J Supercomput (2013) 65:1125–1136
DOI 10.1007/s11227-013-0895-x

Skeletal based programming for dynamic programming on MultiGPU systems

Alejandro Acosta · Francisco Almeida

Published online: 7 March 2013

Abstract Current parallel systems composed of mixed multi/manycore systems and/with GPUs become more complex due to their heterogeneous nature. The programmability barrier inherent to parallel systems increases almost with each new architecture delivery. The development of libraries, languages, and tools that allow an easy and efficient use in this new scenario is mandatory. Among the proposals found to broach this problem, skeletal programming appeared as a natural alternative to easy the programmability of parallel systems in general, but also the GPU programming in particular. In this paper, we develop a programming skeleton for Dynamic Programming on MultiGPU systems. The skeleton, implemented in CUDA, allows the user to execute parallel codes for MultiGPU just by providing sequential C++ specifications of her problems. The performance and easy of use of this skeleton has been tested on several optimization problems. The experimental results obtained over a cluster of Nvidia Fermi prove the advantages of the approach.

Keywords Skeleton · MultiGPU · Dynamic programming

1 Introduction

Today's generation of computers is based on an architecture with identical multiple processing units consisting of several cores (multicores). The number of cores per processor is expected to increase every year. It is also well known that the current generation of compilers is incapable of automatically exploiting the ability this architecture affords applications.

A. Acosta (✉) · F. Almeida
Department Statistics and Computer Science, La Laguna University, La Laguna, Spain
e-mail: aacostad@ull.com

F. Almeida
e-mail: falmeida@ull.com

The situation is further complicated by the fact that current architectures are heterogeneous by nature, which offers the possibility of combining these multicore with GPU-based systems, for example, in a general purpose architecture. The programmability of these systems, however, poses a barrier that hampers efficient access to its exploitation.

Many proposals exist to address this problem. Some are based on carrying out a source-to-source transformation from sequential to parallel code, or on transforming parallel code designed for one architecture into parallel code designed for another [1–6]. Many of these are oriented toward different application domains. Yet another approach is based on skeletons, in which a set of parallel standards is available that allows the developer to write sequential code and obtain parallel code [7–13]. Worth noting as well the development of frameworks devoted to building source-to-source translators [14–17].

Skeletal programming for GPUs on domain specific applications have been provided by several authors, Patus [18] for stencil computations or Delite [19] for machine learning problems among others. More general approaches have been presented by SkelCL [20] or SkePU [21] to make easier the programming of GPU architectures.

In this paper, also we propose the use of skeletal based programming to exploit GPUs. An advantage of the paradigm is that the user provides sequential specifications of her problem and the skeleton implements the parallelization of the algorithm to solve it. We instantiate the method over the dynamic programming technique, one that is frequently applied to many areas of research such as control theory, biology, and so forth [22–25].

In [26], we proposed the use of DPSKEL skeletons to offset the dearth of general software dynamic programming (sequential and parallel) tools. Our aim was to bridge the obvious gap existing between general methods and DP applications. The goal of DPSKEL is to minimize the user effort required to work with the tool by conforming as much as possible to the use of standard methodologies. In this paper, we have expanded the original version of DPSKEL to adapt it to new architectures. On this occasion, we developed the solution engine for GPU architectures using CUDA. The proposed implementation shows several advantages; it allows the easy development and fast prototyping of dynamic programming problems on GPUs since it hides the parallel traversing of the dynamic programming table and also hides the difficulty of CUDA programming. Another advantage is that the skeleton can be adapted to changes in the architecture and to the programming interface without altering the dynamic programming code for the specific problem provided by the user. As a proof of the easy of use of our tool, four combinatorial optimization problems have been instantiated: the 0/1 Knapsack problem, the resource allocation problem, the triangulation of convex polygons problem and the guillotine cutting stock problem. Computational results have been provided for all test problems and a comparative analysis of the performance in a cluster of GPUs.

The paper is structured as follows. We present the MultiGPU skeleton developed and its software architecture in Sect. 2, and Sect. 3 describes the expansive computational experiment undertaken as a result of applying the method developed. The ease of development and the increase in productivity are substantial. We conclude the paper with Sect. 4, in which we outline the key findings and propose futures lines of research.

Table 1 Functional recurrence equations of dynamic programming problems

Problem	Recurrence
0/1 Knapsack (KP)	$f_{i,j} = \max\{f_{i-1,j}, f_{i-1,j-w_i} + p_i\}$
Resource Allocation (RAP)	$f_{i,j} = p_{1,j}$ if $i = 1$ and $j > 0$ $f_{i,j} = \max_{0 \le k < j}\{f_{i-1,j-k} + p_{i,k}\}$ if $(i > 1)$ and $(j > 0)$
Triangulation Convex Polygons (TCP)	$f_{i,j} = \cos t_i \cdot \cos t_{i+1} \cdot \cos t_{i+2}$ if $(i = (j-2))$ $f_{i,j} = \min_{i<k<j}\{f_{i,k} + f_{k,j} + (\cos t_i \cdot \cos t_k \cdot \cos t_j)\}$
Guillotine Cut (GCP)	$f_{i,j} = \max \begin{cases} \max_{0 \le k < object}\{profit_k\} \\ \max_{0 \le z \le i/2}\{f_{z,j} + f_{i-z,j}\} \\ \max_{0 \le y \le j/2}\{f_{i,y} + f_{i,j-y}\} \end{cases}$

2 A MultiGPU skeleton for dynamic programming

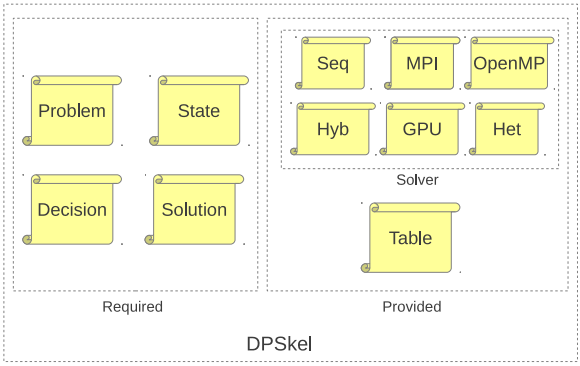
Dynamic Programming (DP) is an important technique that has been widely used to solve problems in various fields like control theory, operations research, economy, biology and computer science [22–25]. DP arrives at the optimal solution to a problem by means of an optimal sequence of decisions that rely on the principle of optimality ("given an optimal sequence of decisions, each subsequence must also be optimal") [27–30]. The principle of optimality is formally expressed as a functional recurrence equation that is established for each problem. Table 1 shows the functional recurrence equations of some dynamic programming problems. The functional equation usually imposes the evaluation of values in a space of solutions that are stored in a table (the dynamic programming table), where the optimal values and the decisions associated with those values are stored. The main obstacle to the parallelization of this technique stems from the dependencies imposed by the functional equation, which varies with each problem and imposes a certain structure in the evaluation progress.

Most of the parallelizations presented for DP correspond to specific problems (see, for example, [31, 32]). A general parallel approach was presented in [33] as an extension of the work in [29], but the considerable theoretical effort that must be overcome in certain cases complicates its use as a model for developing parallel tools. In this regard, we conclude that generic parallel approaches to DP are limited to certain classes of problem.

Analyzing the software approaches to DP, we find a group of general libraries for combinatory optimization problems [34, 35]. These are used to provide interfaces for sequential and parallel executions, but in most cases, dynamic programming is not considered at all. There are specific sequential dynamic programming libraries in [36]. There are also interesting software approaches derived from laboratories that apply tools such as LINGO [37] to dynamic programming problems using specific methods. DPSKEL is presented in [26]. This is a parallel skeleton where a large variety of optimal specifications are defined for dynamic programming problems in various architectures. The end user implements the empty sections in sequential C++ code, and the parallelism is provided automatically.

The goal of DPSKEL is to minimize the user effort required to work with the tool by conforming as much as possible to the use of standard methodologies. In

Fig. 1 DPSkel classes model



this paper, we have expanded the original version of DPSKEL to adapt it to new architectures. On this occasion, we developed the solution engines for single-GPU and MultiGPU architectures. For the MultiGPUs case, we considered systems based on distributed memory architectures where each GPU is managed by a MPI process. This approach is designed for systems composed of multiples nodes in distribute memory, where each node has one GPU. Distribute memory systems with more than one GPU per node or shared memory systems with MultiGPUs will be considered in future skeleton implementations.

Developing software skeletons for DP implies analyzing the technique and determining those elements that can be extracted from a specific case and whose elements depend on the application. Assuming that the user is capable of obtaining the functional equations herself, in DPSKEL the user provides the structure of a state and its evaluation through the functional equations, and the DP table is abstracted as a state table. DPSKEL provides the table and several methods for accessing it during the state evaluation process. These methods allow for different traversing (by rows, columns, diagonally), with the user choosing the best one based on the dependencies of the functional equations. In the sequential case, the traversing chosen for the table indicates that the states of the row (column or diagonal, respectively) will be processed sequentially, while in the single-GPU case, a set of rows (columns or diagonals, respectively) will be assigned to a set of threads to be processed simultaneously. For the MultiGPU execution, we use MPI to distribute workload among processes. Each process is responsible for one GPU. This approach allows us to introduce any of the algorithm parallelization strategies devised for DP.

DPSKEL adheres to the classes model described in Fig. 1. The concepts of State and Decision are abstracted to the user in C++ classes (Required). The user describes the problem, solution, and methods for evaluating the state (the functional equation) and for obtaining the optimal solution. DPSKEL provides the classes (Provided) for assigning and evaluating the DP table, making available the methods needed to yield the solution. The solver class provides solution engines for different platforms. This class is responsible for traversing the DP table and evaluate all states defined by the user. The parallel traversing implementation or architectures details are hidden to the user. The initial versions featured solution engines for managing the sequential and parallel executions on shared and distributed platforms. In this paper, we have

developed the engines for MultiGPU systems based in CUDA and MPI. We will now present some basic classes in DPSKEL and its adaptation to GPU architecture.

2.1 The State class

The State class holds the information associated with a DP state. This class stores and calculates the optimal value in the state and the decision associated with the optimal evaluation. The evaluation of a state implies accessing information on other states in the DP table. DPSKEL provides an object of the `Table` class hidden in each instance of the `Solver` class.

Listing 1 Definition of the `State` class. Implementation of the Evalua method of the State class for the 0/1 knapsack problem

```
1  void State::Evalua(int stage, int index) {
2     Decision dec;
3     int val;
4     if (index < w[stage]) {
5        val = 0;
6        dec.setDecision(0);
7     }
8     else if (stage == 0) {
9        val = (p[stage]);
10       dec.setDecision(1);
11    }
12    else {
13       val = max(table->getState((stage-1),index),0,
14          (table->getState(stage-1,index-w[stage])+p[stage]),1,dec);
15    }
16    setValue(val);
17    setDecision(dec);
18    if ((stage == sol->getRowSol()) && (index == sol->getColSol()))
19       sol->setSolucion(this);
20 }
```

The code shown in Listing 1 defines the state class for the knapsack problem. A problem (`pbm`), a solution (`sol`), a decision (`d`), and the DP table (`table`) are defined. These variables can be regarded as generic variables, since they must be present in any problem to be solved. The `value` variable stores the optimal profit. We should mention a particular method in this class, the `Evalua` method, which implements the functional equation. The `Evalua` function receives the indices for a state from the DP table. Any of the recurrences in Table 1 can be expressed with this prototype. If the functional equation for a specific problem requires a different prototype, the skeleton is open to method overloading using the polymorphism present in C++.

For implementation in GPU, the attribute `__host__ __device__` is added in the header to allow the methods of this class to be executed both in the GPU and in the host system (Listing 2).

2.2 The Table class

The `Table` class holds the set of `States` that configure the problem. Each entry in the DP table stores all of the information associated with a state. It holds methods

to get (`getState(i,j)`) and put (`putState(i,j)`) states from the table. The class `Solver` takes charge of building the table at the beginning of the execution. Listing 3 shows the method to allocate the memory for the dynamic programming table and to initialize the states involved in it. For implementation in GPU we used pinned memory to create the table, which provides page-locked memory that provides higher transfer throughput between CPU and GPU memory. For access to states in the table, we used the Unified Virtual Addressing (UVA) provided by CUDA that is automating active when using pinned memory.

Listing 2 Header of the `State` class

```
requires class State{
   int _value;
   Decision decision;
   Problem* pbm;
   Solution* sol;
   Table* table;
public:
   __host__ __device__ void init(Problem* pbm,Solution* sol,Table* table);
   __host__ __device__ void Evalua(int stage, int index);
   ...
};
```

2.3 The Solver class

The solver class provides solution engines for different platforms. This class contains the data structures and methods needed to carry out a DP execution in keeping with the specifications. In practice, this is a virtual class, with the solver classes provided being defined as a sub-class of this main class. To the already known solution algorithms in DPSKEL:

Listing 3 Header of the `State` class

```
void Table::init(const Setup &setup, Problem* pbm, Solution* sol){
   NumStages = setup.getNumStages();
   NumStates = setup.getNumStates();
   cudaMallocHost(&cTABLE, Num_Stages*Num_States*sizeof(State));
   for(int i=0; i<Num_Stages*Num_States; i++)
      cTABLE[i].init(pbm,sol,this);
}
```

- Solver_Seq. A sequential solver.
- Solver_OpenMP. A solver for shared-memory systems.
- Solver_MPI. A solver for distributed-memory systems.
- Solver_Hybrid. A solver for hybrid distributed and shared memory systems.
- Solver_Heterogen. A solver for heterogeneous environments.

in the current design, we added two new solvers for the GPU:

- Solver_GPU. A solver for single GPU systems.
- Solver_MultiGPU_MPI. A solver for MultiGPU in distributed-memory systems.

The Solver class contains the CUDA kernels definitions and the methods for traversing the DP table. Listing 4 shows how the DP table is accessed by the runByRows method of the Solver_cuda class. First, we distribute the workload to be allocated on each MPI process. Next, the parallel traversing of the table is performed calling the method (`kernelRunByRows`, see Listing 5). This method obtains the number of GPU threads to solve the problem and calls the CUDA kernel (`kernelRunByRows` see Listing 6). Note that we use a one-dimensional kernel; this is due to most of the dynamic programming recurrences involve data dependencies that impose traversing of the dynamic programming table. This fact prevents the use of bidimensional grids where threads should be synchronized to avoid race conditions. The kernel `kernelRunByRows` takes care of calling the methods to evaluate a row that has been provided by the end user. Each one of the threads evaluates a `state` using the Evalua method supplied by the user. Several kernels implementing different traversing modes have been implemented:

- KernelrunByRows. Traverses the DP table by rows, one thread per column.
- KernelrunByDiag. Traverses the DP table by diagonally, starting from the main diagonal upward.
- KernelrunByDiag2. Traverses the DP table by diagonally, downward until the secondary diagonal.

Listing 4 Header of the `State` class

```
1  void Solver_MultiGPU_MPI::runByRows() {
2     // Initialized field
3     // Distribute workload across multiple GPUs
4     for (i = 0; i < setup.get_Num_Stages(); i++) {
5        // Call Kernel
6        kernelRunByRows(i, vec_pos[myid]/tam_cas, vec_pos[myid+1]/tam_cas,
7                        table, numBlock[myid]);
8        // Pack States
9        for (j = vec_pos[myid]; j < vec_pos[myid+1]; j+=tam_cas)
10          op << *(table->GET_STATE(i, j/tam_cas));
11       // Distribute States
12       MPI_Allgatherv(&op.tabla[vec_pos[myid]], vec_tam[myid], MPI_CHAR,
13                      ip.tabla, vec_tam, vec_pos, MPI_CHAR, MPI_COMM_WORLD);
14       // Unpack States
15       for (j = 0; j < last_state; j++)
16          ip >> *(table->GET_STATE(i, j));
17    }
18 }
```

To distribute the results to all MPI processes, each process packs the states calculated. Next, we use a collective operation to share the results among all processes. Finally, we unpack the states (see Listing 4).

As usual in skeletons, the proposed implementation shows several advantages, the parallelism is hidden, and allows the end user to express her problem as a sequential code, moreover it also hides the complexity of the CUDA programming. The user just implements her problem using sequential C++ and adds the headers that enable the methods to be used by the GPU. Another important advantage is that new changes in the architecture and in the programming interface, can be faces by adapting the skeleton without introducing any change in the code provided by the end user.

Listing 5 Header of the `State` class

```
void Solver_MultiGPU_MPI::kernelRunByRows(int i, int init, int end,
                                          Table *table, int numBlock) {
    dim3 dimGrid(numBlock,1,1);
    dim3 dimBlock(NTHREAD,1,1);
    kernelDefRun<<<dimGrid,dimBlock>>>(i,init,end,table);
    cudaThreadSynchronize();
}
```

Listing 6 Header of the `State` class

```
__global__ void kernelDefRunByRows(int i,int init,int end,Table *table){
    int myid = (blockIdx.x*blockDim.x+threadIdx.x) + init;
    if (myid < end)
       table->GET_STATE(i, myid)->Evalua(i, myid);
}
```

3 Computational results

In order to validate the skeleton developed, we tested them on several dynamic programming problems (Table 1). We must remember that the data dependencies are different in most of the formulas considered. This means that we have to use different parallel traversing in the DP table. RAP and KP access the table by rows, while TCP and CGP access it diagonally, TCP starts from the main diagonal upward, and GCP moves diagonally downward traversing all diagonals of the table.

The parallel platform used to execute the MultiGPU skeleton is a distributed-memory system with three different nodes in message passing. One of them has four Intel Xeon E5-2660 processors with a Nvidia C2090 (512 CUDA cores) attached. The other nodes have two Intel Xeon E5649 and a Nvidia C2075 (448 CUDA cores) each one. All nodes are interconnected using a gigabit ethernet network. We compare the MultiGPU skeleton with the single-GPU skeleton. To simplify the experiment, the tests were carried out using square matrices of order 1000, 2000, 5000, and 10000.

Table 2 shows the time in seconds for all the problems when using only one GPU. The problems were executed to compare the performance of GPUs. We can see how the KP and RAP problems have the finest granularity, while the TCP and GCP problems exhibits the coarsest granularity. In the first column (GPU 1), you can see the execution time using the fastest GPU (Nvidia C2090), the GPU 2 (Nvidia C2075) and GPU 3 (Nvidia C2075) have a similar execution times. We observe as the granularity of the problem affects the performance of the GPU, for the fine grain problems (KP and RAP), the ratio obtained for GPU 2 and GPU 3 is lower than the obtained for the coarse grain problems, while the GPU 4 provides better ratios for the fine grain problems.

Figure 2 shows the execution time for all problems proposed using single-GPU (label 1 GPU) and MultiGPU (labels 2 GPUs and 3 GPUs) skeletons. We can see how the finest granularity problems (KP and RAP in Figs. 2(a) and 2(b)) obtains their best execution time using the single-GPU skeleton. This is because the computational cost of the fine-grained problems is small and most of the time is devoted to communications, what penalizes the MultiGPU skeleton that is based on message passing. When the problems size increases, MultiGPU skeleton improves the execution time,

Table 2 Time of the problems analyzed using single-GPU skeleton

Problem	Size	GPU 1	GPU 2		GPU 3	
		time	time	speedup	time	speedup
KP	1000	0.11511	0.17526	0.66	0.17546	0.66
	2000	0.45534	0.70893	0.64	0.70407	0.65
	5000	2.05729	4.42721	0.46	4.44475	0.46
	10,000	8.55532	17.3642	0.49	17.3581	0.49
RAP	1000	0.62865	0.76991	0.82	0.77072	0.82
	2000	2.76128	3.74270	0.74	3.74270	0.74
	5000	19.5798	27.0547	0.72	27.0496	0.72
	10,000	175.488	239.343	0.73	239.981	0.73
TCP	1000	6.93643	7.64279	0.91	7.64870	0.91
	2000	55.6100	58.5435	0.95	58.6373	0.95
	5000	882.879	915.102	0.96	916.097	0.96
	10,000	7258.47	7807.39	0.93	7823.39	0.93
GCP	1000	27.6493	29.9156	0.92	29.9262	0.92
	2000	221.562	231.082	0.96	231.282	0.96
	5000	3561.29	3684.03	0.97	3732.61	0.95
	10,000	30327.1	33741.3	0.90	33686.5	0.90

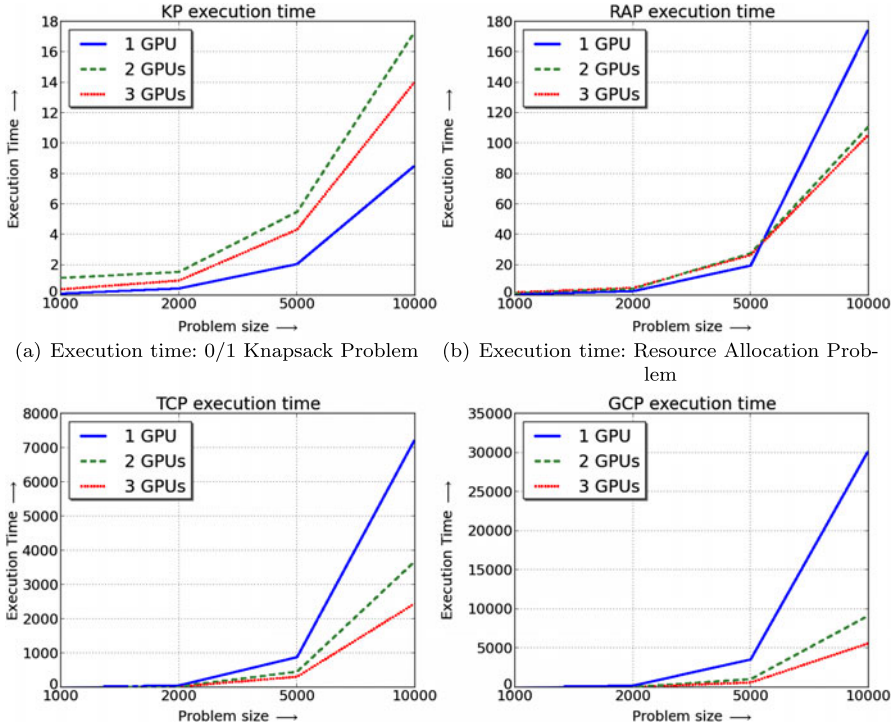
as shown in Fig. 2(b) for a problem size of 10000. For the coarsest granularity problems (TCP and GCP in Figs. 2(c) and 2(d)), the best results are obtained with the MultiGPU skeleton. For all problems, the executions time using 3 GPUs is faster than using 2 GPUs.

Figures 3 show the speedup of all problems. For MultiGPU executions, we used the fastest GPUs configuration, i.e., when using configurations with two GPUs, we used GPU 1, GPU 2. To compute the speedup, we compare the running time of the MultiGPU code with the running time obtained when using the fastest GPU (GPU 1 in Table 2).

In Figs. 3(a) and 3(b), we show the speedup of the MultiGPU skeleton compared with the single-GPU skeleton. In this case, the fine-grain problems (KP and RAP) behave differently than coarse-grain problems (TCP and GCP). KP and RAP need a larger problem size to get good performance. As discussed above, this is because the communications cost, and what penalizes the skeleton based on message passing. TCP and GCP presented good speedups in all cases. For GCP, the speedup is greater than expected presenting superlinear speedups, we assume that a better memory management is achieved when more that one GPU is used.

4 Conclusion

We developed a dynamic programming skeleton for systems equipped with one or more GPUs. The skeleton developed shows the known advantages of skeletal pro-



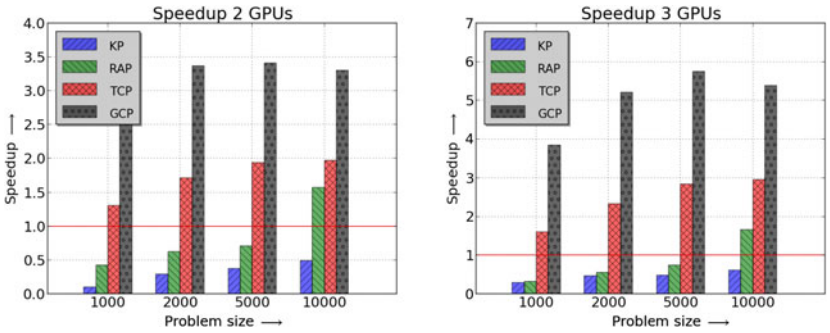
(a) Execution time: 0/1 Knapsack Problem

(b) Execution time: Resource Allocation Problem

(c) Execution time: Triangulation Convex Polygons Problem

(d) Execution time: Guillotine Cut Problem

Fig. 2 Execution time of all problems using single-GPU and MultiGPU skeletons



(a) Speedups: 2 GPUs

(b) Speedups: 3 GPUs

Fig. 3 Speedups of the MultiGPU skeleton

gramming of ease of use, programmability and efficiency while hiding the parallelism at the same time. The use of the DP skeleton avoids to the end user the learning of a new API like CUDA while keeps the code portable for many different architectures. The high productivity of the approach is tested by using four combinatorial optimiza-

tion problems. We analyze the performance of the MultiGPU skeleton by computing the speedup comparing executions that use only one GPU against MultiGPU systems. The multiple GPU system presents a good scalability in the cluster of GPUs that we have used as testing platform.We conclude that, in general, the skeletal approach can be a good alternative to broach multiple GPU systems and, in particular, its application to the dynamic programming domain has been successful. As a future line of research, we will explore the convenience of using clusters of GPUs in shared memory systems or even considering then in hybrid systems.

Acknowledgements This work has been supported by the EC (FEDER) and the Spanish MEC with the I + D + I contract number: TIN2011-24598.

References

1. Schordan M, Quinlan DJ (2003) A source-to-source architecture for user-defined optimizations. In: JMLC, pp 214–223
2. Blume W, Doallo R, Eigenmann R, Grout J, Hoeflinger J, Lawrence T, Lee J, Padua D, Paek Y, Pottenger B, Rauchwerger L, Tu P (1996) Parallel programming with Polaris. Computer 29:78–82
3. Dooley I (2006) Automated source-to-source translations to assist parallel programmers. Master's thesis, Dept of Computer Science, University of Illinois http://charm.cs.uiuc.edu/papers/DooleyMSThesis06.shtml
4. Ueng Sz, Lathara M, Baghsorkhi SS, Hwu WmW (2008) Cuda-lite: reducing CPU programming complexity. In: LCPC'08. Lecture notes in computer science, vol 5335. Springer, Berlin, pp 1–15
5. Lionetti FV, McCulloch AD, Baden SB (2010) Source-to-source optimization of cuda C for GPU accelerated cardiac cell modeling. In: Proceedings of the 16th international Euro-Par conference on parallel processing: part I (EuroPar'10). Springer, Berlin, pp 38–49
6. Par4All. www.par4all.org
7. Cole MI (1988) Algorithmic skeletons: a structured approach to the management of parallel computation. PhD thesis. AAID-85022
8. Bischof H, Gorlatch S (2002) Double-scan: introducing and implementing a new data-parallel skeleton. In: Proceedings of the 8th international Euro-Par conference on parallel processing (Euro-Par '02). Springer, London, pp 640–647
9. Darlington J, Field AJ, Harrison PG, Kelly PHJ, Sharp DWN, Wu Q, While RL (1993) Parallel programming using skeleton functions. Springer, Berlin
10. Cole M (2004) Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput 30:389–406
11. Benoit A, Cole M (2005) Two fundamental concepts in skeletal parallel programming. In: The international conference on computational science (ICCS 2005), part II. Lecture notes in computer science, vol 3515. Springer, Berlin, pp 764–771
12. Buono D, Danelutto M, Lametti S (2010) Map, reduce and mapreduce, the skeleton way. Proc Comput Sci 1(1):2095–2103
13. González-Vélez H, Leyton M (2010) A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw Pract Exp 40:1135–1160
14. ROSE. www.rosecompiler.org
15. Pai S, Govindarajan R, Thazhuthaveetil MJ (2010) Plasma: portable programming for SIMD heterogeneous accelerators
16. Benkner S, Mehofer E, Pllana S (2008) Towards an intelligent environment for programming multicore computing systems. In: Proceedings of the 2nd workshop on highly parallel processing on a chip (HPPC 2008), in conjunction with Euro-Par 2008, August 2008
17. Dave C, Bae H, Min S-J, Lee S, Eigenmann R, Midkiff SP (2009) Cetus: a source-to-source compiler infrastructure for multicores. Computer 42(11):36–42
18. Christen M, Schenk O, Burkhart H (2011) Automatic code generation and tuning for stencil kernels on modern shared memory architectures. Comput Sci 26(3-4):205–210

19. Brown KJ, Sujeeth AK, Lee HJ, Rompf T, Chafi H, Odersky M, Olukotun K (2011) A heterogeneous parallel framework for domain-specific languages. In: Proceedings of the 2011 international conference on parallel architectures and compilation techniques (PACT '11). IEEE Computer Society, Washington, pp 89–100
20. Steuwer M, Kegel P, Gorlatch S (2011) Skelcl—a portable skeleton library for high-level CPU programming. In: Proceedings of the 2011 IEEE international symposium on parallel and distributed processing workshops and PhD forum (IPDPSW '11). IEEE Computer Society, Washington, pp 1176–1182
21. Enmyren J, Kessler CW (2010) Skepu: a multi-backend skeleton programming library for multi-CPU systems. In: Proceedings of the fourth international workshop on high-level parallel programming and applications (HLPP '10). ACM, New York, pp 5–14
22. Nascimento J, Powell W (2010) Dynamic programming models and algorithms for the mutual fund cash balance problem. Manage Sci 56:801–815
23. Erdelyi A, Topaloglu H (2010) A dynamic programming decomposition method for making overbooking decisions over an airline network. INFORMS J Comput 22:443–456
24. Huang K, Liang Y-T (2011) A dynamic programming algorithm based on expected revenue approximation for the network revenue management problem. Transp Res Part E, Logist Transp Rev 47(3), 333-341
25. Shachter R, Bhattacharjya D (2010) Dynamic programming in influence diagrams with decision circuits. In: Twenty-sixth conference on uncertainty in artificial intelligence, pp 509–516
26. Peláez I, Almeida F, Suárez F (2007) Dpskel: a skeleton based tool for parallel dynamic programming. In: Seventh international conference on parallel processing and applied mathematics (PPAM2007)
27. Helman P (1989) A common schema for dynamic programming and branch and bound algorithms. J ACM 36:97–128
28. Karp RM, Held M (1967) Finite state process and dynamic programming. SIAM J Appl Math 15:693–718
29. Ibaraki T (1988) Enumerative approaches to combinatorial optimization, part II. Ann Oper Res 11:1–4
30. de Moor O (1999) Dynamic programming as a software component. In: Mastorakis N (ed) Proc 3rd WSEAS int conf circuits, systems, communications and computers
31. Andonov R, Balev S, Rajopadhye S, Yanev N (2001) Otimal semi-oblique tiling and its application to sequence comparison. In: 13th ACM symposium on parallel algorithms and architectures (SPAA)
32. Andonov R, Rajopadhye S (1997) Optimal orthogonal tiling of 2-d iterations. J Parallel Distrib Comput 45:159–165
33. Morales D, ALmeida F, Rodríguez C, Roda J, Delgado CAI (2000) Parallel dynamic programming and automata theory. Parallel Computing 26(1), 113–134
34. Eckstein J, Phillips CA, Hart WE (2000) PICO: an object-oriented framework for parallel branch and bound. Technical report, RUTCOR
35. Le Cun B (2001) Bob++ library illustrated by VRP. In: European operational research conference (EURO'2001), Rotterdam, p 157
36. Lubow BC (1997) SDP: generalized software for solving stochastic dynamic optimization problems. Wildl Soc Bull 23:738–742
37. Lohmander P Deterministic and stochastic dynamic programming. www.sekon.slu.se/PLO/diskreto/dynp.htm