Computers & Operations Research 39 (2012) 42–47

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

Solving knapsack problems on GPU

V. Boyer [a,b,*], D. El Baz [a,b], M. Elkihel [a,b]

[a] CNRS, LAAS, 7 avenue du Colonel Roche, F-31077 Toulouse, France
[b] Université de Toulouse, UPS, INSA, INP, ISAE, LAAS, F-31077 Toulouse, France

ARTICLE INFO

Available online 8 April 2011

Keywords:
Combinatorial optimization problems
Dense dynamic programming
Parallel computing
GPU computing
CUDA

ABSTRACT

A parallel implementation via CUDA of the dynamic programming method for the knapsack problem on NVIDIA GPU is presented. A GTX 260 card with 192 cores (1.4 GHz) is 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 3.0 GHz. The results show a speedup factor of 26 for large size problems. Furthermore, in order to limit the communication between the CPU and the GPU, a compression technique is presented which decreases significantly the memory occupancy.

1. Introduction

Since a few years, graphics card manufacturers have developed technologies in order to use their cards for High Performance Computing (HPC); we recall that Graphics Processing Units (GPUs) are HPC many-core processors. In particular, NVIDIA has developed the Compute Unified Device Architecture (CUDA) so as to program efficiently its GPUs. CUDA technology is based on a Single Instruction, Multiple Threads (SIMT) programming model. The SIMT model is akin to Single Instruction, Multiple Data (SIMD) model [1].

Using GPU architectures for solving large scale or difficult optimization problems like combinatorial optimization problems is nevertheless a great challenge due to the specificities of GPU architectures.

We have been solving recently difficult combinatorial optimization of the knapsack family like multidimensional knapsack problems (see [2,3]), multiple knapsack problems [4] and knapsack sharing problems [5]. We are presently interested in the use of GPUs in order to speed up the solution of NP-hard problems. In particular, we are developing a series of parallel codes on GPUs that can be combined in order to produce efficient parallel methods. We have studied parallel branch and bound methods for the solution of knapsack problems (KP) on GPUs in [6]. We have also presented a parallel simplex algorithm on GPUs for linear programming problems and bound computation purpose in [7]. In this paper, we propose a parallel implementation of dynamic programming algorithm on GPU for KP (see also [8]). A data compression technique is also presented. This technique permits one to reduce significantly memory occupancy and communication between the CPU and the GPU. Parallel implementation of dense dynamic programming algorithm for KP on NVIDIA GPUs via CUDA is detailed.

The paper is structured as follows. Section 2 deals with related work. The knapsack problem and its solution via dynamic programming is presented in Section 3. Section 4 focuses on GPU computing and the parallel dynamic programming method. In Section 5, we propose a compression method that permits one to reduce significantly the memory occupancy and communication between CPU and GPU. Section 6 deals with computational experiments. Conclusions and future work are presented in Section 7.

2. Related work

Several parallel dynamic programming methods have been proposed for KP in the literature (see, for example: [9,10]). In particular, implementations on SIMD machines were performed on a 4K processor ICL DAP [11], a 16K Connection Machine CM-2 (see [12,13]) and a 4K MasPar MP-1 machine [13].

Reference is also made to [14] for a study on a parallel dynamic programming list algorithm using dominance techniques. Several load balancing methods have been proposed in [14,15] in order to improve algorithm efficiency. The parallel algorithm and load balancing methods have been carried out on an Origin 3800 SGI supercomputer. The reader is also referred to [16] for parallel dynamic programming algorithm for subset sum problems.

3. The knapsack problem

The knapsack problem is a NP-hard combinatorial optimization problem. It is one of the most studied discrete optimization

* Corresponding author at: CNRS, LAAS, 7 avenue du Colonel Roche, F-31077 Toulouse, France.
E-mail addresses: vboyer@laas.fr (V. Boyer), elbaz@laas.fr (D. El Baz), elkihel@laas.fr (M. Elkihel).

0305-0548/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2011.03.014

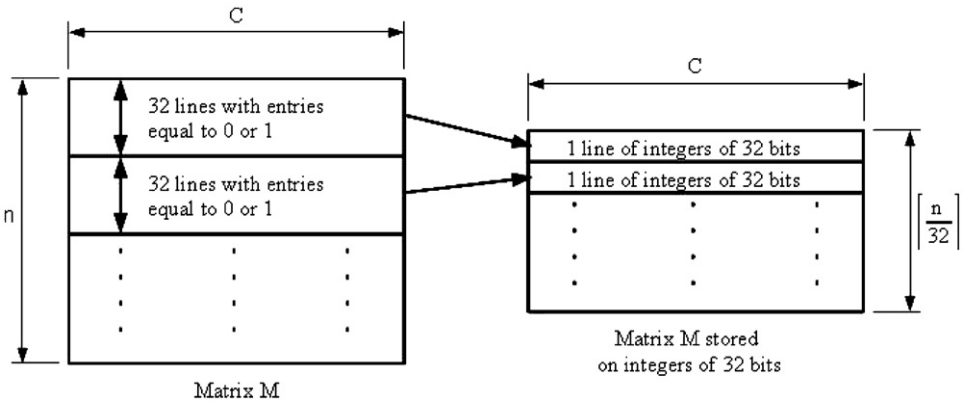

Fig. 1. Storage of matrix M.

problems as it is among the simplest prototypes of integer linear programming problems and it arises as a sub-problem of many complex problems (see, for example [2,17–19]).

3.1. 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 capacity $C \in \mathbb{N}_+^*$, KP can be defined as the following integer programming problem:

$$(\mathrm{KP}) \begin{cases} \max \sum_{i=1}^{n} p_i x_i, \\ \text{s.t.} \sum_{i=1}^{n} w_i x_i \le C, \\ x_i \in \{0,1\}, \quad i \in \{1,\ldots,n\}. \end{cases} \tag{3.1}$$

To avoid any trivial solution, we assume that

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

3.2. Dynamic programming

Bellman's dynamic programming [20] was the first exact method to solve KP. It consists in computing at each step $k \in \{1,\ldots,n\}$, the values of $f_k(\hat{c})$, $\hat{c} \in \{0,\ldots,C\}$, using the classical dynamic programming recursion:

$$f_k(\hat{c}) = \begin{cases} f_{k-1}(\hat{c}) & \text{for } \hat{c} = 0,\ldots,w_k-1, \\ \max\{f_{k-1}(\hat{c}), f_{k-1}(\hat{c}-w_k)+p_k\} & \text{for } \hat{c} = w_k,\ldots,C, \end{cases} \tag{3.2}$$

with $f_0(\hat{c}) = 0, \hat{c} \in \{0,\ldots,C\}$.

The algorithm, presented in this section, is based on the Bellman's recursion (3.2). A state corresponds to a feasible solution associated with the $f_k(\hat{c})$ value. Toth [21] 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 (*Toth [21]*). *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 procedure of Toth is described in Algorithm 1. The matrix M stores all the decisions and is used to build a solution vector, corresponding to the optimal value, by performing a backward procedure. The entries of M are denoted by $M_{i,\hat{c}}$ with $i \in \{1,\ldots,n\}$ and $\hat{c} \in \{1,\ldots,C\}$. The time and space complexities are $\mathcal{O}(nC)$.

Algorithm 1. (Dynamic programming).

for $\hat{c} \in \{0,\ldots,C\}, f(\hat{c}) := 0$,
for $i \in \{1,\ldots,n\}$ *and* $\hat{c} \in \{1,\ldots,C\}$, $M_{i,\hat{c}} = 0$,
$sumW := \sum_{i=1}^{n} w_i$,
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*
      $f(\hat{c}) := f(\hat{c}-w_k)+p_k$,
      $M_{k,\hat{c}} := 1$,
    end if,
  end for,
end for.
return $f(C)$ (*the optimal value of the KP*)

The high memory requirement is frequently cited as the main drawback of dynamic programming. However, this method has a pseudo-polynomial time complexity and is insensitive to the type of instances, i.e. with correlated data or not.

In order to reduce the memory occupancy, the entries of the matrix M, with value 0 or 1, are stored in integers of 32 bits (see Fig. 1), i.e. 32 lines of 0/1 entries are stored in 1 line of 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 5.

4. GPU computing

GPUs are highly parallel, multithreaded, many-core architectures. They are better known for image processing. Nevertheless, NVIDIA introduced in 2006 CUDA, a technology that enables users to solve many complex computational problems on GPU cards. At the time we have made this study, CUDA 2.0 was available.

NVIDIA's GPUs are SIMT (Single Instruction, Multiple Threads) architectures, i.e. the same instruction is executed simultaneously on many data elements by different threads. They are especially well-suited to address problems that can be expressed as

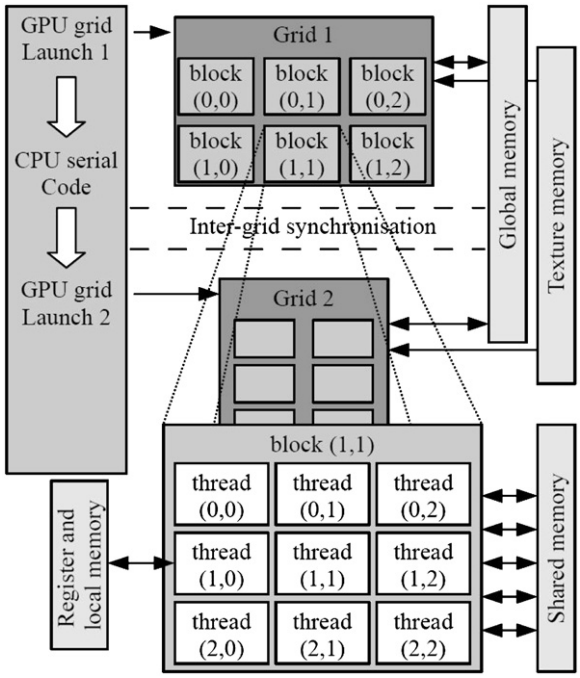

Fig. 2. Thread and memory hierarchy in GPUs.

data-parallel computations since GPUs devote more transistors to data processing than data caching and flow control. GPUs can nevertheless be used for task parallel applications with success.

As shown in Fig. 2, a parallel code on GPU, the so-called device, is interleaved with a serial code executed on the CPU, the so-called host. On the top level, threads are grouped into blocks. These blocks contain up to 512 threads and are organized in a grid which is launched via a single CUDA program, the so-called kernel.

When a kernel is launched, the blocks within a grid are distributed on idle multiprocessors. Threads that belong to different blocks cannot communicate explicitly; they can nevertheless share their results by means of a global memory. A 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. However, a divergent branch may lead to poor efficiency.

Threads have access to data from multiple memory spaces (see Fig. 2). Each thread has its own register and private local memory. Each block has a shared memory (with high bandwidth) only visible to all threads of the block and which has the same lifetime as the block. Finally, all threads have access to a global memory. Furthermore, there are 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, optimized for 2D spatial locality.

In order to have a maximum bandwidth for the global memory, memory accesses have to be coalesced. Indeed, the global memory access by all threads within a half-warp (a group of 16 threads) is done in one or two transactions if:

- the size of the words accessed by the threads is 4, 8, or 16 bytes;
- all 16 words lie:
  - in the same 64-byte segment, for words of 4 bytes;
  - in the same 128-byte segment, for words of 8 bytes;
  - in the same 128-byte segment for the first eight words and in the following 128-byte segment for the last eight words, for words of 16 bytes;
- 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, which degrades significantly the overall processing time. The reader is referred to [1] for further details on the NVIDIA cards architecture and how to optimize the code.

4.1. Parallel dynamic programming

The parallel implementation of the dynamic programming method has been especially designed for NVIDIA GPU architectures. The main computing part of this method is the loop that processes the values of $f(\hat{c})$, $\hat{c} \in \{0,\ldots,C\}$. This step has been parallelized on GPU and each thread computes a value of f. Many efforts have been made in order to limit the communication between the CPU and the GPU and to 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).

$n\_lines \coloneqq \lceil n/32 \rceil$,
Variables stored in the device (GPU):
for $\hat{c} \in \{0,\ldots,C\}$ *do*
  $f0\_d(\hat{c}) \coloneqq 0$ *and* $f1\_d(\hat{c}) \coloneqq 0$,
  $m\_d_{\hat{c}} \coloneqq 0$,
end for
Variables stored in the host (CPU):
for $i \in \{1,\ldots,n\_lines\}$ *and* $\hat{c} \in \{1,\ldots,C\}$, $M_{i,\hat{c}} \coloneqq 0$,
$sumW \coloneqq \sum_{i=1}^{n} w_i$,
$bit\_count \coloneqq 0$,
for k from 1 to n do
  $sumW \coloneqq sumW - w_k$,
  $\underline{c} \coloneqq max\{C - sumW, w_k\}$,
  $bit\_count \coloneqq bit\_count + 1$,
  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*
    $bit\_count \coloneqq 0$,
    copy m_d in the host and update M,
    for $\hat{c} \in \{0,\ldots,C\}$, $m\_d_{\hat{c}} \coloneqq 0$,
  end if
end for.
if n is even then
  return $f1\_d(C)$,
else
  return $f0\_d(C)$.

In Algorithm 2, the vector m_d corresponds to a line of the matrix M, it is stored in the global memory of the device. Since the entries of M are stored in integers of 32 bits, the variable bit_count is used to determine when to retrieve the data saved in m_d from the device in order to update M (this is done every 32 iterations).

On the GPU, The launching of the threads 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* denotes the minimum value of $\hat{c}$.

In order to avoid a copy of *output_f* in *input_f* and to save processing time, these vectors are switched at each iteration. That is the reason why the kernel *Compute_f_and_m_on_device* appears twice in Algorithm 2 since at iteration *k*:

- *input_f*=*f0_d* and *output_f*=*f1_d*, if *k* is even;
- *input_f*=*f1_d* and *output_f*=*f0_d*, otherwise.

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* ($k \in \{1,\ldots,n\}$),
$i := (k+31)\%32$: *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*
  $output\_f(\hat{c}) := input\_f(\hat{c}-w_k)+p_k$,
  $output\_m_{\hat{c}} := output\_m_{\hat{c}} + 2^i$,
else
  $output\_f(\hat{c}) := input\_f(\hat{c})$,
end if

In 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 4. 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 dynamically contrarily to the constant memory; *output_f* and *output_m* are stored in the global memory.

5. Reducing memory occupancy

In Algorithm 2, a 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*. This step is time consuming and we aim at further reducing the amount of data transferred to the CPU.

The analysis of the values stored in the vector *m_d* shows that its right columns are often filled with 1 and that its left columns are filled with 0. Since these bit values are grouped as integers of 32 bits, it corresponds in practice to the value $2^{32}-1$ for the right columns filled with 1 (and 0 for the left columns filled with 0). In the sequel, we will take $a=2^{32}-1$. Fig. 3 gives the basic principle of reduction of memory occupancy for a typical vector *m_d*.

As shown in Fig. 3, we define *m_d_c*, the compressed vector of *m_d* as follows:

$$\text{for } \hat{c} \in \{0,\ldots,rc-lc\}, m\_d\_c_{\hat{c}} = m\_d_{\hat{c}+lc},$$

$$\text{with } lc = \min\{\hat{c} \in \{1,\ldots,C\} | m\_d_{\hat{c}-1} = 0 \text{ and } m\_d_{\hat{c}} \neq 0\},$$

$$rc = \max\{\hat{c} \in \{1,\ldots,C\} | m\_d_{\hat{c}-1} \neq a \text{ and } m\_d_{\hat{c}} = a\}.$$

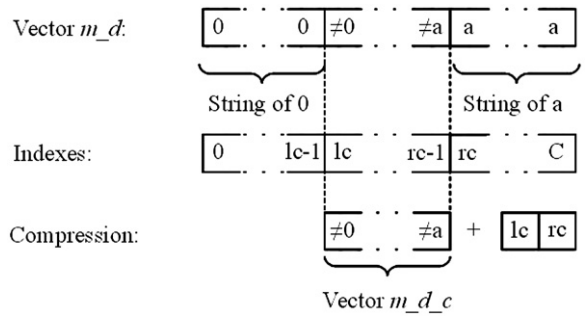

Fig. 3. Principle of reduction of memory occupancy.

Thus, we know that, for $\hat{c} \in \{0,\ldots,C\}$:

- if $\hat{c} < lc$, then $m\_d_{\hat{c}} = 0$,
- if $\hat{c} \geq rc$, then $m\_d_{\hat{c}} = a$,
- else $m\_d_{\hat{c}} = m\_d\_c_{\hat{c}-lc}$.

Then, we retrieve only the values of $m\_d_{\hat{c}}$ for $\hat{c} \in \{lc,\ldots,rc-1\}$ 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*
  $lc := \min\{\hat{c}, lc\}$,
end if,
if $m\_d_{\hat{c}-1} \neq a$ *and* $m\_d_{\hat{c}} = a$ *then*
  $rc := \max\{\hat{c}, rc\}$,
end if.

This compression method permits one to reduce significantly the memory occupancy needed to store all the decisions made throughout the dynamic programming recursion and the amount of data transferred from the GPU to the CPU. Computational experiments show 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:

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

This sorting is quite natural, since it is the one of the greedy algorithm for KP with connections to its continuous relaxation.

6. Computational experiments

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

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

Indeed, dense dynamic programming is well known to be well suited to correlated instances; nevertheless, this algorithm is not sensible to the type of correlation.

Table 1
Factor of data compression.

n	*comp_factor*
10,000	0.00309
20,000	0.00155
30,000	0.00103
40,000	0.00077
50,000	0.00062
60,000	0.00051
70,000	0.00044
80,000	0.00038
90,000	0.00034
100,000	0.00031

Table 2
Processing time (s).

n	t. //	t. seq.	t. comp. //	t. comp. seq.
10,000	3.06	58.95	0.11	1.08
20,000	11.97	226.66	0.31	4.16
30,000	26.57	536.14	0.71	9.27
40,000	47.43	1225.52	1.23	18.66
50,000	73.55	1912.43	1.85	25.66
60,000	105.93	2752.81	3.25	38.14
70,000	143.98	3739.74	3.61	50.15
80,000	183.15	4771.55	4.69	64.09
90,000	238.57	6184.28	5.95	82.56
100,000	289.21	> 7200	7.16	–

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

Table 3
Speedup (t. seq./t. //).

n	Speedup
10,000	18.90
20,000	19.26
30,000	20.17
40,000	25.83
50,000	26.00
60,000	25.98
70,000	25.97
80,000	26.05
90,000	25.92
100,000	–

For each problem, we display the average results obtained for 10 instances. The problems are available at [22]. A NVIDIA GTX 260 graphic card (192 cores, 1.4 GHz) has been used and the parallel computational time is compared with the sequential one obtained on a CPU with an Intel Xeon 3.0 GHz. Results on the memory occupancy are also presented. CUDA 2.0 has been used for the parallel code and g++ for the serial one.

6.1. Memory occupancy

In this subsection, we display the results obtained with the compression method presented in Section 5. Table 1 shows the factor of compression computed as follows:

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

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

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 matrix M, 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 the memory occupancy of the dynamic programming algorithm significantly and is robust when the number of variables increases. This permits one to solve large problems that could not be solved otherwise, like problems with 100,000 variables.

Time spent for the compression step is presented in the next subsection, in order to be compared with the overall processing time.

6.2. Processing time

Table 2 presents the average processing time to solve KP obtained with the sequential and parallel algorithms. It also gives the corresponding average time spent during the compression step. Table 3 provides the resulting speedup.

Without the compression method, the solution of the presented instances requires an amount of memory which exceeds the one available on both the CPU and the GPU. Thus, no results are presented without the compression method.

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 factor increases with the size of the problem and meets a level around 26. Our parallel implementation of the dynamic programming reduces processing time significantly 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.

7. Conclusions and future work

In this article, we have proposed a parallel implementation with CUDA of the dynamic programming algorithm for the knapsack problem. The presented algorithm makes use of data compression techniques. Computational results have shown that large size problems can be solved within small processing time and memory occupancy.

The proposed approach, i.e. implementation on GPU and data compression, seems to be robust since the results do not deteriorate when the size of the problem increases. The observed speedup factor (around 26) appears to be stable for instances with more than 40,000 variables. Furthermore, the reduction of the matrix size improves when the size of the problem increases, resulting in a more efficient compression while the overhead does not exceed 3% of the overall processing time. The proposed parallel algorithm shows the relevance of using GPUs and CUDA technology for solving difficult combinatorial optimization problems in practice.

Further computational experiments are foreseen. In particular, we plan to implement our parallel algorithm in multi-GPU contexts. We plan also to make cooperate parallel dynamic

programming algorithms with parallel branch and bound algorithms in order to design efficient parallel algorithms on GPU for NP-complete combinatorial optimization problems.

Acknowledgment

The authors would like to thank the reviewers for their remarks. Didier El Baz wish to thank NVIDIA for providing support through Academic Partnership.

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] Boyer V, El Baz D, Elkihel M. Heuristics for the 0–1 multidimensional knapsack problem. European Journal of Operational Research 2009;199(3):658–64.
[3] Boyer V, El Baz D, Elkihel M. Solution of multidimensional knapsack problems via cooperation of dynamic programming and branch and bound. European Journal of Industrial Engineering 2010;4(4):434–49.
[4] Lalami M, Elkihel M, El Baz D, Boyer V. A procedure-based heuristic for 0–1 multiple knapsack problems, LAAS report 10035, to appear.
[5] Boyer V, El Baz D, Elkihel M. A dynamic programming method with lists for the knapsack sharing problem. Computers & Industrial Engineering, in press, doi:10.1016/j.cie.2010.10.015.
[6] El Baz D, et al. Parallélisation de méthodes de programmation entière sur GPU, Congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision, ROADEF'2010, Toulouse, France; 2010.
[7] Lalami M, Boyer V, El Baz D. Efficient implementation of the simplex method on a CPU-GPU system. In: 25th symposium IEEE IPDPSW, Anchorage, USA; 2011.
[8] Boyer V, El Baz D, Elkihel M. Programmation dynamique dense sur GPU, Congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision, ROADEF'2010, LAAS report 09740, Toulouse, France; 2009.
[9] Lou DC, Chang CC. A parallel two-list algorithm for the knapsack problem. Parallel Computing 1997;22:1985–96.
[10] Gerash TE, Wang PY. A survey of parallel algorithms for one-dimensional integer knapsack problems. INFOR 1993;32(3):163–86.
[11] Kindervater GAP, Trienekens HWJM. An introduction to parallelism in combinatorial optimization. Parallel Computers and Computations 1988;33:65–81.
[12] Lin J, Storer JA. Processor-efficient algorithms for the knapsack problem. Journal of Parallel and Distributed Computing 1991;13(3):332–7.
[13] Ulm D. Dynamic programming implementations on SIMD machines—0/1 knapsack problem. M.S. Project, George Mason University; 1991.
[14] El Baz D, Elkihel M. Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0–1 knapsack problem. Journal of Parallel and Distributed Computing 2005;65:74–84.
[15] Elkihel M, El Baz D. Load balancing in a parallel dynamic programming multi-method applied to the 0–1 knapsack problem. In: 14th international conference on parallel, distributed and networked-based processing, PDP 2006, Montbéliard, France; 2006.
[16] Cosnard M, Ferreira AG, Herbelin H. The two list algorithm for the knapsack problem on a FPS T20. Parallel Computing 1989;9:385–8.
[17] Kellerer H, Pferschy U, Pisinger D. Knapsack problems. Springer; 2004.
[18] Martello S, Pisinger D, Toth P. New trends in exact algorithms for the 0–1 knapsack problem. European Journal of Operational Research 2000;123:325–32.
[19] Martello S, Toth P. Knapsack problems—algorithms and computer implementations. Wiley & Sons; 1990.
[20] Bellman R. Dynamic programming. Princeton University Press; 1957.
[21] Toth P. Dynamic programming algorithm for the zero–one knapsack problem. Computing 1980;25:29–45.
[22] Knapsack problems benchmark, 〈http://www.laas.fr/laas09/cda/23-31300-knapsack-problems.php〉.