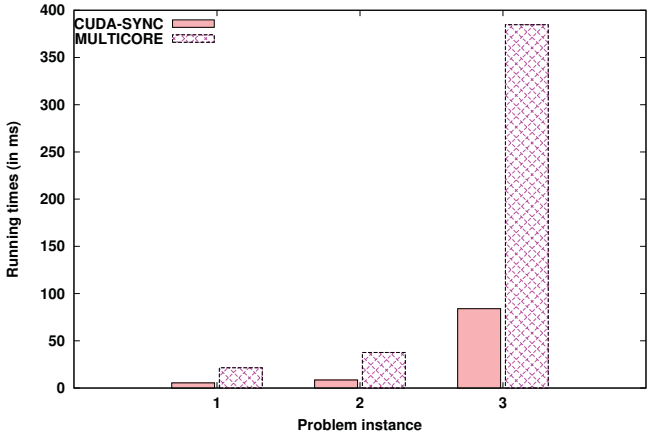

Fig. 3. Comparison of our proposed CUDA-SYNC algorithm against multi-core implementation.

1024. The values and weights (w_{ik}) were assigned random values between 1 and 10000.

Results: We now discuss the results from the experiments conducted. In case of GPU-based implementations, the running times reported here also includes the time required to transfer the input data to the GPU and the time required to fetch the results from the GPU to the host.

Figure 2 shows the comparison between the running times of the sequential algorithm (CPU), our proposed CUDA-SYNC algorithm and the multi-core (8-core) implementation. For smaller problem instances (1 and 2), the running times of the GPU and multicore implementations are not visible, since they coincide with the x-axis. CUDA-SYNC achieves a tremendous speedup of 220 $\times$ over the sequential CPU implementation. Note that the bars for CPU running times for problem instance 3, 4 and 5 are truncated so as to fit them into the figure and the running times are labeled at the top. Notice that the speedup is higher for larger problem instances.

In Figure 3, we plot the running times of the multi-core implementation with CUDA for problem sets 1 to 3 because they were not visible in Figure 2. We report that our CUDA implementation achieves a speedup of 3-4 $\times$ on average over multi-core implementation. This is despite the fact that all the eight cores of our multi-core platform run at significantly higher frequency than the GPU processing elements. This

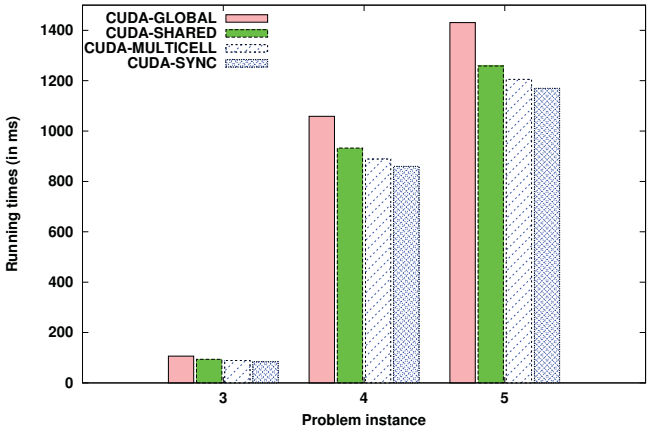

Fig. 4. Comparison of our proposed CUDA-SYNC algorithm against other CUDA-based implementations.

is due to the significant parallelism that our implementation can extract from the dynamic programming algorithm as well as the fact that we have been able to implement various optimizations like GPU-based synchronization.

In Figure 4, we compare CUDA-SYNC against all the other CUDA implementations.For clarity, we have plotted results only for the larger problem instances. CUDA-SYNC outperforms all the other GPU implementations. This shows the impact of each feature in our framework that carefully exploits the dynamic programming characteristics in synergy with CUDA architectural features. CUDA-SYNC, for example, is 20% faster than CUDA-GLOBAL (which is similar to existing techniques [3], [4]) on an average.

REFERENCES

[1] N. K. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler. Systematic integration of parameterized local search into evolutionary algorithms. *IEEE Trans. Evolutionary Computation*, 8(2), 2004.

[2] S. Baruah and N. Fisher. Code-size minimization in multiprocessor real-time systems. In *International Parallel and Distributed Processing Symposium*, 2005.

[3] U.D. Bordoloi and S. Chakraborty. Accelerating system-level design tasks using commodity graphics hardware: A case study. In *International Conference on VLSI Design*, 2009.

[4] V. Boyer, D. El Baz, and M. Elkihel. Solving knapsack problems on GPU. *Comput. Oper. Res.*, 39:42–47, January 2012.

[5] B. Han, J. Leblet, and G. Simon. Hard multidimensional multiple choice knapsack problems, an empirical study. *Comput. Oper. Res.*, 37:172–181, 2010.

[6] H. P. Huynh and T. Mitra. Instruction-set customization for real-time systems. In *DATE*, 2007.

[7] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004.

[8] H. Liu, Z. Shao, M. Wang, J. Du, C. J. Xue, and Z. Jia. Combining coarse-grained software pipelining with DVS for scheduling real-time periodic dependent tasks on multi-core embedded systems. *J. Signal Process. Syst.*, 57, 2009.

[9] Z. Nawaz, T. P. Stefanov, and K.L.M. Bertels. Efficient hardware generation for dynamic programming problems. In *International Conference on Field-Programmable Technology*, 2009.

[10] NVIDIA. CUDA Programming Guide version 4.0, 2011.

[11] D. Pisinger. Core problems in knapsack algorithms. *Oper. Res.*, 47:570–575, 1999.

[12] S. Xiao, A. M. Aji, and W. Feng. On the robust mapping of dynamic programming onto a graphics processing unit. In *International Conference on Parallel and Distributed Systems*, 2009.

[13] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In *International Conference on Parallel and Distributed Systems*, 2010.