

GPU Implementation of the Branch and Bound method for knapsack problems

Didier El Baz, Mohamed Esseghir Lalami

▶ **To cite this version:**

Didier El Baz, Mohamed Esseghir Lalami. GPU Implementation of the Branch and Bound method for knapsack problems. 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, May 2012, Shanghai, China. pp.1763 - 1771, 10.1109/IPDPSW.2012.219 . hal-01149777

HAL Id: hal-01149777
https://hal.science/hal-01149777

Submitted on 13 May 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

GPU Implementation of the Branch and Bound method for knapsack problems

Mohamed Esseghir Lalami, Didier El-Baz
CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
Université de Toulouse, LAAS, F-31400 Toulouse, France
Email: mlalami@laas.fr elbaz@laas.fr

***Abstract*—In this paper, we propose an efficient implementation of the branch and bound method for knapsack problems on a CPU-GPU system via CUDA. Branch and bound computations can be carried out either on the CPU or on a GPU according to the size of the branch and bound list. A better management of GPUs memories, less GPU-CPU communications and better synchronization between GPU threads are proposed in this new implementation in order to increase efficiency. Indeed, a series of computational results is displayed and analyzed showing a substantial speedup on a Tesla C2050 GPU.**

***Keywords*-GPU computing, hybrid computing, CUDA, branch and bound method, knapsack problems, combinatorial optimization.**

I. Introduction and related works

Initially developed for real time and high-definition 3D graphic applications, Graphics Processing Units (GPUs) have gained recently attention for High Performance Computing applications. Indeed, the peak computational capabilities of modern GPUs exceeds the one of top-of-the-line central processing units (CPUs). GPUs are highly parallel, multithreaded, manycore units.
In November 2006, NVIDIA introduced, Compute Unified Device Architecture (CUDA), a technology that enables users to solve many computationally intensive or complex problems on their GPU cards.

In this paper, we consider an important class of integer programming problems, i.e. Knapsack Problems (KP). We propose an efficient approach based on multithreading in order to implement the branch and bound algorithm on a CPU-GPU system via CUDA.

Knapsack problems occur in many domains like logistics, manufacturing, finance and telecommunications; KP occur also very often as subproblems of hard combinatorial optimization problems like multidimensional knapsack problems (see for example [1] - [4]). The knapsack problem is among the most studied discrete optimization problems; it is also one of the simplest prototypes of integer linear programming problems.

Many parallel algorithms have been proposed for KP (see [5] - [7]). In particular, implementations on SIMD machines have been performed on a 4K processor ICL DAP (see [8]), a 16K Connection Machine CM-2 (see [9] and [10]) and a 4K MasPar MP-1 machine (see [10]).

We are presently developing a series of parallel codes on GPUs in order to solve difficult combinatorial optimization problems like problems of the knapsack family. These codes will be combined in order to provide efficient parallel hybrid methods. In [11], we have proposed an implementation of the dynamic programming method for KP on a CPU/GPU system via CUDA. Experiments carried out on a CPU with 3 GHz Xeon Quadro Intel processor and GTX 260 GPU have shown substantial speedup. This work was further extended in [12], where we have proposed an implementation via CUDA of the dynamic programming method on multi GPU architectures. This last solution is well suited to the case where CPUs are connected to several GPUs; it is also particularly efficient.

In [13], we have proposed a first parallel implementation of the branch and bound algorithm on a CPU-GPU system via CUDA. Experiments carried out on a system with a 3 Ghz Xeon Quadro INTEL processor and a Tesla C2050 GPU have shown a speedup of 9 as compared with results obtained on a single core of the CPU. In this paper, we propose a different hybrid implementation of the branch and bound method via CUDA. A better management of GPUs memories, less GPU-CPU communications and better synchronization between GPU threads are presented in this new implementation. This new implementation permits one to obtain a speed up twice as much as in our previous work.

We note that we have also proposed parallel Simplex methods that run on a single GPU or several GPUs (see [14] and [15]). These codes are particularly interesting when one wants for example to compute bounds of knapsack problems. Finally, we refer to [16], for a study on parallel local search methods for combinatorial optimization problems.

The use of CPU-GPU systems is challenging so as to reduce drastically the time needed to solve difficult combinatorial optimization problems and memory occupancy or to obtain exact solutions that could not be obtained otherwise. One of the difficulties with branch and bound methods is that they often lead to irregular data structure (see [17]). This last point is particularly challenging for computations carried out on GPUs.

Section II deals with knapsack problem and branch and bound method. The implementation of the branch and bound method on a CPU-GPU system via CUDA is proposed in Section III. We display and analyze computational results in

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.219

IEEE computer society

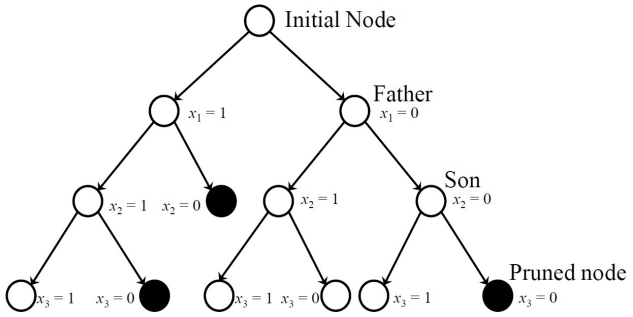


Figure 1. Branch-decision tree

Section IV. Section V presents some conclusions and future work.

II. Knapsack problem and Branch and Bound method

A. Problem formulation

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 $c \in \mathbb{N}_+^*$, problem KP can be defined as follows:

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

To avoid any trivial solution, we assume that we have:

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

These conditions ensure respectively that each item i should fit into the knapsack and the overall weight sum of the different items exceeds the knapsack capacity.

B. Branch and bound method

The branch and bound method is a general method that permits one to find exact optimal solution of various optimization problems like discrete and combinatorial optimization problems (see [3] and [4]). 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. We assume that items are sorted according to decreasing profit per weight ratio.

1) State notation: 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 [18]), 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 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, lower bound of a node can be obtained via a feasible solution of the knapsack problem with $\{0, 1\}$ variables. For example, a good lower bound L_e can be computed via a greedy algorithm and selection of all items after k and before slack variable, s_e, since we have assumed that items are sorted according to decreasing price per weight ratio. Hence, we have:

$$L_e = p_e + \sum_{i=k+1}^{s_e-1} p_i + \sum_{i=s_e+1}^{n} p_i.x_i.$$

where $x_i = 1$ if $w_i \leq \underline{c} - \sum_{j=s_e+1}^{i-1} w_j.x_j$

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

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

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

and

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

We note that we have:

$$L_e = \hat{p}_e + \sum_{i=s_e+1}^{n} p_i.x_i.$$

where $x_i = 1$ if $w_i \leq \underline{c} - \sum_{j=s_e+1}^{i-1} w_j.x_j$

Obtaining this way the bounds U_e and L_e is more efficient since bound computation is time consuming in the branch and bound method.

2) Branch and bound procedure: 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 $\hat{p}_e = \hat{p}_e - p_k$; in this case, one has to compute the new slack variable; moreover, the upper and lower bounds U_e and L_e, respectively, are updated according to the value of $\hat{p}_e$.
- a node with $x_k = 1$ (in this case, the son is similar to its father that is already in the list, in particular, the upper and lower bounds do not change; thus, no new node is created).

The case where $k = s_e$ yields only one son with $x_k = 0$, $\hat{w}_e = \hat{w}_e - w_k$, $\hat{p}_e = \hat{p}_e - p_k$ and $s_e = s_e + 1$. Indeed clearly, the k-th item cannot be packed into the knapsack.

Pruning a 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 can be discarded.

III. Implementation on a CPU-GPU system

In this Section, we present our recent contribution to the implementation of the branch and bound method on a CPU-GPU system. We shall see in Section IV that the improvements we propose in this paper have permitted us to increase substantially speedup as compared with the one obtained in our previous work (see [13]).

We begin this Section with a brief description of the GPU architecture.

A. GPU architecture

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

As shown in Figure 2, a *grid* is a set of blocks where each block contains up to 1024 threads. A grid is launched via a single CUDA program, the so-called kernel. The execution starts with a host execution, i.e. CPU execution. When a kernel function is invoked, the execution is moved to the device, i.e. the GPU. When all threads of a kernel have completed their execution, the corresponding grid terminates, the execution continues on the host until another kernel is invoked. When a kernel is launched, each multiprocessor processes one block by executing threads in group of 32 parallel threads named *warps*. Threads composing a warp start together at the same program address, they are nevertheless free to branch and execute independently. As blocks terminate, new blocks are launched on idle multiprocessors. With CUDA 3.0, threads of different blocks cannot communicate explicitly but can share their results by means of a global memory.

If threads of a given warp diverge when executing a data-dependent *conditional* branch, then the warp serially executes each branch path. This may lead to poor efficiency.

Threads have access to data from multiple memory spaces (see Figure 2). We can distinguish two main types of memory spaces.

- *Read-only memories*: these memories are the *constant* memory for constant data used by the process and *texture* memory optimized for 2D spatial locality. These two memories are accessible by all threads.
- *Read and write memories*: these memories are the *global* memory space accessible by all threads, the *shared* memory space accessible only by threads in the same block with high bandwidth; finally each thread has access to its own *registers* and *private local* memory space.

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

- The size of the words accessed by the threads is 4, 8, or 16 bytes.
- All 16 words lie
  - in the same 64-bytes segment, for words of 4 bytes;
  - in the same 128-bytes segment, for words of 8 bytes;
  - in the same 128-bytes segment for the first 8 words and in the following 128-bytes segment for the last 8 words, for words of 16 bytes.
- Threads access the words in sequence, i.e. the k-th thread in the half-warp has access to the k-th word.

Otherwise, a separate memory transaction is issued for each thread, which degrades significantly the overall processing time. For further details on the GPU architectures and how to optimize the codes, reference is made to [19].

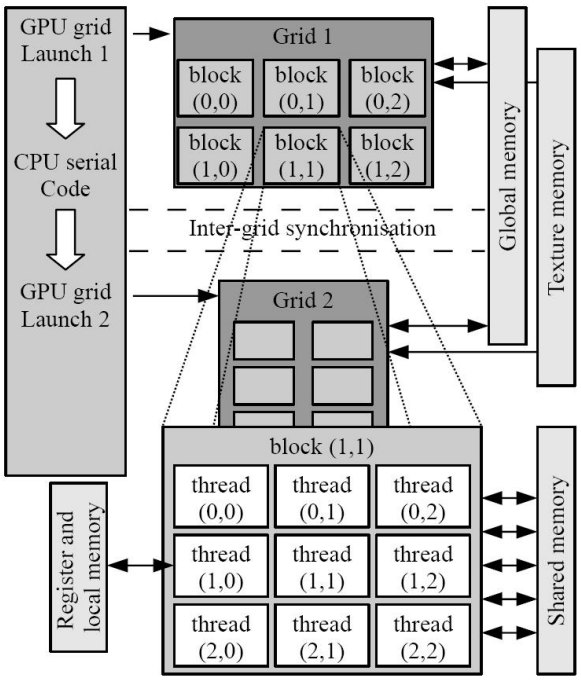


Figure 2. Thread and memory hierarchy in GPUs

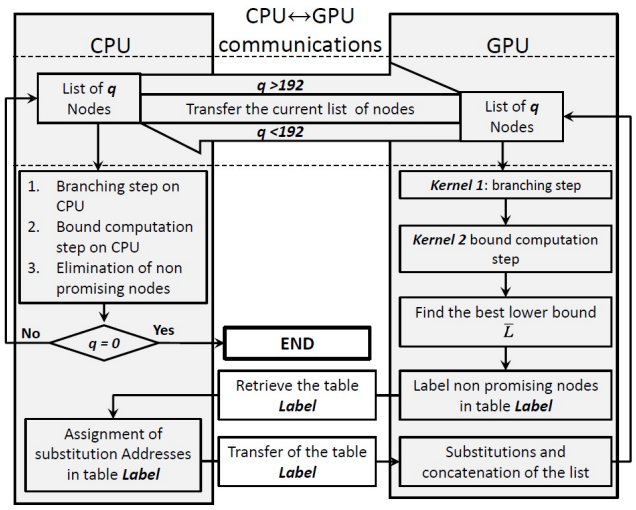


Figure 3. Branch and bound algorithm on a CPU-GPU system

B. Parallel implementation

Principle of the implementation

Bound computation is particularly time consuming in the branch and bound method. This task can be efficiently parallelized on the device. Branching can also be implemented efficiently on the device. The parallel implementation we propose can be summarized as follows (see also Figure 3).

Host

- Branch and bound computations when the size of the list is small.
- Transfer of the current list of nodes to the device if the number of nodes is greater than a given $threshold$.
- Launching branching operations on device.
- Launching bound computations on device.
- Launching labeling on device.
- Retrieval of the table of labeled nodes, denoted by $Label$.
- Assignment of substitution addresses in the table $Label$ and transfer to device.

Device

- Branching operations (Kernel 1).
- Bound computations (Kernel 2).
- Computation of the best lower bound $\bar{L}$.
- Nonpromising nodes labeling.
- Concatenation of the list of nodes via elimination of nonpromising nodes.

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.

1) Computations on the device: each thread in the grid of blocks performs computation on only one node of the branch and bound list, so as to obtain coalesced memory access.

The table of items that contains weights w and profits 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 in the sequel; this will simplify algorithmic notation.

Branching

Kernel 1 corresponds to the branching phase carried out on the device. The e-th thread branches on the node e and generates two sons: a son with x_k = 0 and son with x_k = 1. In practice, only one node is created in the list of nodes (see Figure 4); this node corresponds to the son with x_k = 0. We note that it is no need to create son with $x_k = 1$, since a similar node which corresponds to its father is already in the list of nodes. We have then to consider two cases.

- In the case where $k < s_e$, the e-th thread computes the value of $(\hat{w}_{e+q}, \hat{p}_{e+q}, s_{e+q})$ of the son with x_k = 0. We recall that the son with x_k = 1 is already in the list.
- in the case where $k = s_e$, the slack variable of the son

with x_k = 0 is updated as follows:

$$s_{e+q} = s_e + 1.$$

Moreover, node e with x_k = 1 must be labeled as a nonpromising node; this is done by assigning

$$U_e = 0.$$

In the Kernel 1, we note that the use of conditional instruction if may lead to branch divergence within a warp. However, only operations with registers are included in the conditional part of this new version of our code unlike operations of writing in global memory that were used in our previous work. This permits us in particular to be more efficient than with previous kernel proposed in [13].

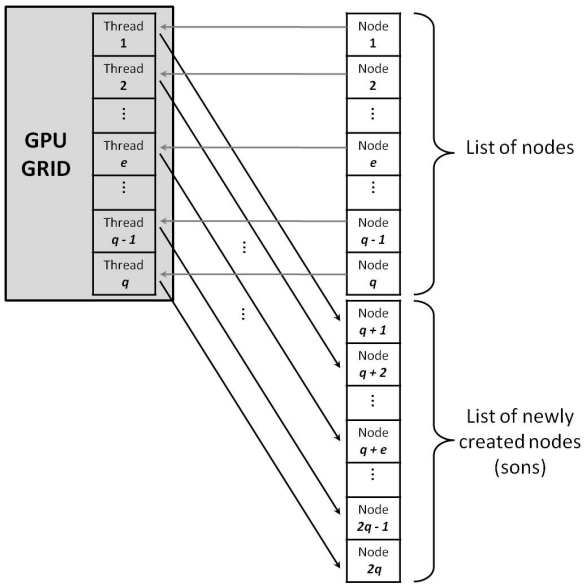


Figure 4. List of nodes and memory accesses of GPU threads.

Kernel 1

global_ void Kernel 1(int *$\hat{w}$, int *$\hat{p}$, int *s, int *U, int k, int q)
{
int e = blockIdx.x * blockDim.x + threadIdx.x ;
int se = $s[e]$, we = $\hat{w}[e]$, pe = $\hat{p}[e]$, Ue = $U[e]$;
if ($k < se$)
{
we = we - $w[k]$;
pe = pe - $p[k]$;
}
else
{
se = se + 1;
Ue = 0;
}
AtomicExch(& $\hat{w}[e+q], we$);
AtomicExch(& $\hat{p}[e+q], pe$);
AtomicExch(& $s[e+q], se$);
AtomicExch(& $U[e], Ue$);
}

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 $\hat{w}_j$, $\hat{p}_j$, the lower bound L_j and the upper bound U_j are updated. We note that the knapsack capacity c is stored in the constant memory of the GPU; this permits one to reduce memory latency.

When Kernel 2 is launched, the j-th thread updates ($\hat{w}_j$, $\hat{p}_j$, s_j, U_j, L_j) at the i-th iteration if the i-th item has not yet been considered by this node, i.e. if $i \geq s_j$. Then, we can distinguish, in the Kernel 2, three main parts: the first loop $while$, the second loop for and finally the third part starting at the first atomic operation.

The first loop is used in order to update the tuple ($\hat{w}_j$, $\hat{p}_j$, s_j, U_j) which is stored in the global memory of the GPU. However, this loop makes use of conditional instructions if that may lead to branch divergence within a warp. If this conditional part would include computations and accesses to global memory, then this could result in poor efficiency. In order to reduce the effect of branch divergence, we separate the computation part from the part related to writing in global memory. This is done by adding new variables $what$, $phat$, sj that are used in order to update the tuple ($\hat{w}_j$, $\hat{p}_j$, s_j) and that are stored in registers. Also, in order to reduce the number of operations in the conditional parts, U_j is computed concurrently between threads in the third part of the kernel 2 (by the operation $phat = phat + (c - what) * psj/wsj$;). However, computing the upper bound U_j requires weight and profit of the slack variable sj and in the same warp, the value of sj will differ between threads. Then, the access to this two data in the texture memory, is non coalesced which results in a memory latency. In order to avoid this memory access, two other variables named wsj and psj, stored in registers, are used to retrieve weight and profit of the slack variable sj in the first part of kernel 2 since these two data are already in wi and pi.

Thus, conditional parts include only computations on registers; as a result efficiency loss is reduced. The part related to global memory write, i.e. updates of $\hat{w}_j$, $\hat{p}_j$, s_j, U_j and

L_j is done in the end of Kernel 2 (the third part starting with atomic functions). In order to ensure also that threads of the same warp are concurrently run, the first loop $while$ is synchronized in the end of each iteration and all threads of the same warp exit the loop at the same time when all of them have already updated their tuple ($\hat{w}_j$, $\hat{p}_j$, s_j) by using the warp instruction ($if(__all(phat))\ break$;), since $phat$ was initialized to 0. Indeed, instruction $__all(phat)$ evaluates $phat$ for all threads of the warp and returns non-zero if and only if $phat$ evaluates to non-zero for all of them. The second loop for is used to compute the lower bound L_j. In practice this loop is bigger than the first one in terms of number of iterations. Then $unrolling$ this loop permits one to improve by 10% the overall computing time of the parallel branch and bound implementation. This is done by including three $While$ loops where the first one unrolls 100 times this loop for, the second one unrolls 10 times the loop for and the last one loops the residual iterations until $i = n$. Indeed, it is not possible to $unroll$ directly the loop for since the number of iterations is not known before the compilation of the program.

We note that the lower bound L_j is obtained only in 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 q, int h)
{
int j = blockIdx.x * blockDim.x + threadIdx.x + q;
int wj = ŵ [j], pj = p̂ [j], sj, = s [j];
int i = h, wi, pi, what, phat = 0, psj, wsj;

While (i ≤ n)
  {
  wi = w [i];
  pi = p [i];
  /* Compute what, phat, wsj, psj, sj which
  are used to get the Upper bound U_j*/
  if (i ≥ sj)
    {
    if (wj + wi ≤ c)
      {
      w = wj + wi;
      p = pj + pi;
      }
    else if (phat == 0)
      {
      wsj = wi;
      psj = pi;
      what = wj;
      phat = pj;
      sj = i;
      }
    }
  if (__all(phat)) {break;}
  __syncthreads();
  i = i + 1;
  }
for(;i ≤ n; i++)
  {
  wi = w [i];
  pi = p [i];
  /* Compute of the lower bound L_j*/
  if (wj + wi ≤ c)
    {
    wj = wj + wi;
    pj = pj + pi;
    }
  }
/* Update of the tuple (ŵ_j, p̂_j, s_j, U_j, L_j)
in the global memory of the GPU*/
AtomicExch(& ŵ[j], what);
AtomicExch(& p̂[j], phat);
AtomicExch(& s[j], sj);
phat = phat + (c − what) * psj/wsj;
AtomicExch(& U[j], phat);
AtomicExch(& L[j], pj);
}
```

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 [20]).

Label of the nonpromising nodes in table *Label*

At this step, the size of the list of nodes is increased by the number q of newly created nodes, i.e. q is updated by taking a value which is twice as much as its previous value. Then, we use a GPU kernel in order to label nonpromising nodes. This kernel returns a table named $Label$. A node e is considered to be nonpromising (NP) if $U_e \leq \bar{L}$ otherwise it is promising (P). If a node e is nonpromising, then the table element $Label[e]$ is set to 0, otherwise, it is set to 1. In the end of this step, the table $Label$ is transfered to the CPU.

Substitution and concatenation in the list of nodes

When the table $Label$ is retrieved, the elements $Label[l]$ set to zero at the previous step will contain the address j of promising nodes located in the right part of the the list of nodes, see Figure 5. Then, the substitution and the concatenation step is done via a GPU kernel where a thread l copies 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). In the end of this step, the size of the list is decreased by the number of nonpromising node. Then if $q < 192$, the list is transfered to the CPU, otherwise a

new item is considered.

The two previous steps are used in order to avoid transfer at each iteration of the whole list to the CPU. If the number of nodes remains important ($q \geq 192$), then the GPU carries out the different steps on the list and only a small size data communication (table *Label*) between the GPU and CPU is needed at each iteration of the branch and bound implementation. Also, since the three previous steps are simple, the design of their corresponding Kernels is facilitated.

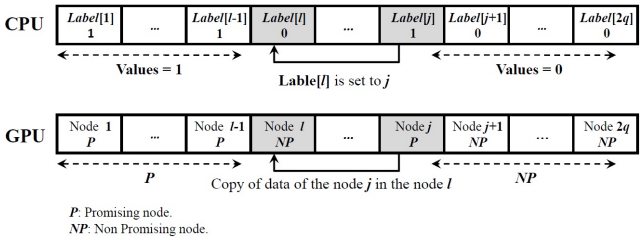


Figure 5. Structure of the list of nodes and substitution of non promising nodes

2) Computations on the host:

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 the device since the GPU occupancy would be very small and computations on the device would not cover communications between host and device. This is the reason why the branch and bound algorithm is carried out on the host 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 host 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 192 nodes (see Figure 3).

Assignment of substitution addresses to the table *Label*

This procedure starts after the table $Label$ has been transfered from the device to the host.

The elements of the table which are set to zero, i.e. elements related to nonpromising nodes are replaced via an iterative procedure that starts from the beginning of the table $Label$. The different steps of the procedure are presented below (see also Figure 5).

- Search in the table a non zero element $Label[j] = 1$ (corresponding to a promising node j) starting form the end of the table.
- Copy the address j in the element $Label[l]$ i.e. $Label[l] = j$.
- Update the size of the list as follows :

$$q = j - 1.$$

IV. Computational results

The CPU-GPU system considered in this study consists of 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, i.e. difficult problems. The problems are available at [21]. They present the following features:

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

For each size of problems, we display results we have obtained for an average of ten instances. We have obtained our best results with 192 threads per block.

Table I
TIME ON CPU AND CPU-GPU SYSTEM OF BRANCH AND BOUND ALGORITHM

size n of the problems	time on CPU (s)	time on CPU-GPU (s)	speedup
100	1.59	0.18	8.48
200	4.85	0.41	11.78
300	9.82	0.65	15.08
400	10.94	0.57	19.04
500	13.39	0.65	20.48

Table I displays computational times of sequential and parallel branch and bound methods carried out on the CPU and the CPU-GPU system, respectively. We see that substantial speedup can be obtained by making use of the Tesla C2050 GPU.

The proposed parallel implementation 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.

We note that the speedup increases with the size of the problem and meets a level around 20. The speedup depends greatly on the size and difficulty of the considered instances. In particular, the best speedups have been obtained for

instances with great number of nodes. As a matter of fact, the GPU occupancy is particularly important in this case since few nodes are discarded.

The speedup obtained with this implementation is generally twice as much as the one in [13], where noncoalesced global memory accesses may occur in conditional part of codes, leading to poor efficiency.

We have also performed experiments for non correlated problems that turn out to be easier. In this last case, pruning is particularly important and thus sequential branch and bound is very efficient. As a consequence, implementation on a CPU-GPU system gives no speedup in this case since most computations are performed on the CPU.

We consider the solution of hard problems of the knapsack family, like multidimensional knapsack problems or knapsack sharing problems, to become possible in reasonable time with the help of GPU architectures and combination of parallel branch and bound and dynamic programming (see [11] and [12]).

V. Conclusions and future work

In this paper, we have proposed an implementation of the branch and bound method on a CPU-GPU system via CUDA. Computational results show that our approach is efficient since we have obtained stable speedups around 20 for difficult knapsack problems. Our approach permits also one to solve problems with size 500 without exceeding the memory occupancy of the GPUs.

This work shows in particular the relevance of using CPU-GPUs systems for solving difficult combinatorial optimization problems like problems of the knapsack family.

In future work, we plan to consider the solution of very large knapsacks problems via several GPUs. Indeed, 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 Branch and Bound. The combination of these 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*, vol. 199, issue 3, pp. 658–664, 2009.

[2] B. L. Dietrich and L. F. Escudero, "More coefficient reduction for knapsack-like constraints in 0-1 programs with variable upper bounds," *IBM T.J. Watson Research Center*, vol. RC-14389,Yorktown Heights (NY), 1989.

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

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

[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*, vol. 65, pp. 74–84, 2005.

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

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

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

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

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

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

[12] ——, "Dense dynamic programming on multi gpu," in *Proceedings of the 19th International Conference on Parallel Distributed and networked-based Processing (PDP 2011)*, 2011, pp. 545–551, in Ayia Napa (Cyprus).

[13] A. Boukedjar, M. Lalami, and D. El Baz, "Parallel branch and bound on a cpu-gpu system," in *Proceedings of the 20th International Conference on Parallel, Distributed and network-based Processing (PDP 2012)*, 2012, pp. 392–398, in Garching (Germany).

[14] M. Lalami, V. Boyer, and D. El Baz, "Efficient implementation of the simplex method on a cpu-gpu system," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium and Workshops (IPDPSW 2011)*, 2011, in Anchorage (USA).

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

[16] T. Van Luong, N. Melab, and E. G. Talbi, "Large neighborhood local search optimization on graphics processing units," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium and Workshops (IPDPSW 2010)*, 2010, in Atlanta (USA).

[17] B. Gendron and C. T. G., "Parallel branch-and-bound algorithms: Survey and synthesis," *Operation Research*, vol. 42, pp. 1042–1066, 1994.

[18] G. B. Dantzig, "Discrete variable extremum problems," *Operations Research*, vol. 5, pp. 266–277, 1957.

[19] NVIDIA, "Cuda 3.2 programming guide," http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf, 2011.

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

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