Solving knapsack problems on GPU

Vincent Boyer, Didier El Baz, Moussa Elkihel

Solving knapsack problems on GPU

V. Boyer, D. El Baz, M. Elkihel

CNRS; LAAS; 7 avenue du Colonel Roche, F-31077 Toulouse, France
Université de Toulouse; UPS; INSA; INP; ISAE; LAAS; F-31077 Toulouse, France

Abstract

In this article, we propose a parallel implementation of the dynamic programming method for the knapsack problem on NVIDIA GPU. A GTX 260 (192 cores, 1.4GHz) was used for computational tests and processing times obtained with the parallel code are compared to the sequential one on a CPU with an Intel Xeon 3.0GHz. The results show a speedup up to 26 and permit one to solve large size problems within a reasonable processing time. Furthermore, in order to limit the communication between the CPU and the GPU, a compression technique is presented which decreases significantly the memory occupancy.

Keywords: Knapsack problems, Dense dynamic programming, Parallel computing, GPU computing.

1. Introduction

Since a few years, graphics card manufacturers have developed tools to use their products for high performance computing. Graphics Processing Units, or GPUs, are high-performance many-core processors. NVIDIA GPUs are SIMT (Single Instruction, Multiple Thread) architectures which is akin to SIMD (Single Instruction, Multiple Data) architecture [1].
Severals parallel dynamic programming method have been proposed (see, for example, [2], [3] and [4]). Implementations on SIMD machine were performed on a 4K processor ICL DAP [5] and 16K Connection Machine CM-2 systems (see [6], [7]) and a 4K MasPar MP-1 machine [7]. To the best of our knowl-

Email address: {vboyer, elbaz, elkihel}@laas.fr (V. Boyer, D. El Baz, M. Elkihel)

edge, no parallel implementation of dynamic programming for combinatorial optimization problems has been done on a GPU.
Using GPU architectures for solving combinatorial optimization problems is a great challenge in order to reduce the computing time needed to solve these NP-hard problems. In this paper, we propose a parallel implementation of the dynamic programming algorithm on a NVIDIA GPU for exactly solving the KP. Furthermore, a data compression is presented. This compression permits one to limit significantly, the communication between the CPU and the GPU and the memory needed to build the solution vector.
The paper is structured as follows. Section 2 deals with the knapsack problem and its solution via dynamic programming. Section 3 focuses on GPU computing and the parallel dynamic programming method. In section 4 we propose a compression method which permits one to reduce significantly the memory occupancy and communication between CPU and GPU. Section 5 deals with computational experiences. Conclusion and future work are presented in section 6.

2. The knapsack problem

The knapsack problem, KP, is a NP-hard combinatorial optimization problem. It is one of the most studied discrete programming problem as it is among the simplest prototypes of integer linear programming problems and it arises in several sub-problems of many more complex problems (see, for example, [8], [9], [10], [11], [12], [13] and [14]).

2.1. Problem formulation

Given a set of n items i, with profit $p_i \in \mathbb{N}_+^*$ and weight $w_i \in \mathbb{N}_+^*$, and a knapsack with the capacity $C \in \mathbb{N}_+^*$, KP can be defined as the following linear integer programming problem:

$$(KP) \begin{cases} \max \sum_{i=1}^{n} p_i.x_i, \\ s.t. \sum_{i=1}^{n} w_i.x_i \leq C, \\ \quad x_i \in \{0,1\},\ i \in \{1,...,n\}. \end{cases} \tag{2.1}$$

To avoid any trivial solution, we assume that:

$$\begin{cases} \forall i \in \{1, ..., n\},\ w_i \leq C, \\ \sum_{i=1}^{n} w_i > C. \end{cases}$$

2.2. Dynamic programming

Bellman's dynamic programming [15], presented in 1957, was the first exact algorithm to solve KP. It consists in computing at each step $k \in \{1, ..., n\}$, the values of $f_k(\hat{c})$, $\hat{c} \in \{0, ..., C\}$, using the classical recursion:

$$f_k(\hat{c}) = \begin{cases} f_{k-1}(\hat{c}),\ for\ \hat{c} = 0, ..., w_k - 1, \\ \max\left\{f_{k-1}(\hat{c}),\ f_{k-1}(\hat{c} - w_k) + p_k\right\}, for\ \hat{c} = w_k, ..., C. \end{cases} \tag{2.2}$$

with, $f_0(\hat{c}) = 0,\ \hat{c} \in \{0, ..., C\}$.
The algorithm, presented in this section, is based on the the Bellman's recursion (2.2). A state corresponds to a feasible solution associated with the $f_k(\hat{c})$ value. Toth [16] has proposed an efficient recursive procedure in order to compute the states of a stage and used the following rule to eliminate states:

Proposition 1 *[16] If a state defined at $k - th$ stage with total weight $\hat{c}$ satisfies:*

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

then the state will never lead to an optimal solution and can be eliminated.

The dynamic programming method is described in algorithm 1. The matrice M stores all the decisions and is used to build a solution vector corresponding to the optimal value by doing a backtracking. The time and space complexities are $\mathcal{O}(n.C)$.

Algorithm 1 (Dynamic programming)
for $\hat{c} \in \{0, ..., C\},\ f(\hat{c}) := 0$,
for $i \in \{1, ..., n\}$ *and* $\hat{c} \in \{1, ..., C\},\ M_{i,\hat{c}} = 0$,
$sumW := \sum_{i=1}^{n} w_i$,

for k from 1 to n do
    $sumW := sumW - w_k$,
    $\underline{c} = max\{C - sumW, w_k\}$,
    for $\hat{c}$ *from* C *to* $\underline{c}$ *do*
        if $f(\hat{c}) < f(\hat{c} - w_k) + p_k$ *then*
            $f(\hat{c}) := f(\hat{c} - w_k) + p_k$,
            $M_{i,\hat{c}} := 1$,
        end if,
    end for,
end for.
return $f(C)$ *(the optimal bound of the KP)*

The high memory requirement are frequently cited as the main drawback of dynamic programming. However, this method has a pseudo-polynomial time complexity and is insensitive to the kind of instances, i.e. with correlated datas or not.
In order to reduce the memory occupancy, the entries of the matrix M have been stored in integers of 32 bits. This permits one to divide by 32 the number of lines of the matrix and the memory needed. However, the memory occupancy is still important and an efficient compression method will be presented in section 4.

3. GPU computing

GPUs are highly parallel, multithreaded, many-core architectures. NVIDIA introduced, in 2006, CUDA, a software development kit that enables users to solve many complex computational problems on their GPU cards. This tool has been used in order to implement our parallel dynamic programming code.

3.1. NVIDIA GPU architecture

On CUDA compatible NVIDIA cards, the threads are separated in blocks and these blocks are distributed on the multiprocessors. A multiprocessor processes one block at a time. When the threads of a block terminate, a new block is launched on the idle multiprocessor. The multiprocessor executes threads in groups of 32 parallel threads called warps. Threads composing a warp start together at the same program address, they are nevertheless free to branch and execute independently.

Another important aspect in GPU computing is the memory hierarchy. Indeed, threads have access to data from multiple memory spaces. Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block which has the same lifetime as the block. Finally, all threads have access to a global memory. Furthermore, there is two other read-only memory spaces accessible by all threads which are cache memories:

- the constant memory, for constant data used by the process,
- the texture memory space, designed for graphics applications.

In order to have a maximum bandwidth for the global memory, we have to insure a coalesced memory accesses. Indeed, the global memory access by all threads is done in one or two transactions if:

- threads access:
  - either 32-bit words, resulting in one 64-byte memory transaction,
  - or 64-bit words, resulting in one 128-byte memory transaction,
  - or 128-bit words, resulting in two 128-byte memory transactions;
- all 16 words lie in the same segment of size equal to the memory transaction size (or twice the memory transaction size when accessing 128-bit words);
- threads access the words in sequence (the kth thread in the half-warp accesses the kth word).

Otherwise, a separate memory transaction is issued for each thread. For further details on the NVIDIA cards architecture and how to optimize the code, see [1].

3.2. Parallel dynamic programming

Parallel implementation of the dynamic programming method is optimized for GPU NVIDIA architectures. The activity that consumes processing time is the loop that processes the values of $f(\hat{c})$, $\hat{c} \in \{0, ..., C\}$. This step has been parallelized on the GPU: one thread processes one value of f. Many efforts have been made in order to limit the communication between the CPU and the GPU and ensure coalesced memory access in order to significantly reduce the processing time. The procedures implemented on the CPU and the GPU, respectively, are described in algorithms 2 and 3, respectively.

Algorithm 2 (CPU processing)
$n\_lines := \lceil n/32 \rceil$,
Variables stored on the device:
for $\hat{c} \in \{0, ..., C\}$ *do*
$f0\_d(\hat{c}) := 0$ *and* $f1\_d(\hat{c}) := 0$,
$m\_d_{\hat{c}} := 0$,
end for
Variables stored on the host:
for $i \in \{1, ..., n\_lines\}$ *and* $\hat{c} \in \{1, ..., C\}$, $M\_h_{i,\hat{c}} := 0$,
$sumW := \sum_{i=1}^{n} w_i$,
bit_count:=0 and k_M:=1,
for k from 1 to n do
$sumW := sumW - w_k$,
$\underline{c} := max\{C - sumW, w_k\}$,
bit_count:=bit_count+1,
if k is even then
Compute_f_and_m_on_device(f0_d,f1_d,m_d,$\underline{c}$*),*
else
Compute_f_and_m_on_device(f1_d,f0_d,m_d,$\underline{c}$*),*
end if
if bit_count=32 then
bit_count:=0,
copy m_d in $M\_h_{k\_M,\hat{c}}$,
for $\hat{c} \in \{0, ..., C\}$, $m\_d_{\hat{c}} := 0$,
k_M:=k_M+1,
end if
end for.
if n is even then
return $f1\_d(C)$,
else
return $f0\_d(C)$.

In algorithm 2, the launching of the threads on GPU is done via the following function:

Compute_f_and_m_on_device(input_f,output_f,output_m,c_min)

where:

- input_f are the values of f processed at the previous step,
- output_f are the output values of f,
- output_m are the output values of the decisions stored as integers of 32 bits and
- c_min is the minimum value of $\hat{c}$.

This function creates $C - c\_min + 1$ threads for the GPU and groups them into blocks of 512 threads (the maximum size of a block of one dimension), i.e. $\lceil (C - c\_min + 1)/512 \rceil$ blocks. All threads carry out on the GPU the procedure described in algorithm 3.

Algorithm 3 (Thread processing on GPU)
blocks_id: the ID of the belonging block,
thread_id: the ID of the thread within the belonging block,
k: the step number of the dynamic programming ($k \in \{1, ..., n\}$),
$i := (k + 31)\%32$*: the rest of the division of* $k + 31$ *by 32,*

$\hat{c}$:=*blocks_id * 512 + thread_id,*
if $\hat{c} < c\_min$ *or* $\hat{c} > C$ *then STOP end if,*
if $input\_f(\hat{c}) < input\_f(\hat{c} - w_k) + p_k$ *then*
    $output\_f(\hat{c}) := output\_f(\hat{c} - w_k) + p_k$,
    $output\_m_{\hat{c}} := output\_m_{\hat{c}} + 2^i$,
else
    $output\_f(\hat{c}) := output\_f(\hat{c})$,
end if

In the algorithm 3, threads have to access the values of $input\_f(\hat{c} - w_k)$, this results in un-coalesced memory accesses as described in section 3.1. In order to reduce the memory latency, the texture memory is used to access the data stored in $input\_f$. We used the texture memory since this type of memory can be allocated dynamicaly contrarily to the constant memory. $output\_f$ and $output\_m$ are stored in the global memory.

4. Reducing memory occupancy

The analysis of the values stored in the matrix M_h shows that the right columns are often filled with 1 and that the left columns are filled with 0. As

these bits values are grouped in integers of 32 bits, in practice it corresponds to the value $2^{32}-1$ for the right columns (and 0 for the left columns). Thus, the communication between the CPU and the GPU occurs every 32 iterations in order to retrieve all the decisions stored in m_d into the matrix M_h (see algorithm 2). This step is time consuming and we have tried to further reduce the amount of data transfered to the CPU.
A simple way to reduce the vector m_d is then to compress it as follows:

$$\begin{array}{l} for\ \hat{c} \in \{0, ..., rc - lc\},\ m\_d\_c_{\hat{c}} = m\_d_{\hat{c}+lc} \\ with \quad lc = min\{\hat{c} \in \{1, ..., C\} \mid m\_d_{\hat{c}-1} = 0\ and\ m\_d_{\hat{c}} \neq 0\}, \\ \qquad\quad rc = max\{\hat{c} \in \{1, ..., C\} \mid m\_d_{\hat{c}-1} \neq 2^{32}-1\ and\ m\_d_{\hat{c}} = 2^{32}-1\}. \end{array}$$

Thus, we know that:

- if $\hat{c} < lc$, then $m\_d_{\hat{c}} = 0$ and
- if $\hat{c} \geq rc$, then $m\_d_{\hat{c}} = 2^{32}-1$.

Then we have to retrieve only the values of $m\_d_{\hat{c}}$ for $\hat{c} \in \{lc, ..., rc-1\}$ and we process lc and rc directly on the GPU via the algorithm 4.

Algorithm 4 (Thread compression on GPU)
blocks_id: the ID of the belonging block,
thread_id: the ID of the thread within the belonging block,
m_d: the input vector,
lc: shared variable initiate with the value C,
rc: shared variable initiate with the value 0,

$\hat{c}$:=*blocks_id * 512 + thread_id,*
if $\hat{c} \leq 0$ *or* $\hat{c} > C$ *then STOP end if,*
if $m\_d_{\hat{c}-1} = 0$ *and* $m\_d_{\hat{c}} \neq 0$ *then*
$\quad lc := min\{\hat{c}, lc\}$,
end if,
if $m\_d_{\hat{c}-1} \neq 2^{32}-1$ *and* $m\_d_{\hat{c}} = 2^{32}-1$ *then*
$\quad rc := max\{\hat{c}, rc\}$,
end if.

This compression method decreases the amount of data transfered from the GPU to the CPU and permits one also to decrease significantly the memory occupancy needed to store all the decisions made throughout the dynamic

programming recursion. Computational experiences shows that the efficiency of the compression depends on the sorting of the variables of the KP and, in average, the best results have been obtained with the following sorting:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq ... \geq \frac{p_n}{w_n}.$$

5. Computational experiences

Computational tests have been carried for randomly generated correlated problems, i.e. problems such that:

- w_i, $i \in \{1, ..., n\}$, is randomly draw in $[1, 1000]$,
- $p_i = w_i + 50$, $i \in \{1, ..., n\}$,
- $C = \frac{1}{2}. \sum_{i=1}^{n} w_i$.

For each instance, the average results displayed have been obtained with 10 problems.
A NVIDIA GTX 260 graphic card (192 cores, 1.4GHz) has been used and the parallel computational time is compared with the sequential one obtained on a CPU with an Intel Xeon 3.0GHz. Results on the memory occupancy are also presented.

5.1. Memory occupancy

In this section, we display the results obtained with the compression method presented in section 4. Table 1 shows the factor of compression computed as follow:

$$comp\_factor = \frac{size\ of\ M\_c + 2.\lceil n/32 \rceil}{size\ of\ M},$$

where M is the matrix of decision and M_c the corresponding compressed matrix. $2.\lceil n/32 \rceil$ corresponds to the values of lc and rc needed for each lines.

Table 1 shows that in the worst case the size of the compressed data ($size\ of\ M\_c + 2.\lceil n/32 \rceil$) corresponds to only 0.3% of the size of the initial the matrix M,

n	comp_factor	n	comp_factor
10000	0.00309	60000	0.00051
20000	0.00155	70000	0.00044
30000	0.00103	80000	0.00038
40000	0.00077	90000	0.00034
50000	0.00062	100000	0.00031

Table 1: Factor of data compression.

which leads to a very small memory occupancy as compared with the original dynamic programming algorithm. Furthermore, the factor of compression decreases with the size of the knapsack.
This method of compression reduces significantly the memory occupancy of the dynamic programming algorithm and is robust when the number of variables increases. This permits one to solve larger problems that could not be solved otherwise, like problems with 100000 variables.
Time spent for the compression step is presented in the next subsection, in order to be compared to the overall processing time.

5.2. Processing time

Table 2 presents the average processing time to solve KP obtained with the sequential and parallel algorithms. It also shows the corresponding average time spent during the compression step. Table 3 provides the resulting speedup.
We can see that the processing time cost of the compression step is relatively small as compared with the overall one. These results include the compression step and the transfer of data to the CPU. Thus, this approach is very efficient both in terms of memory occupancy and processing time.
The comparison of the parallel implementation with the sequential one shows that the resulting speedup increases with the size of the problem and meets a level around 26. Our parallel implementation of the dynamic programming reduces significantly processing time and shows that solving hard knapsack problems is possible on GPU.
The parallel implementation of the dynamic programming algorithm on GPU combined with our compression method permits one to solve large size problems within a small processing time and a small memory occupancy.

n	t. //	t. seq.	t. comp. //	t. comp. seq.
10000	3.06	58.95	0.11	1.08
20000	11.97	226.66	0.31	4.16
30000	26.57	536.14	0.71	9.27
40000	47.43	1225.52	1.23	18.66
50000	73.55	1912.43	1.85	25.66
60000	105.93	2752.81	3.25	38.14
70000	143.98	3739.74	3.61	50.15
80000	183.15	4771.55	4.69	64.09
90000	238.57	6184.28	5.95	82.56
100000	289.21	>7200	7.16	-

t. //: average parallel time
t. seq: average sequential time
t. comp. //: average parallel time for compression
t. comp. seq.: average sequential time for compression

Table 2: Processing time (s.).

n	speedup	n	speedup
10000	18.90	60000	25.98
20000	19.26	70000	25.97
30000	20.17	80000	26.05
40000	25.83	90000	25.92
50000	26.00	100000	-

Table 3: Speedup ($\frac{t.\ seq.}{t.\ //}$).

6. Conclusion

In this article we have proposed a parallel implementation of the dynamic programing algorithm for the knapsack problem on NVIDIA GPU with CUDA. This algorithm has been combined with data compression techniques. Computational experiences have shown that large size problems can be solved within small processing time and memory occupancy.
The proposed approaches, i.e. implementation on GPU and data compression, seems to be robust as the results are not deteriorated when the number

of variables increases. The observed speedup appears to be stable (around 26) for instances with more than 40000 variables. The reduction of the size on the matrix increases with the number of variables, resulting in a more efficient compression and the overhead does not exceed 3% of the overall one.
The proposed parallel algorithm to solve knapsack problems on GPU shows the relevance of using this type of architecture for combinatorial optimization. Further computational experiences are foreseen, in particular with a NVIDIA Tesla cards and hybrid supercomputers which are dedicated to hight performance computing.

References

[1] NVIDIA, Cuda 2.0 programming guide, http:// developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf (2009).

[2] D. El Baz, M. Elkihel, Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0-1 knapsack problem, Journal of Parallel and Distributed Computing 65 (2005) 74–84.

[3] D. C. Lou, C. C. Chang, A parallel two-list algorithm for the knapsack problem, Parallel Computing 22 (1997) 1985–1996.

[4] T. E. Gerash, P. Y. Wang, A survey of parallel algorithms for one-dimensional integer knapsack problems, INFOR 32(3) (1993) 163–186.

[5] G. A. P. Kindervater, H. W. J. M. Trienekens, An introduction to parallelism in combinatorial optimization, Parallel Computers and Computations 33 (1988) 65–81.

[6] J. Lin, J. A. Storer, Processor-efficient algorithms for the knapsack problem, Journal of Parallel and Distributed Computing 13(3) (1991) 332–337.

[7] D. Ulm, Dynamic programming implementations on SIMD machines - 0/1 knapsack problem, M.S. Project, George Mason University (1991).

[8] V. Boyer, D. El Baz, M. Elkihel, Heuristics for the 0-1 multidimensional knapsack problem, European Journal of Operational Research 199, issue 3 (2009) 658–664.

[9] V. Boyer, D. El Baz, M. Elkihel, A dynamic programming method with list for the knapsack sharing problem, The 39th International Conference on Computers & Industrial Engineering (CIE39), Troyes (France) (2009).

[10] H. Crowder, E. L. Johnson, M. W. Padberg, Solving large-scale zero-one linear programming problems, Operations Research 31 (1983) 803–834.

[11] B. L. Dietrich, L. F. Escudero, More coefficient reduction for knapsack-like constraints in 0-1 programs with variable upper bounds, IBM T.J. Watson Research Center RC-14389,Yorktown Heights (NY).

[12] B. L. Dietrich, L. F. Escudero, New procedures for preprocessing 0-1 models with knapsack-like constraints and conjunctive and/or disjunctive variable upper bounds, IBM T.J. Watson Research Center RC-14572,Yorktown Heights (NY).

[13] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, 2004.

[14] S. Martello, P. Toth, Knapsack Problems - Algorithms and Computer Implementations, Wiley & Sons, 1990.

[15] R. Bellman, Dynamic programming, Princeton University Press, princeton (1957).

[16] P. Toth, Dynamic programming algorithm for the zero-one knapsack problem, Computing 25 (1980) 29–45.