Parallel Branch and Bound on a CPU-GPU System

Abdelamine Boukedjar, Mohamed Esseghir Lalami, Didier El-Baz
CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse France
Email: aboukedj@laas.fr mlalami@laas.fr elbaz@laas.fr

Abstract

Hybrid implementation via CUDA of a branch and bound method for knapsack problems is proposed. Branch and bound computations can be carried out either on the CPU or on the GPU according to the size of the branch and bound list, i.e. the number of nodes. Tests are carried out on a Tesla C2050 GPU. A first series of computational results showing a substantial speedup is displayed and analyzed.

1 Introduction

Graphics Processing Units (GPUs) are high-performance many-cores processors. Tools like Compute Unified Device Architecture (CUDA) or Open Computing Language (OpenCL) have been developed in order to use GPUs for general purpose computing; this has led to GPU computing and hybrid computing. CUDA-based NVIDIA GPUs are Single Instruction Multiple Thread (SIMT) architectures which is akin to Single Instruction Multiple Data (SIMD) architecture (see [17]).

In this paper, we concentrate on the implementation via CUDA of branch and bound algorithms on GPU for an important class of integer programming problems, i.e. Knapsack Problems (KP).

Knapsack problems occur in many domains like logistics, manufacturing, finance and telecommunications or as subproblems of hard problems in combinatorial optimization like multidimensional knapsack problems (see for example [1], [10] and [16]).
The knapsack problem is among the most studied discrete optimization problems; it is also one of the simplest prototypes of integer linear programming problems.
Several parallel algorithms have been proposed for KP (e.g. see [5], [7] and [15]). In particular, implementations on a SIMD machine have been performed on a 4K processor ICL DAP [11], a 16K Connection Machine CM-2 (see [14] and [18]) and a 4K MasPar MP-1 machine (see [18]).

We are interested in the solution via GPU of difficult combinatorial optimization problems like problems of the knapsack family. We are presently developing a series of parallel codes that will be combined in order to provide efficient parallel hybrid methods. In [3], we have presented an original implementation via CUDA of the dynamic programming method for KP on a CPU/GPU system with a single GPU. Experiments carried out on a CPU with 3 GHz Xeon Quadro Intel processor and GTX 260 GPU have shown substantial speedup. In [2], Boyer et al. have proposed an implementation via CUDA of the dense dynamic programming method on multi GPU architectures. This solution is well suited to the case where CPUs are connected to several GPUs; it is also particularly efficient. We have also proposed parallel simplex methods that run on a GPU or several GPUs (see [12] and [13]); these codes are particularly interesting when one wants to compute bounds of knapsack problems. We refer also to [19], for a study on local search methods and GPU computing for combinatorial optimization problems.

The use of CPU-GPU systems for solving difficult combinatorial optimization problems is a great challenge so as to reduce drastically the time needed to solve problems and the memory occupancy or to obtain exact solutions yet unknown. One of the difficulties of branch and bound methods is that they often lead to an irregular data structure that is not well suited to GPU computing. In this paper, we propose an original hybrid implementation of the branch and bound method via CUDA. The reader is referred to [6] for a survey on parallel branch and bound algorithms.

The knapsack problem and the branch and bound method are presented in Section 2. Section 3 deals with the implementation via CUDA of the branch and bound method on the CPU-GPU system. Computational results are displayed and analyzed in Section 4. Section 5 deals with conclusions and future work.

2 Knapsack problem

Given a set of items $i \in \{1, ..., n\}$, with profit $p_i \in \mathbb{N}_+^*$ and weight $w_i \in \mathbb{N}_+^*$ and a knapsack with the capacity

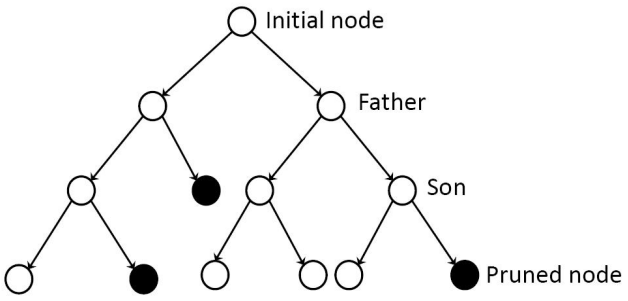

Figure 1. Data exchange between CPU and GPU

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

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

To avoid any trivial solution, we assume that we have

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

This insures that each item i fits into the knapsack and the overall weight sum of the items exceeds the knapsack capacity.

2.1 Branch and bound method

The branch and bound method is a general method that permits one to find optimal solution of various optimization problems like discrete and combinatorial optimization problems (see in particular [10] and [16]). It is based on enumeration of all candidate solutions and pruning of large subsets of candidate solutions via computation of lower and upper bounds of the criterion to optimize (see Figure 1). We concentrate here on breadth-first search strategy that is well suited to parallel implementation on GPU. We assume that items are sorted according to decreasing profit per weight ratio. At each step of the branch and bound method, the same branching and bounding tasks are carried out on a list of states called also nodes. Let k denote the index of the current item relative to a branching step. We denote by N_e^k the set of items in the knapsack at step k for a given node e. A node e is usually characterized by a tuple $(w_e, p_e, X_e, U_e, L_e)$ where w_e represents the weight of node e, p_e represents the profit of the node, X_e is the solution subvector associated with node e, U_e and L_e, respectively, are an upper bound and a lower bound of the node, respectively. We have:

$$w_e = \sum_{i \in N_e^k} w_i, p_e = \sum_{i \in N_e^k} p_i.$$

The so-called Dantzig bound (see [4]), derived from the solution of the continuous knapsack problem is a classical upper bound; it is given as follows:

$$U_e = p_e + \sum_{i=k+1}^{s_e-1} p_i + \left\lfloor \underline{C}.\frac{p_{s_e}}{w_{s_e}} \right\rfloor,$$

where s_e is the so-called slack variable such that

$$\sum_{i=k+1}^{s_e-1} w_i \leq C - w_e < \sum_{i=k+1}^{s_e} w_i,$$

(we note that one can compute index s_e for a node e and item k+1 via a greedy algorithm applied to a knapsack of capacity $C - w_e$) and $\underline{C}$ is the residual capacity given by

$$\underline{C} = C - w_e + \sum_{i=k+1}^{s_e-1} w_i.$$

Classically, one can obtain also a lower bound of the problem with a feasible solution of the knapsack problem with $\{0,1\}$ variables. For example, a lower bound L_e, can be computed via a greedy algorithm after selection of all items before the slack variable, s_e, since we have assumed that items are sorted by decreasing price per weight ratio.

For the sake of simplicity and efficiency, the following representation of a given node e will be used in the sequel: $(\hat{w}_e, \hat{p}_e, s_e, U_e, L_e)$, where

$$\hat{w}_e = w_e + \sum_{i=k+1}^{s_e-1} w_i,$$

and

$$\hat{p}_e = p_e + \sum_{i=k+1}^{s_e-1} p_i.$$

So, we have:

$$U_e = \hat{p}_e + \left\lfloor \underline{C}.\frac{p_{s_e}}{w_{s_e}} \right\rfloor.$$

We note that bound computation is time consuming in the branch and bound method and obtaining this way the bounds U_e and L_e is more efficient.

2.2 Branching

If $k < s_e$, then a node generates two sons at step k :

- a node with $x_k = 0$, $\hat{w}_e = \hat{w}_e - w_k$ and $\hat{p}_e = \hat{p}_e - p_k$;
- a node with $x_k = 1$ (in this case, no new node is created and the father is just kept in the list).

The case where $k = s_e$ yields only one son with $x_k = 0$. This comes to keep the father in the list and set $s_e = s_e + 1$. Indeed, item k cannot be packed into the knapsack.

2.3 Bound computation

We note that the upper and lower bounds do not change in the case where $x_k = 1$.

In the case where $x_k = 0$, one has to compute the new slack variable. So, $\hat{w}_e$ and $\hat{p}_e$ are updated as presented in subsection 2.1. Consequently, the upper and lower bounds U_e and L_e, respectively, are updated.

Remark: Pruning subset of candidate solutions is then done via comparison of the best current lower bound with the upper bound of each node. We denote by $\bar{L}$ the best lower bound. If we have $U_e \leq \bar{L}$, then node e is discarded.

3 Hybrid computing

3.1 CPU/GPU systems and CUDA

NVIDIA GPU cards and computing systems are highly parallel, multithreaded, many-core architectures. GPU cards are well known for image processing applications. NVIDIA has introduced in 2006 the Compute Unified Device Architecture (CUDA). CUDA is a software development kit that enables users to solve many complex computational problems on GPU cards. Our parallel dynamic programming code has been implemented via CUDA 3.2.

CUDA-based GPUs and computing systems are Single-Instruction, Multiple-Threads (SIMT) architectures, i.e. the same instruction is executed simultaneously on many data elements by different threads. GPUs are well suited to address problems that can be expressed as data-parallel computations since GPUs devote more transistors to data processing than to data caching and flow control. GPUs can nevertheless be used for task parallel applications with success.

As shown in Figure 2, a parallel code on GPU (the so-called device) is interleaved with a serial code executed on the CPU (the so-called host). At the top level, the threads are grouped into blocks. These blocks contain up to 1024 threads and are organized in a grid which is launched via a single CUDA program (the so-called kernel).

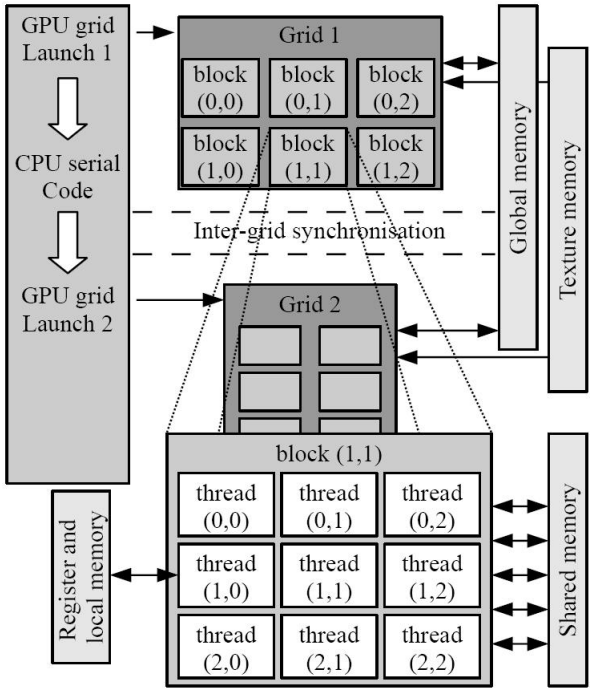

Figure 2. Thread and memory hierarchy in GPUs

When a kernel is launched, the blocks whithin the grid are assigned to idle groups of processors, the so-called multiprocessors. Threads in different blocks cannot communicate with each other explicitly. They can nevertheless share their results by means of a global memory.

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. However, a divergent branch may lead to a poor efficiency.

Threads have access to data from multiple memory spaces (see Figure 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 whithin 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 8 words and in the following 128-byte segment for the last 8 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. This degrades significantly the overall processing time.

Reference is made to [17] for further details on the NVIDIA GPUs and computing systems architecture and how to optimize the code.

In this paper, we study the parallel implementation of the branch and bound methods on a Tesla C2050 GPU.

3.2 Parallel algorithm

Bound computation is particularly time consuming in branch and bound algorithm. Nevertheless, this task can be efficiently parallelized on GPU.
The main tasks of our parallel algorithm are presented below (see also Figure 3).

Task carried out on the CPU

- Perform branch and bound algorithm on CPU when the size of the list is small.
- Transfer the current list of nodes to the GPU.
- Launch branching phase on GPU.
- Launch bound computation phase on GPU.
- Get the sublist of created nodes.
- Discard the nonpromising nodes.

Task carried out on the GPU

- Perform branching (kernel 1).

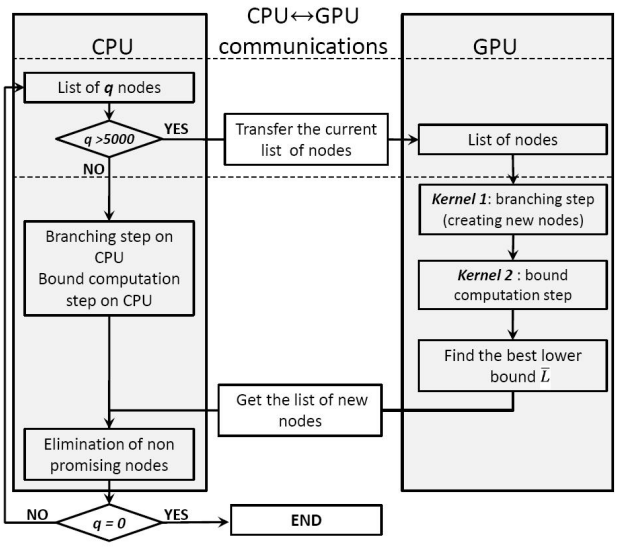

Figure 3. Data exchange between CPU and GPU

- Perform bound computation (kernel 2).
- Find the best lower bound $\bar{L}$.

In the sequel, we detail the different tasks carried out on the host and device, respectively. We begin by the tasks implemented on the device.

3.2.1 Computations on GPU

We note that each thread in the grid of blocks performs computation on only one node of the branch and bound list, so as to have coalesced memory access.

The table of items that contains weight w and profit p is stored in the texture memory in order to reduce the memory latency. As a consequence, these variables will not be referred to as arguments in the different kernels presented below so as to simplify algorithmic notation.

Branching

The kernel 1 corresponds to the branching step carried out on the GPU. The e-th thread branches on the node e and creates a new node at the address e.

- In the case where $k < s_e$, the e-th thread computes the value of $(\hat{w}_e, \hat{p}_e, s_e)$ of its son with $x_k = 0$. We recall that its son with $x_k = 1$ is already in the list stored in the CPU.

- The case where $k = s_e$, is treated differently: the slack variable of the son with $x_k = 0$ is updated as follows:

$$s_e = s_e + 1,$$

and values of $\hat{w}_e$ and $\hat{p}_e$ are not modified. Moreover, node e with $x_k = 1$ must be labelled as a nonpromising node in the CPU; this is done by assigning

$$U_e^{CPU} = 0.$$

Kernel 1

```
global_ void Kernel 1(int *ŵ, int *p̂, int *s, int *U^CPU, int k)
{
int e = blockIdx.x * blockDim.x + threadIdx.x ;
int s = s [e];
if (k < s)
    {
    AtomicSub(& ŵ [e] , w [k]);
    AtomicSub(& p̂ [e] , p [k]);
    }
else
    {
    AtomicAdd(& s [e] , 1);
    U^CPU [e] = 0;
    }
}
```

Bound computation

The parallel bound computation procedure computes bounds of new nodes, this is made via kernel 2 which is a loop of $n - h$ iterations; where $h = \max\{k, s\}$ and s is the first slack variable computed when the knapsack is empty. At each new iteration, a new item is considered and the current value of $\hat{w}_j$ and $\hat{p}_j$ and of the lower bound L_j are updated. Moreover, it may happen that the upper bound U_j is computed according to some condition that will be detailed in the sequel.

We note that the knapsack capacity C is stored in the constant memory of the GPU since it remains constant. This permits one to reduce memory latency.

When kernel 2 is launched, the j-th thread updates at iteration i, ($\hat{w}_j$, $\hat{p}_j$, s_j, L_j) if the item i has not been considered yet by this node i.e. if $i \geq s_j$.

The upper bound U_j is computed and $\hat{w}_j$, $\hat{p}_j$ and s_j are updated as soon as the condition $w+w_i \leq C$ is not satisfied. We note that the lower bound L_j is obtained only at the end of the procedure, i.e. at iteration n.

Kernel 2

```
global_ void Kernel 2 (int *ŵ, int *p̂, int *s, int *L, int *U, int h)
{
int j = blockIdx.x * blockDim.x + threadIdx.x + q;
int i = h;
int w = ŵ [j];
int p = p̂ [j];
int s = s [j];
int wi, pi;
While(i ≤ n)
    {
    wi = w [i];
    pi = p [i];
    /* Update of ŵ_j, p̂_j, s_j and compute U_j */
    if (i ≥ s)
        {
        if (w + wi ≤ C)
            {
            w = w + wi;
            p = p + pi;
            }
        else
            {
            U [j] = p + (C − w) ∗ pi/wi;
            ŵ [j] = w;
            p̂ [j] = p;
            s [j] = i;
            break;
            }
        }
        i = i + 1;
    }
While(i ≤ n)
    {
    wi = w [i];
    pi = p [i];
    /* Compute L_j */
    if (w + wi ≤ C)
        {
        w = w + wi;
        p = p + pi;
        }
    i = i + 1;
    }
    L [j] = p;
}
```

Finding the best lower bound

The best lower bound $\bar{L}$ is obtained in the GPU via a reduction method making use of the atomic instruction $atomicMax$ applied to the table of lower bounds (see [8]).

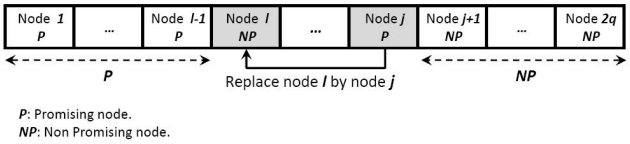

Figure 4. Structure of the list of nodes and substitution of non promising nodes

3.2.2 Computations on CPU

Branch and bound algorithm

If the size of the list is small, then it is not efficient to launch the branch and bound computation kernels on GPU since the GPU occupancy would be very small and computations on GPU would not cover communications between CPU and GPU. This is the reason why the branch and bound algorithm is implemented on CPU in this particular context. We note that for a given problem, the branch and bound computation phases can be carried out several times on the CPU according to the result of the pruning procedure. In this study, GPU kernels are launched only when the size of the list, denoted by q, is greater than 5000 nodes (see Figure 3). We have noticed that this condition ensures generally a 100 $\%$ occupancy for the GPU.

Pruning

This procedure starts after the list of nodes and the best lower bound $\bar{L}$ have been transfered from the GPU to the CPU. The size q of the list is updated by taking a value of q which is twice as much as its original value, then the list is treated from $e = 1$ to q. A node e is considered to be non promising (NP) if $U_e \leq \bar{L}$ otherwise it is promising (P) (see Figure 4).

Non promising states are replaced via an iterative procedure that starts from the beginning of the list. The different steps of the procedure that permits one to replace a non promising node with index l by a promising node are presented below.

- search a promising node j starting form the end of the list.
- copy data ($\hat{w}_j$, $\hat{p}_j$, s_j, U_j) of the promising node j in tuple ($\hat{w}_l$, $\hat{p}_l$, s_l, U_l).
- update the size of the list as follows :

$$q = j - 1.$$

4 Computational results

The CPU/GPU system considered in this study is a DELL Precision T7500 Westmere with processor Quad-Core Intel Xeon E5640 2.66 GHz and 12 GB of main memory and NVIDIA Tesla C2050 GPU. The Tesla C2050 GPU, which is based on the new-generation CUDA architecture codenamed Fermi, has 3 GB DDR5 of memory and 448 streaming processor cores (1.15 GHz) that deliver a peak performance of 515 Gigaflops in double precision floating point arithmetic. The interconnection between the host and device is done via a PCI-Express Gen2 interface. We have used CUDA 3.2 for the parallel code and gcc for the serial one.

We have carried out computational tests on randomly generated correlated problems; the problems are available at [9]. They present the following features:

- w_i, $i \in \{1, ..., n\}$, is randomly drawn in $[1, 100]$,
- $p_i = w_i + 10$, $i \in \{1, ..., n\}$,
- $C = \frac{1}{2} . \sum_{i=1}^{n} w_i$.

For each problem, the displayed results correspond to an average of ten instances. We have ovserved that the best number of threads per block is 192.

Table 1. time on CPU and CPU-GPU system of the branch and bound algorithm

size n of the problem	time on CPU (s)	time on CPU-GPU(s)	speedup
100	1.59	0.41	3.84
200	4.85	0.91	5.33
300	9.82	1.44	6.80
400	10.94	1.27	8.61
500	13.39	1.44	9.27

Table 1 displays computational times of the branch and bound algorithm on the CPU and the CPU-GPU system.

We see that substantial speedup can be obtained by using the Tesla C2050 GPU.

The proposed parallel branch and bound algorithm permits one to reduce drastically the processing time. The more streaming processor cores of the Tesla C2050 GPU are made available for a given computation, the more threads are executed in parallel and better is the global performance.

In general, the speedup increases with the size of the problem. The speedup meets a level around 9.

The speedup depends greatly on the size and difficulty of the considered instance. In particular, the best speedups have been obtained for instances with great number of nodes. As a matter of fact, there are few nodes that are discarded in this case and the GPU occupancy is particularly important. For example, we have experienced speedup equal to 11 in some cases with 500 items.

We have also performed experiments for non correlated problems that turn out to be easy. In this last case, pruning is particularly important and thus sequential branch and bound is very efficient. Thus, implementation on a CPU-GPU system has given no speedup since all computation is performed on the CPU.

We consider the solution of hard problems of the knapsack family, like multidimensional knapsack problems or multiple knapsack problems, to become possible in reasonable time with the help of GPU architectures.

5 Conclusions and future work

In this paper, we have proposed an original parallel implementation via CUDA of the branch and bound method for knapsack problems on a CPU-GPU system with Tesla C2050 GPU. The proposed approach is very efficient. In particular, computational results have shown that difficult problems can be solved within small processing time. This work shows the relevance of using CPU/GPU systems for solving difficult combinatorial optimization problems.

Our approach seems to be robust since the results remain good when the size of the problem increases.

We believe that further speedup can be obtained on multi GPU clusters.

We are currently parallelizing a series of methods for integer programming problems like dynamic programming and Simplex methods. The combination of parallel methods will permit us to propose efficient hybrid computing methods for difficult integer programming problems like multidimensional knapsack problems, multiple knapsack problems and knapsack sharing problems.

Acknowledgment

Dr Didier El Baz would like to thank NVIDIA for his support through Academic Partnership.

References

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

[2] V. Boyer, D. El Baz, and M. Elkihel. Dense dynamic programming on multi gpu. pages 545–551, 2011. Ayia Napa (Cyprus).

[3] V. Boyer, D. El Baz, and M. Elkihel. Solving knapsack problems on gpu. *Computers and Operations Research*, 39:42–47, 2012.

[4] G. B. Dantzig. Discrete variable extremum problems. *Operations Research*, 5:266–277, 1957.

[5] D. El Baz and 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*, 65:74–84, 2005.

[6] B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operation Research*, 42:1042–1066, 1994.

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

[8] M. Harris. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology report*, 2007.

[9] http://www.laas.fr/laas09/CDA/23-31300-Knapsack-problems .php. Knapsack problems benchmark.

[10] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

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

[12] M. Lalami, V. Boyer, and D. El Baz. Efficient implementation of the simplex method on a cpu-gpu system. 2011 IEEE IPDPS Workshops(IPDPSW 2011), 2011. Anchorage (USA).

[13] M. Lalami, D. El Baz, and V. Boyer. Multi gpu implementation of the simplex algorithm. 13th IEEE International Conference on High Performance Computing and Communications(HPCC 2011), 2011. Banff (Canada).

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

[15] D. C. Lou and C. C. Chang. A parallel two-list algorithm for the knapsack problem. *Parallel Computing*, 22:1985–1996, 1997.

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

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

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

[19] T. Van Luong, N. Melab, and E. G. Talbi. Large neighborhood local search optimization on graphics processing units. Proceedings of IEEE IPDPS, 2010. Atlanta, USA.