Dense Dynamic Programming on Multi GPU

Vincent Boyer, Didier El Baz, Moussa Elkihel

Dense Dynamic Programming on Multi GPU

Vincent Boyer, Didier El Baz, Moussa Elkihel
CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
Universit de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse France
Email: vboyer@laas.fr elbaz@laas.fr elkihel@laas.fr

***Abstract*—The implementation via CUDA of a hybrid dense dynamic programming method for knapsack problems on a multi-GPU architecture is considered. Tests are carried out on a Bull cluster with Tesla S1070 computing systems. A first series of computational results shows substantial speedup close to 30 with two GPUs.**

***Keywords*-hybrid computing, multi GPU architectures, CUDA, dynamic programming, knapsack problems, combinatorial optimization.**

I. Introduction

Recently, tools like Compute Unified Device Architecture (CUDA) or Open Computing Language (OpenCL) have been developed in order to use Graphic Processing Unit (GPU) for general purpose computing. This has led to GPU computing and hybrid computing.

GPUs are high-performance many-core processors. CUDA-based NVIDIA GPUs are Single Instruction Multiple Thread (SIMT) architectures which is akin to Single Instruction Multiple Data (SIMD) architecture (see [1]).

In this paper, we concentrate on the exact solution via parallel dynamic programming of an important class of integer programming problems, i.e. Knapsack Problems (KP) on multi GPU architectures.

Knapsack problems occur in many domains like logistics, manufacturing, finance and telecommunications. Knapsack problems occur often as subproblems of hard problems in combinatorial optimization like multidimensional knapsack problems.

The parallel implementation of dynamic programming is an important topic since dynamic programming can be combined with other methods like branch and bound in order to produce efficient algorithms for solving problems of the KP family.

Several parallel dynamic programming algorithms have been proposed for KP (e.g. see [2], [3] and [4]). In particular, implementations on a SIMD machine have been performed on a 4K processor ICL DAP [5], a 16K Connection Machine CM-2 (see [6] and [7]) and a 4K MasPar MP-1 machine (see [7]).

In [8], we have presented an original implementation via CUDA of the dense 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 card have shown substantial speedup.

In this paper, we propose an original solution based on multithreading in order to implement via CUDA 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.

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 the problem and the memory occupancy.

We refer to [8] for aspects related to memory occupancy and data compression techniques. Reference is also made to [9], for a study on local search methods and GPU computing for combinatorial optimization problems.

The knapsack problem and the dense dynamic programming method are presented in Section 2. Section 3 deals with the implementation via CUDA of the dense dynamic programming method on a multi GPU system. Computational results are displayed and analyzed in Section 4. Section 5 deals with conclusions and future work.

II. Knapsack problem

The knapsack problem is among the most studied discrete optimization problems; KP is also one of the simplest prototypes of integer linear programming problems. The knapsack problem arises as a sub-problem of many complex problems (see for example [10] - [13]).

A. 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 integer programming problem:

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

To avoid any trivial solution, we assume that:

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

B. Solution via Dynamic programming

Bellman's dynamic programming method, e.g. see [14], was the first exact method to be proposed in order to solve KP. It consists in computing at each step $k \in \{1, ..., n\}$, the values of $f_k(\hat{c})$, $\hat{c} \in \{0, ..., C\}$, via the classical dynamic programming recursion:

$$f_k(\hat{c}) = \begin{cases} f_{k-1}(\hat{c}), \ \hat{c} \leq w_k - 1, \\ \max\left\{f_{k-1}(\hat{c}), \ f_{k-1}(\hat{c} - w_k) + p_k\right\}, \hat{c} \geq w_k, \end{cases} \tag{2}$$

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

The algorithm, presented in this section, is based on Bellman's recursion (2). A state corresponds to a feasible solution associated with the value of $f_k(\hat{c})$. Toth [15] has proposed an efficient recursive procedure in order to compute the states; he has used the following rule in order to eliminate states:

Proposition 1: (see [15]) If a state defined at the $k-th$ step 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 thus can be eliminated.

The dynamic programming procedure of Toth is described in Algorithm 1. The time and space complexities of the method are $\mathcal{O}(n.C)$.

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

III. Hybrid computing

A. 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. The proposed parallel dynamic programming method has been implemented via CUDA 2.3.

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

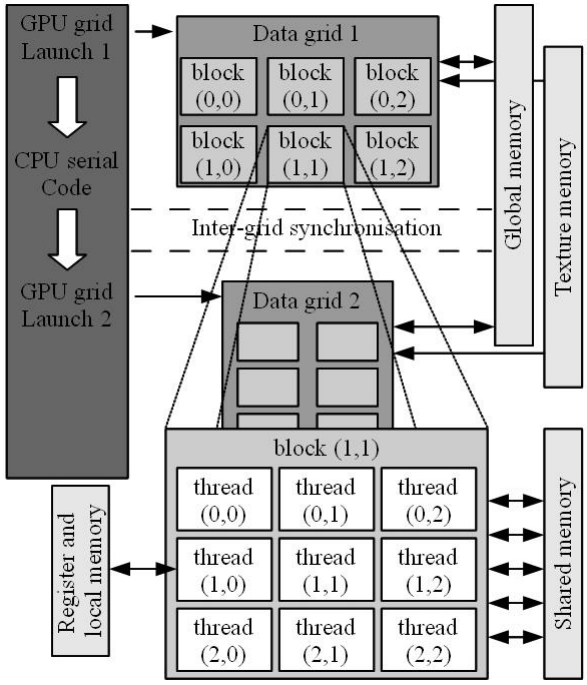

Figure 1. Thread and memory hierarchy in GPUs

As shown in Figure 1, 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 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 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 1). 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 [1] for further details on the NVIDIA GPU cards and computing systems architecture and how to optimize the code.

In [8], we have studied the parallel implementation via CUDA of the dense dynamic programming method on a CPU/GPU system with a 3 GHz Xeon Quadro Intel processor and a GTX 260 GPU card.

In this paper, we study the parallel implementation of the dense dynamic programming method on a multi GPU architecture, i.e. the Bull Iblis cluster at CINES, Montpellier, France.

The Bull Iblis cluster (see Figure 2) consists of 44 nodes with 8 cores (see [16]). Some nodes are connected to a half Tesla S1070 computing system, i.e. they are connected to 2 GPUs, since the Tesla S1070 computing system consists of 4 GPUs with a total of 960 streaming processor cores, i.e. 240 cores per GPU, see Figure 3. The frequency of streaming processors core is around 1.44 GHz. The Tesla S1070 computing system is a four Teraflop system. A total of 12 Tesla S1070 computing systems are available on the Bull Iblis cluster.

Figure 2. Bull Iblis cluster

Figure 3. NVIDIA Tesla S1070 computing system

B. Parallel Algorithm

Principle

We denote by m the number of avalaible GPUs. In the dynamic programming method, the activity that consumes the major part of processing time is the loop that processes

the values of $f(\hat{c})$, $\hat{c} \in \{0, ..., C\}$. This step is parallelized on m GPUs. The total work, i.e. the computation of the different values of f, is decomposed so that each GPU computes a subset of values of f. The computation of a subset of values of f corresponds to the task of a particular kernel.

The proposed hybrid computing approach is based on the concurrent implementation of m kernels. Each kernel is managed via a CPU thread. Kernels create GPU threads for each value of f to be computed. The main benefit of this approach is to maintain the context of each CPU thread all along the application, i.e. CPU threads are not killed at the end of each step of the parallel dynamic programming method. As a consequence, communications are minimized.

Load balancing

A load balancing procedure is implemented in order to maintain efficiency of the parallel algorithm.

At the kth step of the parallel dynamic programming method, with $k \in \{1, ..., n\}$, the ith GPU, $i \in \{1, ..., m\}$, works with the values of $\hat{c} \in [c\_min_{i,k}, ..., c\_max_{i,k}[$ where $c\_min_{i,k}$ and $c\_max_{i,k}$, respectively are computed as follows, respectively.

$$c\_min_{i,k} = \underline{c}_k + (i-1).dT_k,$$

$$c\_max_{i,k} = min\{\underline{c}_k + i.dT_k, C+1\},$$

where

$$\underline{c}_k = max\{C - \sum_{i=k+1}^{n} w_i, w_k\},$$

and

$$dT_k = \left\lceil \frac{C+1-\underline{c}_k}{m} \right\rceil.$$

Since the values of $c\_min_{i,k}$ and $c\_max_{i,k}$ may change at each step of the parallel dynamic programming method, GPUs must exchange data. The solution we propose consists in performing data exchange via the CPU, i.e. GPU will write and read data in the CPU memory.

At step $k+1$ the ith GPU computes all the values of $f(\hat{c})$ with $\hat{c} \in [c\_min_{i,k+1}, c\_max_{i,k+1}[$.

It is useless to communicate all the data at each step since the ith GPU already get the values of $f(\hat{c})$ with $\hat{c} \in [c\_min_{i,k}, c\_max_{i,k}[$. Thus, only the missing values of $f(\hat{c})$ need to be exchanged at each step of the parallel dynamic programming method.

The assignation of data to GPUs resulting from the load balancing procedure is displayed in Figure 4.

Thread processing

The procedures implemented on the CPU and the GPU, respectively, are described in Algorithms 2 and 3, respectively.

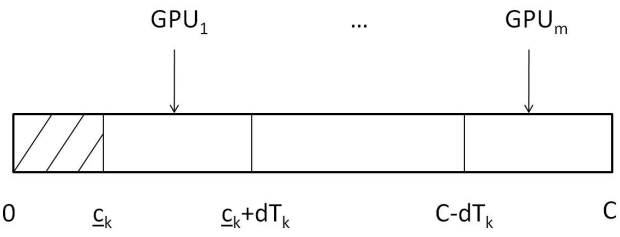

Figure 4. Allocation of data to GPUs at each step of the parallel dynamic programming algorithm with load balancing

Algorithm 2 (ith processing thread on CPU):
Variables stored in the device (GPU):
for $\hat{c} \in \{0, ..., C\}$ do
$f\_input\_d(\hat{c}) := 0$ and $f\_output\_d(\hat{c}) := 0$,
end for
$sumW := \sum_{j=1}^{n} w_j$,
for k from 1 to n do
$sumW := sumW - w_k$,
$\underline{c}_k := max\{C - sumW, w_k\}$,
Comp_c_min_and_c_max($\underline{c}_k$,i)
Synchronize()
GPU_to_CPU_com(),
Synchronize()
CPU_to_GPU_com(),
Synchronize()
Comp_f_on_device(f_input_d,f_output_d,$c\_min_{i,k}$, $c\_max_{i,k}$),
Synchronize()
f_input_d := f_output_d,
end for.

return $f\_input\_d(C)$,

Threads are launched in the GPU via the function: *Comp_f_on_device(f_input_d,f_output_d,c_min,c_max)* where:

- f_input_d are the values of f processed at the previous step,
- f_output_d are the output values of f,
- c_min denotes the minimum value of $\hat{c}$,
- c_max denotes the maximum value of $\hat{c}$.

The kernel $Compute\_f\_on\_device$ creates $c\_max - c\_min$ threads for the GPU and groups them into blocks of 512 threads (the maximum size of a block of one dimension), i.e. $\lceil (c\_max - c\_min)/512 \rceil$ blocks. All threads carry out on the GPU the procedure described in Algorithm 3.

The function $Synchronize()$ performs a global synchronization of all CPU threads in order to insure data consistency.

Data exchanges between GPUs during the load balancing

phase are made via the following two functions:

- $GPU\_to\_CPU\_com()$ where GPU writes in the CPU the values of $f(\hat{c})$ needed by its neighbors,
- $CPU\_to\_GPU\_com()$ where GPU reads in the CPU the missing values of $f(\hat{c})$ computed by its neighbors.

Algorithm 3 (threads 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 ($k \in \{1, ..., n\}$)

$\hat{c}$:=blocks_id * 512 + thread_id,
if $\hat{c} < c\_min$ or $\hat{c} > c\_max$ then STOP end if,
if $f\_input\_d(\hat{c}) < f\_input\_d(\hat{c} - w_k) + p_k$ then
$f\_output\_d(\hat{c}) := f\_input\_d(\hat{c} - w_k) + p_k$,
else
$f\_output\_d(\hat{c}) := f\_input\_d(\hat{c})$,
end if

In Algorithm 3, threads have access to the values of $input\_f(\hat{c} - w_k)$, this may result in un-coalesced memory accesses as described in section III-A.

IV. Computational results

We have used 64 bits Xeon 3 GHz Intel processors with 1 Gb memory together with Tesla S1070 computing system of the Bull Iblis cluster at CINES, Montpellier France.

We have used CUDA 2.3 for the parallel code and gcc for the serial one.

We have carried out computational tests on randomly generated correlated problems. The problems considered are available at [17]. They present the following features:

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

Table I
SPEEDUP WITH ONE AND TWO GPUS

size of the problem	1 GPU	2 GPUs
10,000	12.86	9.39
20,000	13.75	20.54
30,000	13.96	23.56
40,000	14.42	25.66
50,000	14.56	26.41
60,000	14.40	27.39
70,000	14.82	28.05
80,000	14.64	27.89
90,000	14.78	27.99

For each problem, we display the average results obtained for 10 instances.

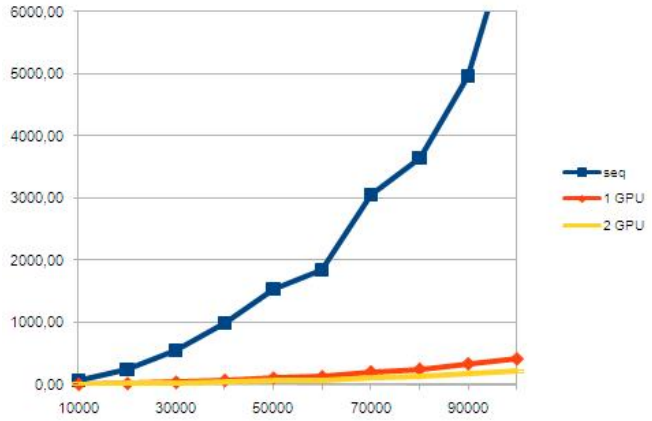

Figure 5. Computing time (s) of sequential and parallel algorithms with one and two GPUs in function of the size of the problem

We note that dense dynamic programming is well known to be suited to correlated instances. Dense dynamic programming is also not sensitive to the type of correlation.

Figure 5 presents the average processing time to solve KP obtained with the sequential and parallel algorithms on one and two GPUs in function of the size n of the problem. Table I displays the resulting speedup.

In the sequential case, we note that the computing time exceeds two hours for problems with n = 100,000 variables.

The proposed parallel dynamic programming algorithm permits one to reduce drastically the processing time. The more streaming processor cores of the Tesla S1070 computing system are made available for a given computation, the more threads are executed in parallel and the better is the global performance. As a consequence, the use of several GPU permits one to solve in reasonnable time problems that would be time consuming otherwise.

In general, the speedup increases with the size of the problem. The speedup meets a level around 14.7 with one GPU and 27.95 with two GPUs. Globally, the use of two GPUs leads to a very small lost of efficiency. This shows the interest of the proposed approach.

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

V. Conclusions and future work

In this paper, we have proposed an original parallel implementation via CUDA of the dense dynamic programming method for knapsack problems on a multi GPU system.

The proposed approach is well suited to the case where a CPU is connected to several GPUs; it is also very efficient. In particular, computational results have shown that large size problems can be solved within small processing time. This work shows the relevance of using multi GPU systems for solving difficult combinatorial optimization problems.

The presented approach seems also to be robust since the results remain good when the size of the problem increases: the speedup which is approximatively equal to 14 with one GPU and 28 with two GPUs remains stable for instances with more than 40,000 variables.

We believe that further speedup can be obtained on multi GPU clusters with extensive use of texture memory in GPUs and by improving data compression techniques, just as in [8]. Experiments have also to be performed with more GPUs.

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

We are also investigating a solution that combines GPU computing with the environment P2PDC for high performance peer to peer computing that we have developped at LAAS-CNRS (see [18] and [19]). The environment P2PDC is based on the self adaptive communication protocol P2PSAP dedicated to high performance distributed computing. Unlike different approaches in the literature that are dedicated to applications where tasks are independent and direct communication between machines is not needed, P2PDC allows frequent direct communication between peers. The environment P2PDC is devoted to scientific computing applications like combinatorial optimization and numerical simulation. This research is made in the framework of an NVIDIA Academic Partnership (see [20]).

ACKNOWLEDGMENT

Part of this study has been made possible with back up of CINES, Montpellier, France.

Dr Didier El Baz would also like 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] 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.

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

[4] 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.

[5] 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.

[6] 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.

[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, and M. Elkihel, "Solving knapsack problems on gpu," *LAAS report 10009, submitted for publication*, 2010.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] R. Bellman, "Dynamic programming," Princeton University Press, 1957.

[15] P. Toth, "Dynamic programming algorithm for the zero-one knapsack problem," *Computing*, vol. 25, pp. 29–45, 1980.

[16] http://www.cines.fr/spip.php?article504, "Bull Iblis cluster, Cines Montpellier France."

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

[18] T. T. Nguyen, D. El Baz, P. Spitéri, J. Jourjon, and M. Chau, "High performance peer-to-peer distributed computing with application to obstacle problem," in Proceedings of IEEE IPDPS, Workshop HOTP2P, 2010, atlanta, USA.

[19] http://spiderman-2.laas.fr/CIS-CIP, "ANR Project CIP, ANR-07-CIS7-011."

[20] http://projects.laas.fr/NVIDIAacademic, "Dr. Didier El Baz NVIDIA Academic Partnership, 2010."