Knapsack Problem on Xeon Phi

Martin Hořeňovský
Thursday, 12:45 - 14:15
Otevřená Informatika
horenmar@fel.cvut.cz

Abstract

This document describes work on developing a parallel algorithm for solving the Knapsack Problem on Xeon Phi accelerator.

I. Assignment

A. Problem statement

Knapsack problem (KP) is a well known problem in combinatorial optimization. It can be applied to problems such as material cutting with minimal waste and can be a subproblem of more complex problems.

KP takes three inputs

- C - the capacity of a knapsack
- **p** - vector of item profits
- **w** - vector of item weights

where $p_i \in \mathbb{N}_+$, $w_i \in \mathbb{N}_+$ and $C \in \mathbb{N}_+$. p_i, w_i is the profit from ith item and weight of ith item respectively.

We can also assume that $\sum_{i=1}^{n} w_i > C$ as otherwise the problem's solution would be trivial.

Given these inputs, ILP can be formulated in the form of

$$\begin{aligned} max & \sum_{i=1}^{n} p_i x_i \\ s.t. & \sum_{i=1}^{n} w_i x_i \leq C, \\ & x_i \in \{0,1\}, \\ & i \in \{1,..,n\} \end{aligned} \tag{1}$$

and the solution of this ILP is price-optimal packing of knapsack with capacity C.

The goal of this assignment is to find a way to speed up this computation using a Xeon Phi coprocessor and be able to solve large problem instances within less time than using current CPU.

B. Problem categorization

KP is known to be NP-hard, with a pseudopolynomial solution based on Dynamic Programming (DP). The complexity of DP-based solution is $\mathcal{O}(nC)$, where n is the number of considered items to pack and C is the knapsack capacity.

Architecturally, the Xeon Phi coprocessor is a departure from classic x86 CPU towards GPGPU, as it contains relatively high (~ 50) number of x86 cores with large vector units, where a GPGPU usually contains thousands of cores, but the cores do not have vector units and have much less single threaded performance.

There has been previous work on paralelizing KP using GPGPU computation[1], which has achieved a speed up of ~ 26 times and in this task I will attempt to apply the same approach for parallelizing KP to the Xeon Phi.

II. Related works

There is a large body of work done on the Knapsack Problem, including parallel DP computation on a GPGPU by Boyer et al.[1], parallel computation on a general purpose CPU by Loots et al.[2] and DP computation on a massivelly parallel CPU by Chang et al.[3].

- Boyer et al.'s paper shows a data compression technique used to reduce memory occupancy caused by the DP calculation, decreasing significantly the bandwidth needed for communication between a host execution unit and a coprocessor.
- Loots et al.'s paper shows a different approach for parallel computation of knapsack problem, exploiting parallelism during backtracking steps of branch-and-bound algorithm, allowing for nearly linear speed-up over small number of parallel execution units.

- Chang et al.'s paper shows approach to solving Knapsack problem trading large amount of execution cores for total execution complexity. For a knapsack of n components, Chang shows an algorithm solving the problem in $\mathcal{O}(2^{n/2})$, given $\mathcal{O}(2^{n/8})$ execution units.

There is also a large body of work on unbounded KP, giving strong heuristics for reducing the search space, such as the threshold dominance[4], but these results do not affect the bounded knapsack problem.

III. Problem Solution

I decided to use pseudo-polynomial Bellman's[5] algorithm based on dynamic programming solution for the Knapsack problem, as seen in equation 2. After testing several approaches to parallelization of the DP solution, I ended adapting algorithm used by Boyer et al[1] for GPGPU, as it proved to be significantly faster than both of my original approaches.

$$f(k,c) = \begin{cases} 0 & k = 0 \\ f(k-1,c) & c < w_k \\ f(k-1,c) & f(k-1,c) > f(k-1,c-w_k) + p_k \\ f(k-1,c-w_k) + p_k & f(k-1,c) < f(k-1,c-w_k) + p_k \end{cases} \tag{2}$$

A. Final Implementation

In my final implementation, I do not materialize the full DP table, materializing instead only two rows and use table entry level parallelism, that is when a row is being filled, the work is split between all threads. When a row is filled, the two rows are swapped and the previously filled row is used as an input for next iteration of the computation. In each iteration, I also improve the starting point by using Toth's inequality[6]. Toth's inequality says that while checking kth item, all weights for which the following inequality holds, will not lead to optimal solution.

$$w_i < C - \sum_{i=k+1}^{n} w_i$$

I constructed three versions of this algorithm, one using only a single thread, one running on the CPU across all cores and one specialized to run on the Xeon Phi. The Xeon Phi version forces use of aligned memory, as the Xeon Phi cores have a significant performance penalty when loading from memory not aligned on 64 bytes. The CPU version can optionally force use of aligned memory, but it did not measurably improve its performance during testing.

Both parallel version were made using OpenMP parallel loop pragma. An attempt was also made to force vectorization, but it did not speed up the resulting binary. This is likely because any vectorization attempt would have to deal with loading unaligned data pairs with changing offsets and thus not end up being faster than naive loop.

The algorithm also allocates a decision table, where it stores a bit-packed decision record (that is, if for item i and weight w it seems better to use the item than not to use it, d_{iw} will be set to true).

B. Failed Implementations

Before settling on the final implementation, I benchmarked two different approaches to paralelization of filling the DP table for knapsack. Both were based on the fact, that to fill in an entry in the DP table, we only have to know two previous entries, both of which are in the row above the desired entry.

1) Parallel diagonal filling: My first approach was based on filling in the table in diagonal order, with each thread getting its own diagonal (and taking another once it is finished). This approach was meant to weakly enforce ordering of access to the table and thus minimize the amount of waiting each thread has to do, before it can access entries needed to calculate its own entries. Because of bad cache affinity and high requirements for synchronization (every entry in the table had an atomic flag on whether it is ready) of this approach, it has been the worst performing approach, being worse than single threaded naive approach by more than an order of magnitude.

2) Parallel row filling: My second approach was based on filling in the table in row-by-row order, with each thread getting its own row. This has lead to better cache affinity and the synchronization was changed to being an integer per row, indicating how far has given row been computed. This approach was an improvement over the first one, being faster by $\sim 20\%$. However, it was still slower than the naive approach using a single thread.

IV. Experiments

A. Benchmark settings

I performed my tests on a Department's of Control Engineering (DCE) server, comprising of

- Two Intel Xeon E5-2620 v2 CPUs (for a total of 24 cores at 2.1 GHz)
- 64GB of RAM

- Two Xeon Phi accelerators
- Gentoo Linux operating system
- Intel C++ Compiler in version 16.0.0

For the purpose of the experiments, only one of the Xeon Phi accelerators was used. This is because the DP solution for Knapsack problem requires shared memory between all computing cores and this is impossible to do with two Xeon Phi coprocessors.

I randomly generated problems of various sizes, run each problem 10 times and took the median time required to get correct result. The "Basic" column represents basic DP-based approach with full profit and decision tables and single thread. The "ST" column represents the final implementation, using only single thread, the "MT" column represents the final implementation running on all CPU cores and the "Phi" column represents results of the same algorithm, but run on the Xeon Phi coprocessor. Results can be found in table

B. Results

Table I
SOLVER TIMES ON PROBLEMS OF DIFFERENT SIZES

Number of items	Knapsack capacity	Solver run time (ms)			
		Basic	ST	MT	Phi
10	22	0.0	0.0	8.0	7.0
100	223	0.0	0.0	2.0	17.0
200	398	0.0	0.0	16.0	29.5
300	643	0.0	0.0	20.0	37.0
400	883	1.0	0.0	25.0	50.0
500	1041	1.0	0.0	16.5	56.5
1000	2104	7.0	1.0	5.0	108.0
2000	4237	44.0	6.0	10.0	201.5
5000	10549	288.0	41.0	26.0	499.0
10000	20978	1131.5	167.0	63.0	1012.5
20000	41946	4399.5	733.0	172.5	2126.5
30000	62565	9787.0	1630.0	320.0	3275.5
40000	84261	17511.0	2890.5	566.0	4711.5
50000	105006	27142.5	4438.5	821.5	5618.5
60000	126151	39333.0	6371.0	1174.5	7094.0
70000	147108	53362.5	8726.0	1494.5	8325.5
80000	167230	69756.5	11327.0	2032.5	10210.5
90000	188730	—	14362.5	2351.0	11790.5
100000	209188	—	17696.5	3027.5	13435.0
110000	231339	—	21508.5	3549.5	14487.0
125000	262182	—	27933.5	4524.5	17209.5
150000	315544	—	41394.0	6371.5	21324.5
200000	419649	—	73667.0	11193.5	—
250000	525054	—	115488.5	17379.0	—
300000	630977	—	166739.0	24911.0	—
350000	735416	—	223512.0	33610.0	—
400000	839697	—	305762.0	45900.0	—

The results tableI starts at 10 item to try and illustrate the rough start-up times of the different solvers, and stops at 400k items, because that is approximately the limit of RAM on the kepler server under usual idle load. It would be possible to improve the limit by writing partial decision table to the disk, but doing so seemed out of scope of evaluating the Xeon Phi coprocessor.

Empty table entries represent problem sizes for which given solver was unable to finish.

1) "Basic" (naive) solver: For problems containing more than $\sim$ 80k items, the memory requirements of "Basic" solver were too large to run. Not surprisingly it was also the worst scaling solver of the group, although for small problems it was able to run faster than the parallel solver, as the startup time was significant for small problems.

2) ST solver: The single threaded (ST) version of the parallel solver trivially outperforms the naive solver for all input instances. Unlike the naive and Phi solvers, it also scales to large inputs, even if its runtime becomes significant (the largest instance that the test server was able to contain in its RAM has run for longer than 5 minutes).

3) MT solver: The CPU-multithreaded (MT) version of the parallel solver has shown the best results during the testing, being able to scale as well as the ST solver, but being almost 8x faster while utilizing the 24 cores on the server's CPU. However, its start-up time means that for problems under $\sim$ 5000 items, it is better to use the ST solver.

4) Phi Solver: The Xeon Phi coprocessor (Phi) version of the parallel solver has shown significant start-up times, being able to outperform the ST solver only for sufficiently large problems (over $\sim$ 70k items), but because the coprocessor has fixed amount of RAM it can use - 8 GBs - it stopped being able to scale at $\sim 150k$ items, where attempt to offload the required data to Xeon Phi caused program crash with out-of-memory message.

However, in problem sizes ranges where it is able to run and the start-up time is not significant anymore ($50000-150000$ items), it seems to scale better than the MT solver.

C. Discussion

Because I have not tried to perform manual vectorization after autovectorization failed to speed up the resulting binary, there might be further speed up possible for the Xeon Phi implementation. However, the CPU implementation it was benchmarked against could have also been manually vectorized, assuming a Xeon CPU from newer generation. The Xeon CPUs in the test server did not benefit from autovectorization, as they are lacking the necessary AVX2[7] vector instructions.

Another possible performance improvement for the Xeon Phi would be to have the whole work done on the coprocessor, instead of the work only being offloaded there. This would decrease the memory traffic between host and coprocessor significantly, but it would also limit it to even smaller problem instances, as this would mean that the Xeon Phi would have to allocate and fit full decision table in its own memory. Since Xeon Phi has only $\sim$ 7.5 GB of memory left after system overhead and item transfer, it would impose much strong limit on the problem scale it can solve, as the two DP table rows allocated for the 150k items large problem, need $\sim$ 1.2 GB of memory each. If the full decision table had to be allocated on the Xeon Phi, the largest problem it would be able to solve is at around 50k items. In contrast, the current approach allows it to tackle any problem whose decision table fits into the host memory. For the test server, this is almost an order of magnitude difference, and getting more RAM for CPU is usually easy, where Xeon Phi has a set amount of RAM and cannot be upgraded.

V. Conclusion

I attemped to develop a high performance algorithm for solving the Knapsack Problem on Xeon Phi accelerator and measured the results against a naive single threaded implementation and the same algorithm but running on the CPU, using either one or all CPU cores.

The result of this work is multithreaded knapsack solver, that reaches speedups of $\sim$ 8 times on CPU for large knapsack problems, and Xeon Phi implementation that outperforms single threaded solver and shows better scaling than the MT knapsack solver, but runs out of memory for problems that are too small to let it catch up.

Limited testing suggests, that if the decision table requirement was removed and the Xeon Phi implementation only had to answer what is the optimal value of a knapsack problem, instead of answering the composition of optimally filled knapsack, it could scale further and nearly catch up with the MT solver on CPU, possibly outperforming a weaker CPU or leaving it capable of working on different problem.

Also, since the data transfer between host and the coprocessor puts large performance tax on the Xeon Phi, it might be worthwhile to investigate various compression data compression techniques to decrease the amount of data that have to be transfered and possible implement continous streaming of decision table data to further increase the Xeon Phi's performance.

Vectorization is unlikely to result in significant improvement in the Xeon Phi situation for two reasons. One, it is lacking a load-unaligned vector instruction, which is important for performance in this specific problem, as the DP algorithm will always have to access to unaligned data. It only has a gather vector instruction, which caries a performance penalty compared to load-unaligned. Furthermore, the CPU-based MT solver could also be vectorized, assuming a newer Xeon with AVX2 instructions, giving the same performance increase to the CPU implementation.

In conclusion, I do not think that the Xeon Phi is well suited to speeding-up the knapsack problem.

References

[1] V. Boyer, D. E. Baz, and M. Elkihel, "Solving knapsack problems on gpu," *Computers and Operations Research*, vol. 39, pp. 42–47, 2012.

[2] W. Loots and T. H. C. Smith, "A parallel algorithm for the 0–1 knapsack problem," *International Journal of Parallel Programming*, vol. 21, pp. 349–362, 1992.

[3] H. K.-C. Chang, J. J.-R. Chen, and S.-J. Shyu, "A parallel algorithm for the knapsack problem using a generation and searching technique," *Parallel Computing*, vol. 20, pp. 233–243, 1994.

[4] W. Andonova, V. Poirrieza, and S. Rajopadhyeb, "Unbounded knapsack problem: Dynamic programming revisited," *European Journal of Operational Research*, vol. 123, pp. 394–407, 2000.

[5] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[6] P. Toth, "Dynamic programming algorithms for the zero-one knapsack problem," *Computing*, vol. 25, no. 1, pp. 29–45, 1980. [Online]. Available: http://dx.doi.org/10.1007/BF02243880

[7] "Ark page for intel xeon e5-2620 v2," Intel ARK. [Online]. Available: http://ark.intel.com/products/75789/Intel-Xeon-Processor-E5-2620-v2-15M-Cache-2_10-GHz