A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU

Tiago Carneiro, Albert Einstein Muritiba, Marcos Negreiros, G. A. L. De Campos

myjournal manuscript No.
(will be inserted by the editor)

A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU

Tiago Carneiro ·
Albert Einstein Muritiba ·
Marcos Negreiros ·
Gustavo Augusto Lima de Campos

the date of receipt and acceptance should be inserted later

Abstract This work presents a new parallel procedure designed to process combinatorial branch-and-bound (B&B) algorithms by using GPU. Our strategy is to perform B&B sequentially until a specific depth, saving the current path in the B&B tree as a node into the *Active Set*, and then force the backtracking. Each node into the *Active Set* is a DFS-B&B root that will be concurrently processed by the GPU. We compare our results with multicore and serial versions of the same search schema, using explicit enumeration (all possible solutions) and implicit enumeration (branch-and-bound search), for some asymmetrical traveling salesman problem instances. Our computational results indicate the superiority of our GPU computing-based method mainly for the B&B's worst case.

Keywords Branch-and-Bound · ATSP · CUDA · Depth-first Search · GPGPU

T. Carneiro · G. A. L. de Campos
Mestrado Acadêmico em Ciências da Computação - MACC, Universidade Estadual do Ceará, Fortaleza - CE, Brazil.
E-mail: {carneiro,gustavo}@larces.uece.br

A. E. Muritiba
Departamento de Estatística e Matemática Aplicada - DEMA, Universidade Federal do Ceará, Campus do Pici, Bloco 910, Fortaleza - CE, Brazil.
E-mail: einstein@ufc.br

M. Negreiros
Mestrado Profissional em Computação Aplicada – MPCOMP/UECE-IFCE, Universidade Estadual do Ceará, Fortaleza-CE, Brazil.
E-mail: negreiro@graphvs.com.br

Extended version of: A new parallel schema for branch-and-bound algorithms using GPGPU , In: 23*rd* International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), *p*. 41–47, (2011). ISSN 1550-6533, DOI: 10.1109/SBAC-PAD.2011.20

1 Introduction

Improvement in computer's processing capability, along years, were mainly based on rising of the processors' clock frequencies. For each generation of new processors, clock frequencies were almost twice faster than the previous generation's clock. Under the perspective of this fast performance growth, a great variety of applications have aroused, from scientific to real-time 3D applications [45]. Due to this gradual clock rising, programmers got used to see the computer as a serial machine and to wait for the next generation of processors and the inherent speedup in their applications.

Economical and physical limitations, such as exorbitant project prices, power consumption, heat dissipation and miniaturization, forced the computer industry to change their approach: performance gain of a processor would no longer be achieved by increasing the clock, but by adding more cores in a single chip [3,10]. Brodtrorb et al. [5] believe that the tendency to the future is a processor containing a large number of simple cores, known as manycore processors.

Manycore's properties mitigate some economical and physical limitations, and make this kind of processor is suitable for applications with a large number of threads [29]; but, according to Amdahl [2], code's performance is limited by its serial portion; Therefore, something is interesting regarding many cores, the system design possesses a faster core to accelerate code's inherent serial regions or regions with few threads [26]. To the use of different architectures to process distinct portions of the same code, to maximize the performance, it is given the name Heterogeneous Computing [34,39]; Asanovic et al [3] agree that Heterogeneous Computing is a trend not only for the high-performance computing, but to the general-purpose computing too, for, at least, the next thirty years.

The use of graphics processing units (GPU) to process general-purpose applications [35], in order to accelerate computationally intensive routines (GPGPU), has been the most highlighted heterogeneous architecture due to some factors: high rate of megaflop per dollar, low power consumption and the GPUs are, nowadays, easy to program [43].

These great shifts, recently faced by the computers industry, has been named as *"Concurrency Revolution"* [16,40]. The revolution is not only a hardware revolution but a software revolution too, since it drastically affected the whole creative process involved in the construction of algorithms, known as *"Computational Thinking"* [10,44].

This work explores the point of view in which programmers must learn how to *"think in parallel"*, and presents a new parallel procedure designed to process combinatorial branch-and-bound (B&B) algorithms by using GPUs. B&B is a methodological type of complete algorithms of paramount importance to academy and industry; they are mainly designed to search for optimal solutions to hard combinatorial problems. Serial and even parallel B&B algorithms must be reformulated in order to take advantage of new emergent parallel architectures, since improvements in processors' computational power will no longer mean clock frequencies increasing.

In some scenarios, it's too expensive to get tight bounds for a combinatorial problem; but, to prove the optimality of a problem, it is necessary to evaluate the whole solution space [36,47]. Once this space could be huge, and due to the B&B's inherent parallel design [1,23], B&B algorithms has been parallelized with great success along the years; but, according to [12,19,20], little attention has been given to the study of B&B in massively parallel environments like in GPGPU.

GPUs are, nowadays, easier to program, but GPU's architecture has so many peculiarities that not all kinds of applications are suitable to be executed in this kind of hardware. The main objective of this paper is to show that even very complex algorithms, such as branch-and-bound search algorithms, can be adapted to be executed by a GPU, in such a way that the high throughput of this kind of hardware can be exploited, and to present details about a new GPGPU parallel schema for combinatorial B&B algorithms. The schema involves so elementary concepts such way that any combinatorial B&B algorithm based on DFS could be easily adapted, taking advantage of the high GPU's throughput. The schema was tested to prove optimality of asymmetrical travelling salesman problem (ATSP)'s instances, and showed to be far superior to the serial and multicore versions of the same DFS-B&B algorithm, mainly in B&B's worst case.

The remainder of this paper is organized as follows. Section 2 presents a brief introduction about branch-and-bound method. Section 3 presents the proposed GPGPU B&B schema. Section 4 presents details about how the implementation was performed. Section 5 presents details about the computational evaluation of the method by using serial, manycore and multicore approaches. In Section 6 conclusions and future remarks are considered.

2 Branch-and-bound search algorithms

The combinatorial optimization, when compared to other areas of knowledge, is indeed young. But, because of its great number of problems applications, this matter is, nowadays, one of the most active topics in operational research and number theory [22,36].

Due to the combinatorial optimization problems' relevance, a great number of algorithms were created to solve them. These algorithms can be divided into complete and approximate methodologies [4]. Complete algorithms are those that guarantee to find an optimal solution to the instance at hand, in a certain amount of time. Among the complete algorithms, the branch-and-bound method [25,31], whose origin dates back to the fifties, is the most popular way of solving combinatorial optimization problems to optimality [46, 48].

The B&B method is not a specific algorithm, but a class of; all implementations of the B&B method largely depends on the problem at hand, but they all follow three main concepts: branching, a step that breaks the problem in sub problems; bounding, concerning the use of upper and lower bounds to guide the search strategy and evaluate subproblems and, finally, the step is known as pruning, which represents the act of eliminating large portions of the search space.

Generally, to determine how nodes will be branched, B&B algorithms use Depth-first search (DFS) techniques, where the most recently generated sub-problem is explored first, or Best-first search (BeFS), breadth-first search plus evaluation, where the most promising sub-problem is explored first.

According to Grama and Kumar [13] and Van Le [42], the BeFS is optimal in the number of nodes branched, once the BeFS does not process nodes with a lower bound greater than the optimal solution. But, to prove the optimality, this search strategy needs to keep a large number of nodes in a priority queue, which makes the BeFS space complexity exponential in the search depth and suitable only for smaller or easier instances [30].

Depth-first search, on other hand, has space complexity linear in the search depth and can find new upper bounds quickly [46,48]; these new solutions may not be the optimum, but these new upper bounds make the bounds tighter, leading to a more effective pruning process, which makes DFS suitable to harder instances. The greatest flaw of DFS is the search can get stuck in unpromising branches and generate a B&B tree greater than BeFS one. However, closer is the initial upper bound (solution) to the global optima, smaller is the gap between the DFS B&B and the BeFS tree size [7], mitigating this problem.

3 The GPGPU branch-and-bound schema

In a B&B algorithm, the search starts at a root node. This node is an incomplete solution and, generally, is a representation of a lower bound to the optimal solution. The starting node is kept in the *Active Set*, a set responsible to keep nodes evaluated, but not yet branched. Bounding procedures are responsible for this evaluation. As mentioned in Section 2, the chosen search strategy is the one that determines how nodes belonging to the *Active Set* are branched. A graphical schema of a B&B algorithm can be seen in Figure 1.

GPUs are massively parallel processors, and they are supposed to execute a large number of threads concurrently, each thread doing simple operations. To generate a considerable number of threads, the GPGPU B&B procedure starts serially, populating the *Active Set*, until the set has so many nodes as desirable.

In our schema, each node belonging to the *Active Set* will be a concurrent depth-first branch-and-bound (DFS-B&B, Figure 2) root. This DFS-B&B will be performed by each thread, which means that each thread will be responsible to evaluate a small portion of the search space. In order to evaluate the search space, each thread must have its *Active Set*, be aware of lower and upper bounds, branching rules, and bounding procedures; i.e., these procedures

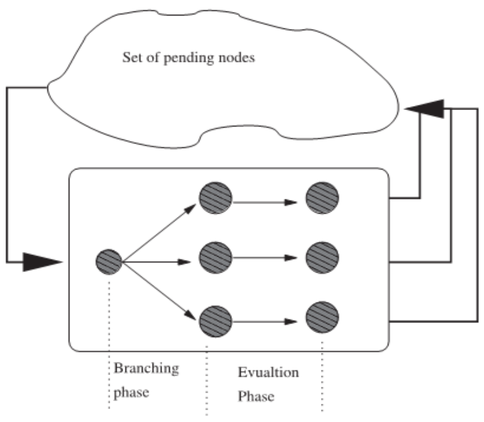

Fig. 1 Crainic, Le Cun and Roucairol's (2006) representation of a B&B algorithm.

must be implemented as device functions (for more details about GPGPU implementation, see [21,35,38]).

```
current_best_solution = ∞;
place the root node on the top of the stack;
while stack is not empty do
    select a node from the top of the stack;
    if selected_node is not a solution then
        evaluate the best possible solution this node can lead to:
        assign this value to node_bound;
        if node_bound < current_best_solution then
            generate successors of selected node;
            install generated successors into the stack;
        end
    else
        if solution_cost < current_best_solution then
            current_best_solution = solution_cost;
        end
    end
end
```

Fig. 2 Grama and Kumar [13] depth-first branch-and-bound algorithm.

Unlike the general-purpose processors, each GPU's stream processor is a simple unity and runs in a frequency lower than mainstream general-purpose processors [5,26,32]. Thus, labored branching rules and bounding procedures could affect the GPU portion's performance.

As a way to mitigate this GPU limitation, new nodes will be generated adding the next element in the sub-solution lexicography, following DFS's properties. These newly generated nodes are then evaluated by a simpler bounding procedure than the serial's portion procedure. If the problem demands a huge computational power to prove the optimality, the kernel (procedure executed by the GPU) can be executed several times; each time producing new nodes to the *Active Set* and updating the bounds. Each time the kernel is called, more threads could run concurrently and subproblems would be more restricted than the subproblems used as DFS-B&B roots by the concurrent searches in the previous kernel execution.

Threads are arranged in blocks of threads. Block's size can be statically determined by the programmer, or based on GPU's properties and characteristics of the instance to be solved. Block's size and the number of threads are parameters to the kernel launch. Initially, the number of threads launched is the size of *ActiveSet*. If this number is greater than the number of threads the GPU can handle or is handleable, but the execution is too heavy to the GPU, leading to poor performance, the *Active Set* must be portioned and the kernel called several times.

If each thread explores the solution space based on its own upper bound, the resulting B&B three could be greater than the serial one [7], i.e., a lot of unnecessary work would be done. In other hand, According to Brodtkorb et al. [5] and Kirk and Hwu [21], synchronization between threads is something costly in GPU computing. To deal with this trade-off in our schema the threads are divided into blocks, and each block has its own upper bound shared by its threads (which is quite faster in the GPU architecture). Threads evaluate the solution space based on the block's upper bound, and blocks synchronize upper bounds among them less often, as shown in Figure 3.

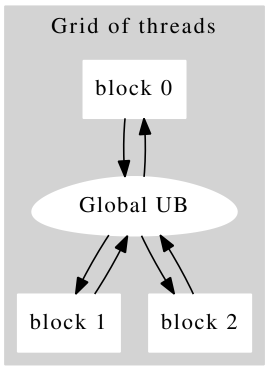

Fig. 3 Blocks of threads share upper bounds through a global upper bound.

To avoid synchronization of upper bound values and to produce small B&B trees, the initial upper bound, called z in Figure 4, must be started by a good heuristic or polynomial-time approximation scheme, which guarantee the quality of the upper bound produced [36].

After kernel execution, the block's master thread (thread's *inblock* identification equals to zero) will inform the best solution found by the block and,

```
I = get_instance( );
p = get_gpu_properties( );
z = calculate_initial_ub(I);
l = calculate_initial_lb(I);
A = generate_initial_active_set(p,z,l);
t = calculate_no_threads_per_block(p,A);
b = calculate_no_blocks(t);
allocate_instance_on_gpu(I);
while A is not empty: do
    S = a subset of A;
    A = A − S;
    allocate_data_on_gpu(S);
    concurrent_DFS_BnB<<<b,t>>>(z,l,S); /* Kernel            */
    synchronize_gpu_cpu_data(z,l);
end
return z;
```

Fig. 4 Pseudo-code of the proposed GPGPU B&B schema.

if the search is set up this way, will put new nodes into the *Active Set*. If necessary, a new number of threads and blocks would be calculated, in order to fit the new properties of the updated *Active Set*.

This procedure will produce B&B trees generally greater than the same search applying labored branching rules and bounding procedures. The strength of the proposed GPGPU schema is to take advantage of great segmentation of the search space, which allows hundreds of tiny pieces of the solutions space to be evaluated concurrently and in less time.

4 Implementation of the proposed GPGPU B&B schema for solving the ATSP

This section presents details about two implementations of the proposed B&B schema presented in Section 3: a complete enumeration and a DFS-B&B, both designed to prove optimality of ATSP's instances.

According to Zhang [47], in the worst case, a B&B algorithm must evaluate the whole solution space to prove the optimality. This behavior is observed when bounds are not tight [8]. Hence, the first implementation will enumerate all possible solutions of an ATSP instance. The objective of this implementation is to eliminate the effect of pruning and to verify whether our GPGPU schema can be more efficient than a multicore CPU implementation, despite GPU's lower clock and simpler processors unities.

The second implementation is a manycore DFS-B&B, designed to prove the optimality of ATSP instances by performing implicit enumeration. This implementation aims to verify whether branch-and-bound search algorithms can be adapted to be executed by a GPU, in such a way that the high throughput of this kind of hardware can be exploited. We also could observer the interference of pruning in massively parallel implementation of B&B where the upper bound is shared.

Both implementations are quite similar and have two main steps: Active Set creation and concurrent search. The only difference between them is the first one, complete enumeration, does not prune subtrees, i.e., the only node evaluation performed on the first implementation is to verify whether a node is a feasible solution or not.

The *Active Set* population for both implementations is better explained in the following section.

4.1 Active Set population

In both implementations, the serial portion starts with the *Ative Set* containing only a root node at level zero. A node, in this context, represents an incomplete Hamiltonian Cycle and its properties; a node at level x represents an incomplete Hamiltonian Cycle containing the start city (level zero) plus x other different cities. A feasible solution is found whenever a Hamiltonian Cycle is formed, i.e., when the search reaches level N, where N is the number of cities of the instance.

The *Active Set* population is responsible to segment the search space in tiny little pieces in such a way that GPU's massively parallel hardware can handle the B&B search procedure. To produce this population, the limit level (L) of the *Active Set* nodes ought to be informed. Figure 5, for instance, represents an *Active Set* population for a *4x4* ATSP instance where 2 was taken as limit level ($L = 2$) in order to populate the *Active Set*.

For the complete enumeration implementation, if $L = x$ and instance's size is n, then, the *Active Set* will contain exactly $\frac{(n-1)!}{x!}$ nodes, since there is no prune. Each one of these nodes represents an incomplete cycle containing the start city plus L different cities. This number of possible nodes in level L is an upper bound to the number of nodes present in DFS-B&B implementation's *Active Set*, since prune of subtrees may occur.

In Figure 5 the *Active Set* is in grey, and each box is a node. Nodes are labeled with their prefixes, i.e., with the cities sub-sequence which indicates the initial visiting order.

As shown in Figure 6, each node i belonging to the *Active Set* will be a DFS-B&B root R_i, that will be used by thread T_i, that will explore the region S_i of the search space. It is important to emphasize that every solution for node R_i is a valid solution to the problem [36].

4.2 The kernel

The DFS-B&B implementation performed by the kernel, in both algorithms, is an implementation of Grama and Kumar [13] DFS-B&B (Figure 2). The search was designed in such a way that no dynamic data structure was necessary to

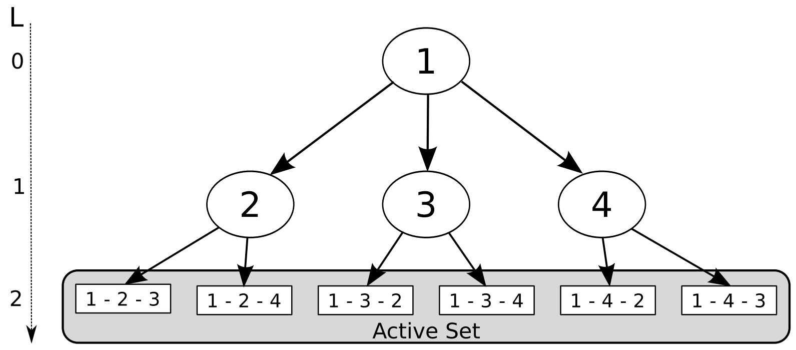

Fig. 5 Initial portion of the code that populates the Active Set.

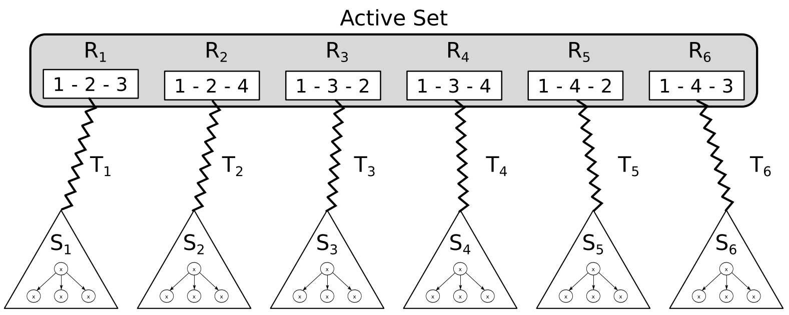

Fig. 6 Each node in Active Set will be a concurrent search root.

represent the *Active Set*, which is a stack in Figure 2; This strategy avoids the use of the GPU's slow global memory and takes advantage of fast thread's local memory.

In GPGPU DFS-B&B implementation, each concurrent DFS-B&B initial cost is the root's lower bound. As mentioned in Section 3, each thread performs DFS-B&B based on blocks' upper bound. This upper bound is started by the master thread with the upper bound value passed by the serial region as a kernel parameter.

At a specific point of the algorithm, the block's master thread verifies the global upper bound value to ensure the block is performing B&B with the lowest possible upper bound. If the global upper bound is greater than the block's upper bound, the block's master thread updates the global upper bound, if it is lesser, updates its own upper bound value. The kernel is launched only once, and the number of threads launched is the size of *Active Set*.

At the moment the block has finished its evaluation, the block's master thread informs the best solution found by the block, and the CPU portion sorts the vector containing all solutions returned by the kernel in order to identify the optimal one, as shown in Figure 7.

In GPGPU complete enumeration, each concurrent search cost starts with the node's cost, i.e., the cost of the incomplete Hamiltonian Cycle the node

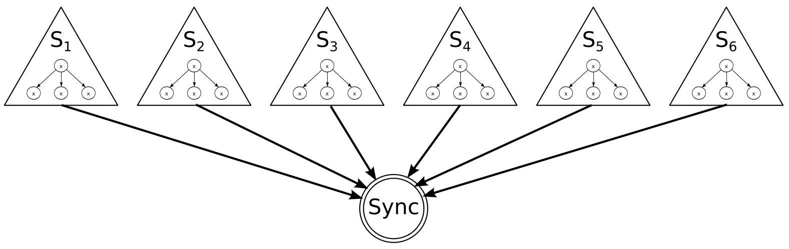

Fig. 7 After the concurrent search, all threads will inform the amount of solutions found and the best one.

represents. Since the instances this implementation is about to solve are small, when the kernel is launched, the cost matrix is copied to block's fast shared memory. This strategy speedups the access to edge values present in the cost matrix, avoiding access to slow global memory. After the kernel execution, each thread returns the amount of found solutions and the best one. The serial portion performs the reduction and informs the optimal solution and the total amount of found solutions.

4.3 Aspects of implementation

Both implementations of the proposed GPU Computing B&B schema were implemented with *C++* and parallelized by using NVidia's CUDA *4.0*.

Other four algorithms were conceived: a serial complete enumeration, serial DFS-B&B, a multicore DFS-B&B, and a multicore complete enumeration. Serial DFS-B&B is a loyal implementation of Grama and Kumar [13] depth-first branch-and-bound algorithm. Complete enumeration is the same algorithm, but do not prune subtrees. In order to provide a fair comparison further, multicore implementations are quite similar to the manycore (CUDA) implementations and were conceived as follows.

At the starting city, the traveling salesman has $N - 1$ possible paths to follow. In OpenMP implementations, this $N - 1$ paths are independent DFS-B&B, and this work is divided by all system's processors. As for the CUDA implementation, all independent DFS-B&B synchronize upper bounds at a certain part of the code, in order to avoid unnecessary evaluations.

Serial and multicore algorithms were implemented with *C++*, compiled by *GCC* [14] 4.6.2. *GCC's -O2* code optimization flag was employed. Multicore implementations were parallelized by using OpenMP API 3.0 [9] provided by GCC 4.6.2.

All DFS-B&B implementations (serial, multicore and manycore) use Martello and Toth's [27] primal-dual implementation of the Assignment Problem (AP) as lower bound and perform implicit enumeration following Tucker [41] schema. As an initial upper bound (solution), all three implementations employ Karp's [18] modified patching procedure, which uses the result of AP, i.e., a set of sub-cycles, to form a Hamiltonian cycle. This decision was made once Karp's patching constructive heuristic generally returns a good solution to the ATSP [6,11] and due to convenience, once patching's main step, the AP, was already available. These three DFS-B&B implementations do not consider the initial cost matrix, but a reduced matrix build with AP's dual variables.

In manycore DFS-B&B, since the instances this implementation is about to solve are greater than complete enumeration's, the cost matrix is placed on texture region; the texture region is slower than block's shared memory, but is faster than global memory. For more details about NVidia's GPUs memory hierarchy, see [33].

The limit depth of the search space that must be evaluated, in order to create the *Active Set*, in all manycore implementations, was statically set. In complete enumeration, instances with size five to seven cities had $L = 3$. Instances greater than seven, $L = 5$. In manycore DFS-B&B, the limit depth was set $L = 7$.

5 Computational evaluation

GPUs are nowadays easy to program, but GPU's architecture has so many peculiarities that not all kinds of applications are suitable to be executed in this kind of hardware. The objective of this section is to verify if branch-and-bound search algorithms are suitable to be executed by a GPU, exploiting GPU's high throughput.

To do this verification, two computational experiments were performed: complete enumeration and B&B search. In each experiment, three implementations of the same algorithm were compared: serial, multicore (OpenMP) and manycore (CUDA). All implementations will prove optimality of asymmetrical traveling salesman problem's instances, and data collected in each experiment is the time necessary by the implementation to experiment. It is important to empathize that the whole application time, not only the time of the concurrent process (in parallel implementations) is considered, since parallel applications have the overhead of thread creation, synchronization, convergence, etc., and this overhead time must be considered too [28].

The traveling salesman problem (TSP) consists of finding the shortest Hamiltonian cycle through a given number of cities, in such a way each city is visited only once. It is the most well-known and studied combinatorial optimization problem of history, and has plenty of real-world applications[15,24, 36]. The versions of the TSP can be defined as symmetric, if the cost matrix is symmetric ($\forall[i,j] : c_{ij} = c_{ji},$), asymmetric otherwise ($\exists[i,j] : c_{ij} \neq c_{ji}$).

The asymmetrical case is a more general way of representing TSP, and its instances are frequently harder to solve than the symmetric case's instances [6,17,50,49]. Although we made this choice for the proposal of this study; the method can be extended to other important combinatorial problems like: knapsack, coloring, covering, max satisfiability, etc.

The parameter employed to compare the implementations is the speedup. The speedup means the benefit of solving a problem in parallel, i.e., how many times the parallel code using n processors is faster than the serial code doing the same job [1], and is represented as follows:

$$S(n) = \frac{t_s}{t_n} \tag{1}$$

where t_s is the serial time, and t_n the parallel time with n processors.

The experiments were conducted on an Intel Core i7 2600 (3.4 GHz, 3.8 GHz when over demand)[1], with four cores, but each core can run two simultaneous threads; thus, the operating system (OS) sees i7 2600 as having eight cores; the system has 4 GB RAM and Ubuntu GNU Linux 11.10 OS (kernel 3.0.1).

The GPU used to run CUDA codes was a NVidia GeForce GTX 580[2]. GTX 580 is a compute capability 2.1 GPU, has 512 stream processors, each stream processor at 1,544 MHz; 1.5 GB of GDDR5 RAM at 2.004 GHz; 384 bits memory interface.

5.1 Computational experiment: complete enumeration

In this experiments, each implementation will perform a complete enumeration, i.e., enumerate all possible solutions of a randomly generated ATSP instance, each instance of size five to sixteen cities. The solution space that each implementation must evaluate is $(N-1)!$ solutions, where N is the number of cities of the instance. The ATSP instances used in this section were constructed by using *glibc*'s *rand( )* function, where each element $r_{i,j}$ belonging to the R_{NxN} matrix can vary in the interval [0,1000].

Figure 8 shows the time required by each implementation (serial, OpenMP and CUDA) to perform complete enumeration. This graphic is in logarithm scale in order to provide better visualization.

In Figure 8, CUDA implementation had equivalent times for five to ten cities. The search space for these instances is small, and there is considerable overhead in CUDA implementation, such as initial *Active Set* population, call GPU driver and serial reduction after kernel execution. For these instances, the overhead time is greater than the time required to process few levels of the solution space in parallel, and corresponds to the major part of the time measured. This behavior is present in OpenMP implementation too, but in five to nine cities, and in a smaller scale.

Figure 9 shows the speedup obtained by OpenMP and GPU implementations while solving complete enumeration.

When the solution space is huge (instances eleven to sixteen), CUDA implementation faces the ideal scenario: lots of independent tasks with no communication among them, no slow global memory access (the cost matrix was

[1] More details about this processor can be found at: `http://ark.intel.com/products/52213/Intel-Core-i7-2600-Processor-(8M-Cache-3_40-GHz)`

[2] More details about GTX-580 can be found at: `http://www.nvidia.com/object/product-geforce-gtx-580-us.html`

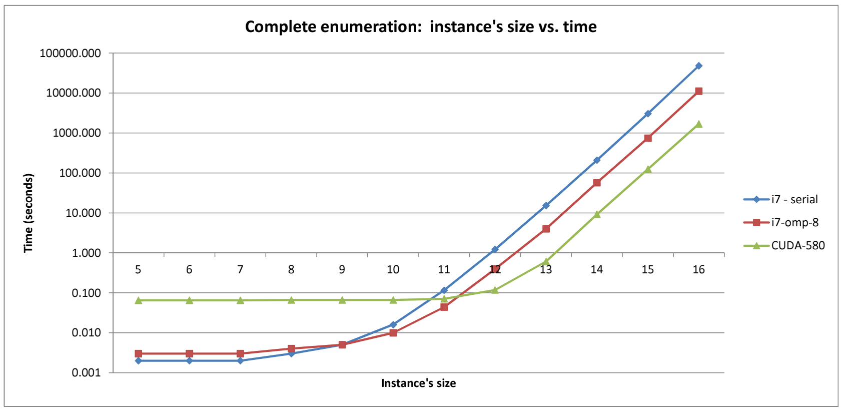

Fig. 8 Time required, in logarithm scale, by each implementation to perform complete enumeration.

placed into the blocks memory), leading to speedups of ten times in size twelve, and speedups of more than twenty times in sizes thirteen to fifteen, reaching the speedup of twenty-eight times in the instance of sixteen cities.

OpenMP implementation was pretty constant and for sizes twelve to sixteen, and obtained speedup between four to five times, showing that enumerate all possible solutions is heavy-duty even for a high-end multicore processor using four cores at high clock frequency and running two simultaneous threads.

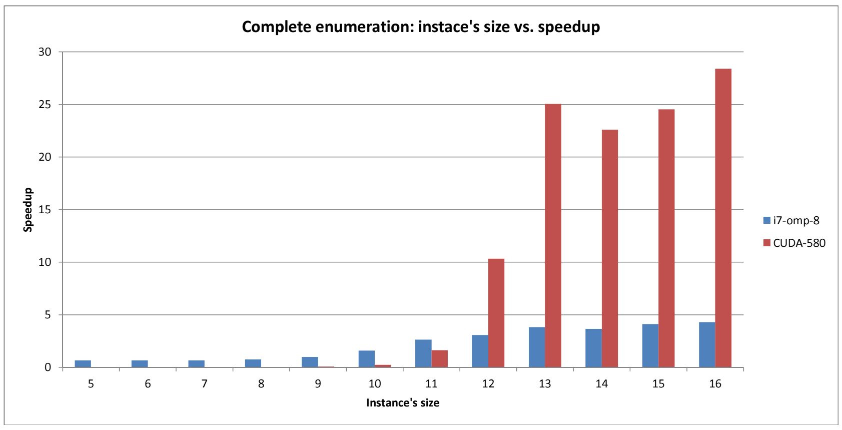

Fig. 9 Speedup obtained by the OpenMP and CUDA implementations performing complete enumeration.

5.2 Computational experiment: B&B search, hard ATSP instances

In this section, each implementation will prove the optimality of ATSP instances performing implicit enumeration (B&B search). Eleven hard instances were collected from the literature. This choice was made because merely random asymmetrical instances are easy to solve [11,30] and have no relation to real-world applications [17].

An instance can be considered hard when bounds obtained are not tight, or there are so many solutions close to the optimal solution, which makes the solution space to be evaluated to prove optimality huge, even for small instances.

The selected instances are Coin (14 to 18 cities), Flow (14 to 17 cities), Atex1 (16 cities) and BR17 (17 cities). Coin and Flow instances were made by Cirasella et al. [6]. This generator generates instances using properties found in real-world situations. The instances *coin* are those of collecting money from payphones in a grid-like city, and the instances *flow* are those from no-wait flow shop for processing of heated materials that must not be allowed to cool down. More details about the selected classes of instances can be found at [6, 17]. *Atex1* is a test instance from Cirasella's generator, and BR17 is a hard instance from the TSPLIB [37].

Figure 10 shows, on logarithmic scale, the time required by each implementation (serial, OpenMP and CUDA) to prove optimality of selected instances by performing a B&B search. Figure 11 shows the speedup obtained by OpenMP and CUDA implementations.

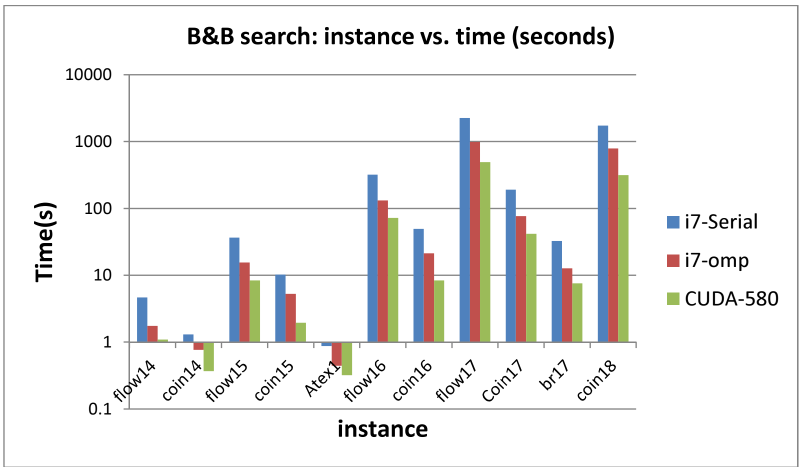

Fig. 10 Time required, in logarithmic scale, by each implementation to perform B&B search.

B&B search is a much more complex scenario than the previous section's scenario (complete enumeration). First, each generated node must be evaluated in order to verify if its lower bound is greater than the current block's upper bound. These prunes lead to divergent branches inside a warp[3], a situation much less often observed for the complete enumeration as all threads are performing the same operation. Second, in B&B there is the need for upper bound synchronization among threads by using block and global memory and, as the search is performed, lots of threads tend to die, once the whole subspace the thread is about to evaluate is pruned or completely evaluated, making the process inefficient. In this case, CUDA implementation can't provide sufficient work to the GPU. This is the main reason complete enumeration's speedup is much greater than B&B speedup.

Despite the adversity faced by CUDA implementation in this scenario, the implementation was pretty constant on the experiments, always dominating OpenMP implementation, and got speedups between four and five for the great majority of selected instances. *Coin* instances are the hardest in the set, once for all instances, the lower bound obtained by AP is always zero. Despite the hardness of these instances, they got the highest speedups of the testbed. Therefore, harder is the instance, more suitable to be processed by a GPU it is, once the threads will take a longer time to die, there will be few upper bound synchronizations, or these synchronizations will occur late on the B&B process.

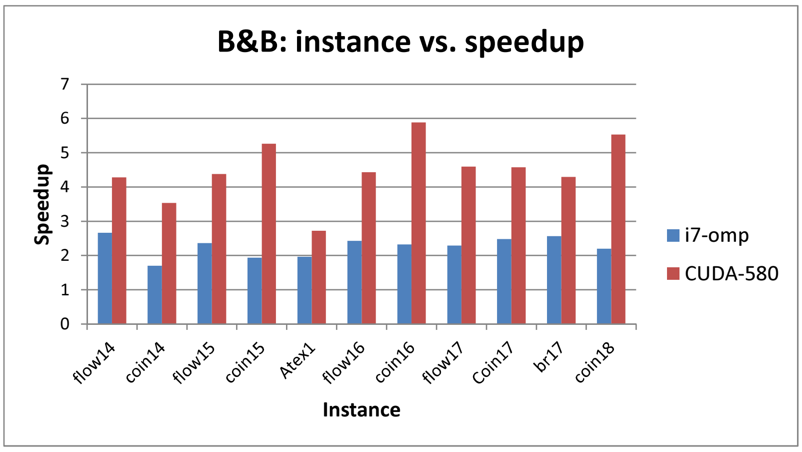

Fig. 11 Speedup obtained by OpenMP and CUDA implementations while performing B&B search.

In order to get higher speedups, CUDA implementation must be improved by using a load balance schema. When the kernel is launched, there is no such way to reorder the threads dynamically and balance the load among the alive threads. So, this load schema could be a trigger that halts the kernel when

[3] Threads are executed in groups of 32 threads called *warps*. Threads in a warp are supposed to execute the same instruction at the same time. Divergent branches inside a *warp* are executed separately. This situation can lead to loss of parallel efficiency [21,33].

a certain percentage of the original amount of threads are dead, such as the schema for load balance in SIMD present in [20]. Kernel's alive nodes are given back to the CPU, which recalculates a new number of threads and blocks, then calls the B&B search again.

6 Conclusions and future remarks

This work presented a new GPGPU schema for B&B algorithms evaluated to prove optimality for ATSP's instances. The computational experiments revealed that, for this first experience using GPUs, there is room for the investigation of better search schemas for GPGPU B&B implementations. Surprising results on time and speedups were obtained, where GPUs got much better results than a high-end multicore processor, mainly in the worst case. The high rate of megaflop/dollar is the main attraction of designing GPGPU based algorithms, even for very complex algorithms like B&B.

Results showed that harder is the instance, more suitable to be processed by GPU the instance is; when the instance has loose bounds, the serial portion will create the desired number of nodes quickly and the threads created to process will take a longer time to die, keeping the GPU busy and the process efficient. If the instance has tight bounds, the serial portion will generate fewer nodes than necessary or take a long time to produce the desired number of nodes. So, the inclusion of GPGPU elements into B&B algorithms can, due to GPUs' great throughput, make difficult unsolved combinatorial problems' instances to be solved to optimality, or to solve instances in less time than serially or in a shared memory environment.

According to Grama and Kumar [13], subtrees generated by DFS tend to be irregular, and work statically allocated tends to result in load imbalance among processors. Hence, during execution, lots of threads tend to be idle, once the whole search space the thread is supposed to evaluate is pruned or fully evaluated. To make effective use of available processors, workload balance schema is a critical issue when planning a parallel B&B algorithm. The current implementation has no trigger to determine when the process must be halted, blocks recalculated, and the remaining search space should be redistributed. Thus, at a certain time of the execution, warps may have few active threads. Therefore, a critical future work is to implement such a mechanism, based on Karypis and Kumar [20] triggering and redistributing schema for SIMD systems, to provide more efficient use of GPU's processors and, as a direct consequence, better speedups.

Classical artificial intelligence problems, such as N+K Queens Problem, are solved by finding a sequence of operations that can lead to a solution [42, 47], and are commonly solved by Backtracking. Another future direction is to apply the proposed search schema to solve artificial intelligence problems.

References

1. Aki, S.: The design and analysis of parallel algorithms. Old Tappan, NJ (USA); Prentice Hall Inc. (1989)
2. Amdahl, G.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference, pp. 483–485. ACM (1967)
3. Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., et al.: The landscape of parallel computing research: A view from berkeley. Tech. rep., Citeseer (2006)
4. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Computing Surveys (CSUR) **35**(3), 268–308 (2003)
5. Brodtkorb, A., Dyken, C., Hagen, T., Hjelmervik, J., Storaasli, O.: State-of-the-art in heterogeneous computing. Scientific Programming **18**(1), 1–33 (2010)
6. Cirasella, J., Johnson, D., McGeoch, L., Zhang, W.: The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. Algorithm Engineering and Experimentation pp. 32–59 (2001)
7. Clausen, J., Perregaard, M.: On the best search strategy in parallel branch-and-bound: Best-first search versus lazy depth-first search. Annals of Operations Research **90**, 1–17 (1999)
8. Crainic, T., Le Cun, B., Roucairol, C.: Parallel branch-and-bound algorithms. Parallel combinatorial optimization pp. 1–28 (2006)
9. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE **5**(1), 46–55 (1998)
10. Dongarra, J., Gannon, D., Fox, G., Kennedy, K.: The impact of multicore on computational science software. CTWatch Quarterly **3**(1), 1–10 (2007)
11. Frieze, A., Karp, R., Reed, B.: When is the assignment bound tight for the asymmetric traveling-salesman problem? SIAM J. Comput. **24**(3), 484–493 (1995)
12. Gendron, B., Crainic, T.: Parallel branch-and-bound algorithms: Survey and synthesis. Operations Research pp. 1042–1066 (1994)
13. Grama, A., Kumar, V.: A survey of parallel search algorithms for discrete optimization problems. ORSA Journal on Computing **7** (1993)
14. Griffith, A.: GCC: the complete reference. McGraw-Hill, Inc. (2002)
15. Helsgaun, K., Universitetscenter, R.: An effective implementation of K-opt moves for the Lin-Kernighan TSP heuristic. Citeseer (2006)
16. Hwu, W., Keutzer, K., Mattson, T.: The concurrency challenge. Design & Test of Computers, IEEE **25**(4), 312–320 (2008)
17. Johnson, D., Gutin, G., McGeoch, L., Yeo, A., Zhang, W., Zverovitch, A.: Experimental analysis of heuristics for the atsp. The traveling salesman problem and its variations pp. 445–487 (2004)
18. Karp, R.: A patching algorithm for the nonsymmetric traveling-salesman problem. SIAM Journal on Computing **8**, 561 (1979)
19. Karypis, G., Kumar, V.: Unstructured tree search on simd parallel computers: Experimental results. Innovative Applications of Massive Parallelism pp. 113–119
20. Karypis, G., Kumar, V.: Unstructured tree search on simd parallel computers. Parallel and Distributed Systems, IEEE Transactions on **5**(10), 1057–1072 (1994)
21. Kirk, D., Hwu, W.: Programming massively parallel processors: A hands-on approach. Published February **5** (2010)
22. Korte, B., Vygen, J.: Combinatorial optimization: theory and algorithms, vol. 21. Springer Verlag (2008)
23. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to parallel computing: design and analysis of algorithms. Benjamin/Cummings Pub. Co. (1994)
24. Laporte, G.: A short history of the traveling salesman problem. Canada Research Chair in Distribution Management, Centre for Research on Transportation (CRT) and GERAD HEC Montréal, Canada (2006)
25. Lawler, E., Wood, D.: Branch-and-bound methods: A survey. Operations research pp. 699–719 (1966)

26. Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., et al.: Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In: ACM SIGARCH Computer Architecture News, vol. 38, pp. 451–460. ACM (2010)
27. Martello, S., Toth, P.: Linear assignment problems. Surveys in Combinatorial Optimization pp. 259–282 (1987)
28. Mattson, T., Sanders, B., Massingill, B.: Patterns for parallel programming. Addison-Wesley Professional (2004)
29. McCalpin, J., Moore, C., Hester, P.: The role of multicore processors in the evolution of general-purpose computing. CTWatch Quarterly **3**(1) (2007)
30. Miller, D., Pekny, J.: Exact solution of large asymmetric traveling salesman problems. Science **251**(4995), 754 (1991)
31. Mitten, L.: Branch-and-bound methods: General formulation and properties. Operations Research pp. 24–34 (1970)
32. Nickolls, J., Dally, W.: The gpu computing era. Micro, IEEE **30**(2), 56–69 (2010)
33. Nvidia, C.: Compute unified device architecture programming guide. NVIDIA: Santa Clara, CA **83**, 129 (2007)
34. Olukotun, K., Hammond, L.: The future of microprocessors. Queue **3**(7), 26–29 (2005)
35. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: Gpu computing. Proceedings of the IEEE **96**(5), 879–899 (2008)
36. Papadimitriou, C., Steiglitz, K.: Combinatorial optimization: algorithms and complexity. Dover Pubns (1998)
37. Reinelt, G.: Tsplib—a traveling salesman problem library. ORSA Journal on Computing **3**(4), 376–384 (1991)
38. Sanders, J., Kandrot, E.: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional (2010)
39. Shalf, J.: The new landscape of parallel computer architecture. In: Journal of Physics: Conference Series, vol. 78, p. 012066. IOP Publishing (2007)
40. Sutter, H., Larus, J.: Software and the concurrency revolution. Queue **3**(7), 54–62 (2005)
41. Tucker, A.: Applied combinatorics, 3rd edn. Wiley (1994)
42. Van Le, T.: Techniques of Prolog programming: with implementation of logical negation and quantified goals. Wiley (1993)
43. Vetter, J., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Rogers, J., Roth, P., Spafford, K., et al.: Keeneland: Bringing heterogeneous gpu computing to the computational science community. IEEE Computing in Science and Engineering **13**(5) (2011)
44. Wing, J.: Computational thinking. Communications of the ACM **49**(3), 33–35 (2006)
45. Yu, A.: The future of microprocessors. Micro, IEEE **16**(6), 46–53 (1996)
46. Zhang, W.: Truncated branch-and-bound: A case study on the asymmetric tsp. In: Spring Symposium on AI and NP-Hard Problems, pp. 160–166 (1993)
47. Zhang, W.: Branch-and-bound search algorithms and their computational complexity. Tech. rep., DTIC Document (1996)
48. Zhang, W.: Depth-first branch-and-bound versus local search: A case study. In: proceedings of the National Conference on Artificial Intelligence, pp. 930–936. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999 (2000)
49. Zhang, W.: Phase transitions and backbones of the asymmetric traveling salesman problem. Journal of Artificial Intelligence Research **21**(1), 471–497 (2004)
50. Zhang, W., Korf, R.: A study of complexity transitions on the asymmetric traveling salesman problem* 1. Artificial Intelligence **81**(1-2), 223–239 (1996)