GPU implementations of scheduling heuristics for heterogeneous computing environments

Article

2 authors, including:

Sergio Nesmachnow
Universidad de la República de Uruguay
202 PUBLICATIONS **1,542** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

STIC-AmSud PER-MARE - Pervasive Map Reduce View project

SAVANT: Automatic Generation of Parallel Approximation Algorithms for Low-power Architectures Based on Machine Learning View project

GPU implementations of scheduling heuristics for heterogeneous computing environments

Sergio Nesmachnow and Mauro Canabé

Centro de Cálculo, Facultad de Ingeniería
Universidad de la República, Uruguay
{mcanabe,sergion}@fing.edu.uy

Abstract. This work presents the application of parallel computing techniques using Graphic Processing Units to improve the efficiency of scheduling heuristics for heterogeneous computing systems. The experimental evaluation of the proposed methods demonstrates that a significant reduction on the computing times can be attained, allowing to tackle large scheduling scenarios in reasonable execution times.

Keywords: GPU computing, heterogeneous computing, scheduling.

1 Introduction

In the last fifteen years, distributed computing environments have been increasingly used to solve complex problems. Nowadays, a common platform for distributed computing usually comprises a heterogeneous collection of computers. This class of infrastructures includes *grid computing* and *cloud computing* environments, where a large set of heterogeneous computers with diverse characteristics are combined to provide pervasive on demand and cost-effective processing power, software, and access to data, for solving many kinds of problems [8,19].

A key problem when using such heterogeneous computing (HC) environments consists in finding a scheduling strategy for a set of tasks to be executed. The goal is to assign the computing resources by satisfying some efficiency criteria, usually related to the total execution time or resource utilization [4,14]. The *heterogeneous computing scheduling problem* (HCSP) became specially important due to the popularization of heterogeneous distributed computing systems [5,9].

Traditional scheduling problems are NP-hard [10], thus classic exact methods are only useful for solving problem instances of very reduced size. Heuristics methods are able to get efficient schedules in reasonable times, but they still require long execution times when solving large instances of the scheduling problem. These execution times (i.e., in the order of an hour) can be extremely high for performing on-line scheduling in realistic HC infrastructures.

High performance computing techniques can be applied to reduce the execution times required to perform the scheduling. The massively parallel hardware in Graphic Processor Units (GPU) has been successfully applied to speed up the computations required to solve problems in many application areas [12], showing an excellent relationship between cost and computing power [17].

The main contribution of this work is the development of parallel implementations on GPU for two classic scheduling heuristics, namely Min-Min and Sufferage [13]. The experimental evaluation of the proposed parallel methods demonstrates that a significant reduction on the computing times can be attained when using the parallel GPU hardware. This performance improvement allows solving large scheduling scenarios in reasonable execution times.

The manuscript is structured as follows. Next section introduces the HCSP mathematical formulation, and the heuristics studied in this work. A brief introduction to GPU computing is presented in Section 3. Section 4 describes the GPU implementations of the Min-Min and Sufferage heuristics. The experimental evaluation of the proposed methods is reported in Section 5, where the efficiency results are also analyzed. Finally, Section 6 summarizes the conclusions of the research and formulates the main lines for future work.

2 Heterogeneous computing scheduling

This section presents the HCSP and its mathematical formulation. It also provides a description of the class of list scheduling heuristics, and describes the Min-Min and Sufferage methods parallelized in this work.

2.1 HCSP formulation

An HC system is composed of many computers, also called *processors* or *machines*, and a set of tasks to be executed on the system. A task is the atomic unit of workload, so it cannot be divided into smaller chunks, nor interrupted after it is assigned to a machine. The execution times of any individual task vary from one machine to another, so there will be competition among tasks for using those machines able to execute them in the shortest time.

Scheduling problems mainly concern about time, trying to minimize the time spent to execute all tasks. The most usual metric to minimize in this model is the *makespan*, defined as the time spent from the moment when the first task begins execution to the moment when the last task is completed [14].

The following formulation presents the mathematical model for the HCSP aimed at minimizing the makespan:

– given an HC system composed of a set of machines $P = \{m_1, \ldots, m_M\}$ (dimension M), and a collection of tasks $T = \{t_1, \ldots, t_N\}$ (dimension N) to be executed on the system,
– let there be an *execution time function* $ET : T \times P \to \mathbf{R}^+$, where $ET(t_i, m_j)$ is the time required to execute the task t_i in the machine m_j,
– the goal of the HCSP is to find an assignment of tasks to machines (a function $f : T^N \to P^M$) which minimizes the *makespan*, defined in Equation 1.

$$\max_{m_j \in P} \sum_{\substack{t_i \in T: \\ f(t_i) = m_j}} ET(t_i, m_j) \tag{1}$$

In the previous HCSP formulation all tasks can be independently executed, disregarding the execution order. This kind of applications frequently appears in many lines of scientific research, specially in Single-Program Multiple-Data applications used for multimedia processing, data mining, parallel domain decomposition of numerical models for physical phenomena, etc. The independent tasks model also arises when different users submit their (obviously independent) tasks to execute in grid computing and volunteer-based computing infrastructures -such as TeraGrid, WLCG, Berkeley's BOINC, Xgrid, etc. [2]-, where non-dependent applications using domain decomposition are very often submitted for execution. Thus, the relevance of the HCSP version faced in this work is justified due to its significance in realistic distributed HC and grid environments.

2.2 List scheduling heuristics

The class of *list scheduling* heuristics comprises many deterministic scheduling methods that work by assigning priorities to tasks based on a particular criterion. After that, the list of tasks is sorted in decreasing priority and each task is assigned to a processor, regarding the task priority and the processor availability. Algorithm 1 presents the generic schema of a list scheduling method.

Algorithm 1 Schema of a list scheduling algorithm.

1: **while** tasks left to assign **do**
2:     determine the most suitable task according to the chosen criterion
3:     **for** each task to assign, each machine **do**
4:         evaluate criterion (task, machine)
5:     **end for**
6:     assign the selected task to the selected machine
7: **end while**

Since the pioneering work by Ibarra and Kim [11], where the first algorithms following the generic schema in Algorithm 1 were introduced, many list scheduling techniques have been proposed to provide easy methods for tasks-to-machines scheduling. This class of methods has also often been employed in hybrid algorithms, with the objective of improving the search of metaheuristic approaches for the HCSP and related scheduling problems.

The simplest list scheduling heuristics use a single criterion to perform the tasks-to-machines assignment. Among others, this category includes: *Minimum Execution Time* (MET), which considers the tasks sorted in an arbitrary order, and assigns them to the machine with lower ET for that task, regardless of the machine availability; *Opportunistic Load Balancing* (OLB), which considers the tasks sorted in an arbitrary order, and assigns them to the next machine that is expected to be available, regardless of the ET for each task on that machine; and *Minimum Completion Time* (MCT), which tries to combine the benefits of OLB and MET by considering the set of tasks sorted in an arbitrary order and assigning each task to the machine with the minimum ET for that task.

Trying to overcome the inefficacy of these simple heuristics, other methods takes into account more complex and holistic criteria to perform the task mapping. This work focuses on two of the most effective heuristics in this class:

– **Min-Min** starts with a set U of all *unmapped* tasks, calculates the MCT for each task in U for each machine, and assigns the task with the minimum overall MCT to the best machine. The mapped task is removed from U, and the process is repeated until all tasks are mapped. Min-Min improves upon MCT by considering all the unmapped tasks and by updating the machine availability for every assignment. It computes balanced schedules and allows finding smaller makespan values than other heuristics, since more tasks are expected to be assigned to the machines that can complete them the earliest.
– **Sufferage** identifies in each iteration step the task that if it is not assigned to a certain host, it will *suffer* the most. The *sufferage value* is computed as the difference between the best MCT of the task and its second-best MCT. Sufferage gives precedence to those tasks with high sufferage value, assigning them to the machines that can complete them at the earliest time.

The computational complexity of both Min-Min and Sufferage heuristics is $O(N^3)$, where N is the number of tasks to schedule. When solving large instances of the HCSP, large execution times are required to perform the task-to-machine assignment (i.e. several minutes for a problem instance with 10.000 tasks). In this context, parallel computing techniques can be applied to reduce the execution times required to find the schedules.

GPU computing has been used to parallelize many algorithms in diverse research areas. However, to the best of our knowledge, there have been no previous proposals of applying GPU parallelism to list scheduling heuristics.

3 GPU computing

GPUs were originally designed to exclusively perform the graphic processing in computers, allowing the Central Process Unit (CPU) to concentrate in the remaining computations. Nowadays, GPUs have a considerably large computing power, provided by hundreds of processing units with reasonable fast clock frequencies. In the last ten years, GPUs have been used as a powerful parallel hardware architecture to achieve efficiency in the execution of applications.

GPU programming and CUDA. Ten years ago, when GPUs were first used to perform general-purpose computation, they were programmed using low-level mechanism such as the interruption services of the BIOS, or by using graphic APIs such as OpenGL and DirectX [6]. Later, the programs for GPU were developed in assembly language for each card model, and they had very limited portability. So, high-level languages were developed to fully exploit the capabilities of the GPUs. In 2007, NVIDIA introduced CUDA [16], a software architecture for managing the GPU as a parallel computing device without requiring to map the data and the computation into a graphic API.

CUDA is based in an extension of the C language, and it is available for graphic cards GeForce 8 Series and superior. Three software layers are used in CUDA to communicate with the GPU (see Fig. 1): a low-level hardware driver that performs the CPU-GPU data communications, a high-level API, and a set of libraries such as CUBLAS for linear algebra and CUFFT for Fourier transforms.
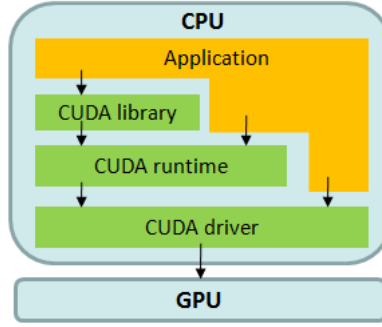

Fig. 1. CUDA architecture.

For the CUDA programmer, the GPU is a computing device which is able to execute a large number of threads in parallel. A specific procedure to be executed many times over different data can be isolated in a GPU-function using many execution threads. The function is compiled using a specific set of instructions and the resulting program is loaded in the GPU. The GPU has its own DRAM, and the data are copied from the DRAM of the GPU to the RAM of the host (and viceversa) using optimized calls to the CUDA API.

The CUDA architecture is built around a scalable array of multiprocessors, each one of them having eight scalar processors, one multithreading unit, and a shared memory chip. The multiprocessors are able to create, manage, and execute parallel threads, with reduced overhead. The threads are grouped in *blocks* (with up to 512 threads), which are executed in a single multiprocessor, and the blocks are grouped in *grids*. When a CUDA program calls a grid to be executed in the GPU, each one of the blocks in the grid is numbered and distributed to an available multiprocessor. When a multiprocessor receives a block to execute, it splits the threads in *warps*, a set of 32 consecutive threads. Each warp executes a single instruction at a time, so the best efficiency is achieved when the 32 threads in the warp executes the same instruction. Each time that a block finishes its execution, a new block is assigned to the available multiprocessor.

The threads access the data using three memory spaces: a *shared memory* used by the threads in the block; the *local memory* of the thread; and the *global memory* of the GPU. Minimizing the access to the slower memory spaces (the local memory of the thread and the global memory of the GPU) is a very important feature to achieve efficiency. On the other side, the shared memory is placed within the GPU chip, thus it provides a faster way to store the data.

4 Implementations of Min-Min and Sufferage on GPU

The GPU architecture is better suited to the Single Instruction Multiple Data execution model for parallel programs. Thus, GPUs provide an ideal platform for executing algorithms that use the domain decomposition strategy, especially when they execute the same instruction set for each element of the domain.

The generic schema for a list scheduling heuristic in Algorithm 1 shows that both Min-Min and Sufferage apply the same strategy: for each unassigned task the criteria are evaluated on all machines and the task that best meets the criteria is selected and assigned to the machine which generates the minimum MCT. Clearly, this schema is an ideal case for applying a domain decomposition to generate a parallel version of the heuristics.

The Min-Min and Sufferage implementations on GPU designed in this work perform in parallel the evaluation of the criteria proposed by each heuristic. For each unassigned task, the evaluation of the criteria for all machines is made in parallel on the GPU, building a vector that stores the identifier of the task, the best value obtained for the criteria, and the machine to get that value. The indicators in the vector are then processed in the reduction phase to obtain the best value that meets the criteria, and then the best pair (task,machine) is assigned. It is worth noting that the processing of the indicators to obtain the optimum value of each step is also performed using the GPU. A graphical schema of the parallel algorithm is presented in Fig. 2.
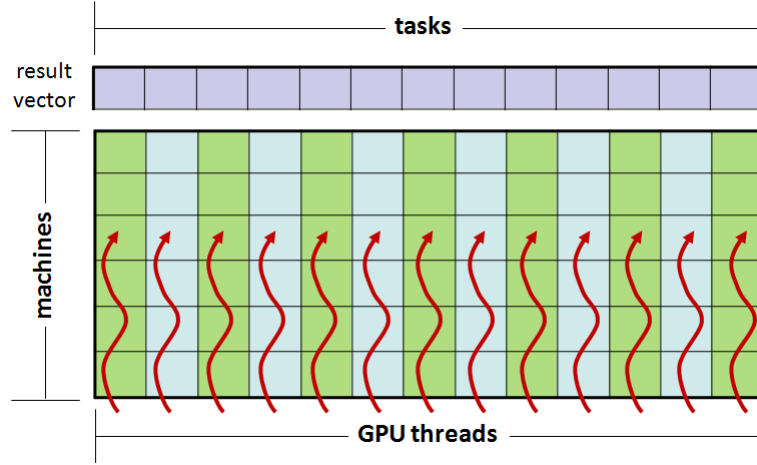

Fig. 2. Parallel strategy on GPU for Min-Min and Sufferage.

In the implementations proposed in this work, one task is assigned to each thread of the GPU. Each thread evaluates the criteria for the set of machines and stores the resulting value in a vector, the thread identifier, and the machine to which the task should be assigned. Later, the reduction phase searches the value that better satisfies the criteria.

A specific data representation was used to accelerate the execution of the sequential heuristics, in order to perform a fair comparison with the execution times of the GPU implementations: they use a data matrix where each row represents a task and each column represents a machine. Thus, when performing the processing for tasks (rows), the entries are loaded to the cache of the processing core, allowing a faster way to access the data.

For parallel algorithms running on GPU, loading the data matrix in the same way reduces the computational efficiency. Adjacent threads would access to the data stored in contiguous rows, but these are not stored contiguously, thus they cannot be stored in shared memory. When the data matrix is loaded so that each column represent a task and each row represent a machine, two adjacent threads in GPU access to the data stored in contiguous columns. These data are stored in contiguous memory locations, so they can be loaded in the shared memory, allowing to perform a faster data access for each thread, and therefore improving the execution of the parallel algorithm on GPU.

Preliminary experiments were also performed using a strategy of domain decomposition that divides the data by machines rather than by tasks, but this option was finally discarded due to scalability issues as the problem size increases.

5 Experimental analysis

5.1 HCSP scenarios

No standardized benchmarks or test suites for the HCSP have been proposed in the related literature [18]. Researchers have often used the suite of twelve instances proposed by Braun et al. [3], following the expected time to compute (ETC) performance estimation model by Ali et al. [1].

In order to study the efficiency of the GPU implementations as the problem instances grow, the experimental analysis consider a test suite of large-dimension HCSP instances, randomly generated to test the scalability of the proposed methods. This test suite was designed following the methodology by Ali et al. [1] The set includes the 96 medium-sized HCSP instances with dimension (tasks×machines) 1024×32, 2048×64, 4096×128 and 8192×256 previously solved using an evolutionary algorithm [15], and new large dimension HCSP instances with dimensions 16384×512, 32768×1024, and 65536×2048, specifically created to evaluate the GPU implementations presented in this work.

These dimensions are much larger than those of the popular benchmark by Braun et al. [3] and they better model present distributed HC and grid systems. The problem instances and the generator program are publicly available to download at `http://www.fing.edu.uy/inco/grupos/cecal/hpc/HCSP`.

5.2 Development and execution platform

The scheduling heuristics were implemented in C, using the standard `stdlib` library. The experimental analysis was performed on a Dell PowerEdge (Quad-Core Xeon E5530 at 2.4 GHz, 48 GB RAM, 8 MB cache), with CentOS Linux 5.4 and a NVidia Tesla C1060 GPU (240 cores at 1.33 GHz, 4GB RAM) [7].

5.3 Experimental results

This section reports the results obtained when applying the parallel GPU implementations of the list scheduling heuristics for each of the HSCP instances tackled in this paper.

Solution quality. Since the proposed parallel implementations do not modify the algorithmic behavior of the MinMin and Sufferage heuristics, the makespan results obtained with the GPU implementations are exactly the same than those obtained with the sequential versions for all the studied HCSP instances. The makespan values obtained for each problem scenario tackled are reported in the HCSP website `http://www.fing.edu.uy/inco/grupos/cecal/hpc/HCSP`.

Execution times. Table 1 compares the execution times (in seconds) of the sequential and parallel versions on GPU of the studied heuristics. The time results in Table 1 correspond to the average values for all the HCSP instances solved for each problem dimension studied, and the comparison is performed considering the optimized sequential algorithms using the specialized data representation described in Section 4. The speedup values in Table 1 summarize the acceleration when using the GPU implementation with respect to the sequential one, by computing the quotient between the execution times of the sequential and parallel implementations.

dimension	Min-Min			Sufferage		
	sequential	parallel	speedup	sequential	parallel	speedup
1024×32	0.07	5.35	0.01	0.11	5.41	0.02
2048×64	0.39	5.53	0.07	0.59	5.51	0.11
4096×128	2.32	6.15	0.38	3.60	6.16	0.58
8192×256	16.27	9.90	1.64	22.54	9.97	2.26
16384×512	114.06	30.06	3.79	149.61	30.34	4.93
32768×1024	815.56	183.82	4.44	1012.16	184.57	5.48
65536×2048	6249.18	1214.86	**5.14**	6825.84	1235.13	**5.53**

Table 1. Execution times and speedups for the GPU implementations.

The results in Table 1 show that significant improvements on the execution times of Min-Min and Sufferage are obtained when using the GPU implementations for problem instances with more than 8.000 tasks. When solving the low-dimension problem instances, the GPU implementations were unable to outperform the execution times of the sequential Min-Min and Sufferage, mainly due to the overhead introduced by the threads creation and management, and the use of the GPU memory. However, when solving larger problem instances that model realistic grid scenarios, significant improvements in the execution times are achieved, specially for the 65536×2048 problem instances.

Fig. 3 presents a graphical summary of the speedup values for the GPU implementations for each problem dimension faced.
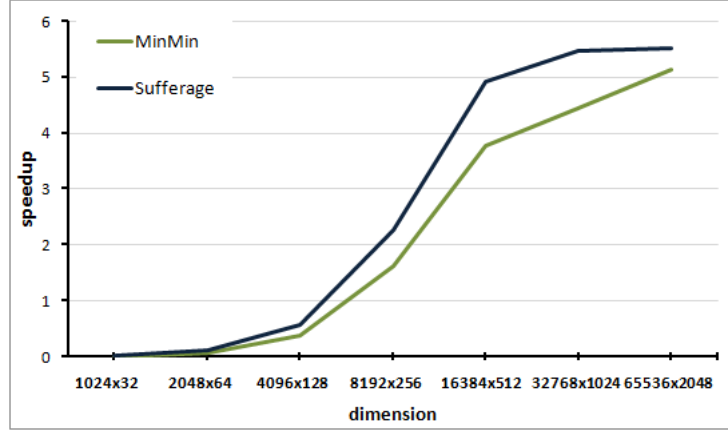
Fig. 3. Speedup of the GPU implementations.

The evolution of the speedup values in Fig. 3 indicates that reasonable accelerations are obtained for the HCSP instances with dimension 8192×256, 16384×512, 32768×1024, and 65536×2048. The best speedup values were obtained for the two largest problem dimensions, with a maximum of 5.14 for the parallel Min-Min implementation on GPU and 5.53 for the parallel Sufferage implementation on GPU. Since the Sufferage heuristic performs an additional computation to evaluate the sufferage value for each task in every iteration step, the speedup values obtained with the parallel Sufferage implementation on GPU are slightly larger than the ones obtained for the parallel Min-Min implementation on GPU.

The previously commented results indicate that the parallel implementation of list scheduling heuristics in GPU provides promising reductions in the execution times when solving large instances of the scheduling problem.

6 Conclusions and future work

This article studied the development of parallel implementations in GPU for two of the most effective list scheduling heuristics algorithms, namely Min-Min and Sufferage, for scheduling in heterogeneous computing environments.

Both algorithms were developed using CUDA, following a simple domain decomposition approach that allows scaling to solve very large dimension problem instances. The experimental evaluation solved HCSP instances with up to 65536 tasks and 2048 machines, a dimension far more larger than the previously proposed in the related literature.

The experimental results demonstrated that the parallel implementations of Min-Min and Sufferage on GPU provide significant accelerations over the time required by the sequential implementations when solving large instances of the HCSP. These parallel implementations allow tackling large scheduling scenarios in reasonable execution times.

The main line for future work is related with improving the proposed GPU implementations, mainly by studying the management of the memory accessed by the threads. In this way, the computational efficiency of the heuristics on GPU can be further improved, allowing to develop even more efficient parallel implementations. We are working in this topic right now.

References

1. S. Ali, H. Siegel, M. Maheswaran, S. Ali, and D. Hensgen. Task execution time modeling for heterogeneous computing systems. In *Proc. of the 9th Heterogeneous Computing Workshop*, page 185, Washington, USA, 2000.
2. F. Berman, G. Fox, and A. Hey. *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons, Inc., New York, NY, USA, 2003.
3. T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, 2001.
4. H. El-Rewini, T. Lewis, and H. Ali. *Task scheduling in parallel and distributed systems.* Prentice-Hall, Inc., 1994.
5. M. Eshaghian. *Heterogeneous Computing.* Artech House, 1996.
6. R. Fernando, editor. *GPU gems.* Addision-Wesley, Boston, 2004.
7. Cluster FING. Facultad de Ingeniería, Universidad de la República, Uruguay. Available online `http://www.fing.edu.uy/cluster`, 2011. Accessed on July 2011.
8. I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure.* Morgan Kaufmann Publishers, 1998.
9. R. Freund, V. Sunderam, A. Gottlieb, K. Hwang, and S. Sahni. Special issue on heterogeneous processing. *J. Parallel Distrib. Comput.*, 21(3), 1994.
10. M. Garey and D. Johnson. *Computers and intractability.* Freeman, 1979.
11. O. Ibarra and C. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24(2):280–289, 1977.
12. D. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 2010.
13. Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.
14. J. Leung, L. Kelly, and J. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* CRC Press, Inc., 2004.
15. S. Nesmachnow. A cellular multiobjective evolutionary algorithm for efficient heterogeneous computing scheduling. In *EVOLVE 2011, A bridge between Probability, Set Oriented Numerics and Evolutionary Computation*, 2011.
16. nVidia. CUDA website. Available online `http://www.nvidia.com/object/cuda_home.html`, 2010. Accessed on July 2011.
17. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
18. M. Theys, T. Braun, H. Siegel, A. Maciejewski, and Y. Kwok. Mapping tasks onto distributed heterogeneous computing systems using a genetic algorithm approach. In *Solutions to parallel and distributed computing problems*, pages 135–178, New York, USA, 2001. Wiley.
19. T. Velte, A. Velte, and R. Elsenpeter. *Cloud Computing, A Practical Approach.* McGraw-Hill, Inc., New York, NY, USA, 2010.